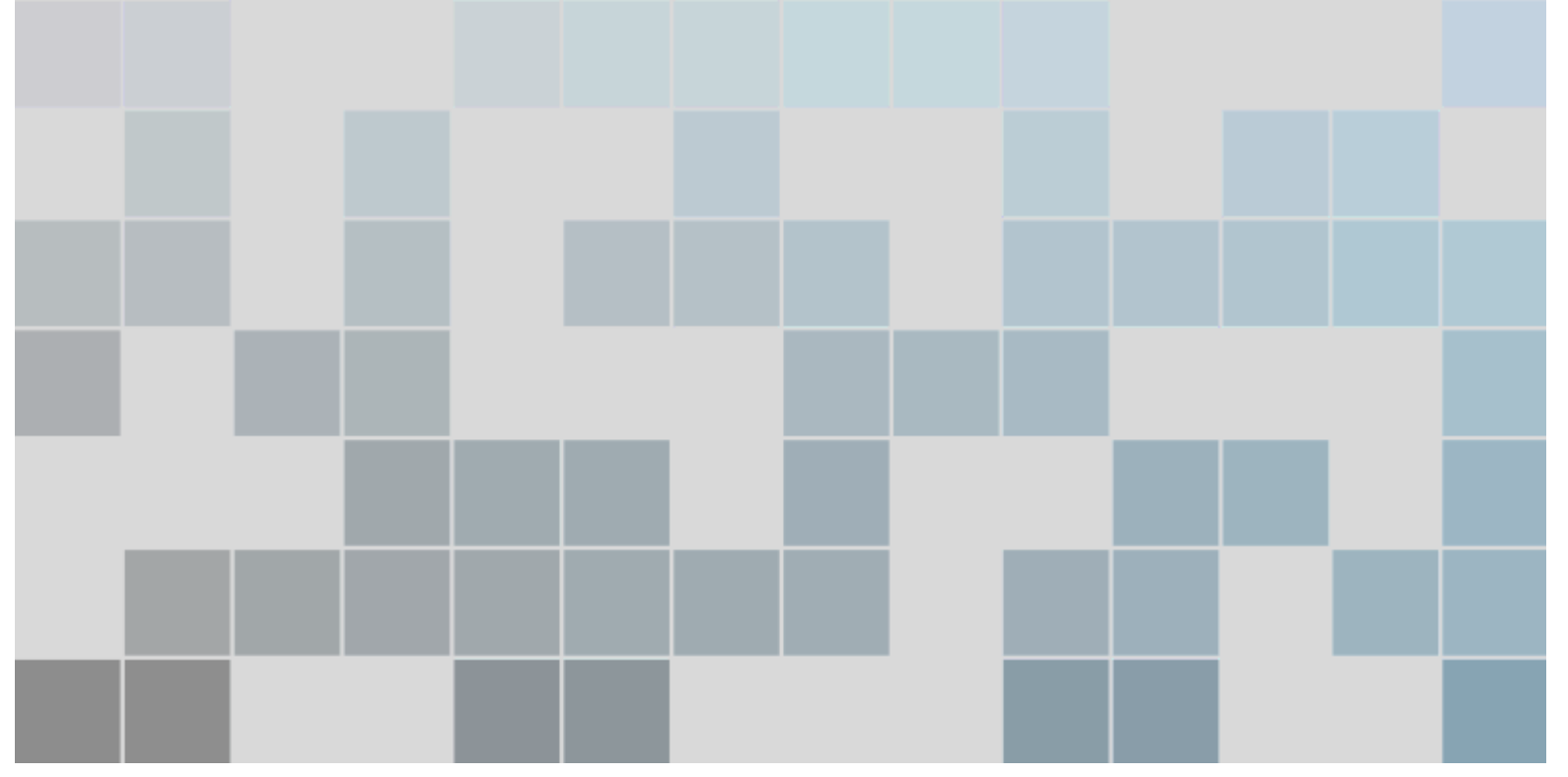




Introduction aux Langages Formels

Notes de cours 2015

G. Bianchi
A. Boussicault
T. Place
M. Zeitoun



Copyright © 2015 it2@labri.fr

PUBLISHED BY UB

WWW.LABRI.FR/PERSO/ZEITOUN/ENSEIGNEMENT/14-15/AS+IT2

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc/3.0>. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Printemps 2015

Table des matières

1	Motivations et Présentation	5
1.1	Objectifs du cours	5
1.2	Intérêt des langages formels	6
1.3	Vocabulaire et opérations sur les mots	7
I	Langages rationnels	9
2	Expressions et Langages Rationnels	11
2.1	Opérations sur les langages	11
2.2	Syntaxe des expressions rationnelles.	13
2.3	Sémantique des expressions rationnelles	14
3	Les Automates Finis	19
3.1	Limites des expressions rationnelles	19
3.2	Les automates finis	19
3.2.1	Qu'est-ce qu'un automate ?	19
3.2.2	Mots et langage acceptés	21
3.3	Automates Complets et Automates Déterministes	28
3.3.1	Automates Complets	28
3.3.2	Automates déterministes	29
4	Déterminisation des automates	33
4.1	Motivation et Présentation Informelle	33
4.2	Un algorithme de déterminisation	35

4.3	Propriétés de Clôture des Automates	37
5	Automates et Expressions Rationnelles	41
5.1	Des expressions vers les automates	42
5.1.1	La construction de Thompson	42
5.1.2	La construction de Glushkov	43
5.2	Des automates vers les expressions	43
6	Minimisation des Automates	45
	Bibliographie	47
	Index	49

1. Motivations et Présentation

1.1 Objectifs du cours

L'objet de ce cours est le traitement algorithmique des langages de mots. Dans notre contexte, un *mot* est simplement une suite de symboles. Ces symboles sont appelés *lettres*. L'ensemble de lettres utilisées pour écrire des mots s'appelle un *alphabet*, c'est un ensemble fini. Enfin, un *langage* est un ensemble de mots. Contrairement à l'alphabet, un langage peut être infini.

Définition 1.1.1 — Lettre, alphabet, mot, langage. Pour résumer, on utilise le vocabulaire suivant.

Lettre Symbole utilisé pour former des mots.

Alphabet Ensemble (fini) des lettres que l'on utilise. En informatique, on peut travailler avec divers alphabets suivant les motivations.

Mot Suite finie de lettres. Par exemple, si l'alphabet est l'alphabet usuel, le mot *informatique* est un mot de 12 lettres.

Langage Ensemble (fini ou infini) de mots.

Ainsi par exemple, à une langue comme le français ou l'anglais correspond un langage : l'ensemble de tous les mots de cette langue, écrits sur l'alphabet utilisé par cette langue (pour le français, cet alphabet comprend les lettres avec leurs versions accentuées, des ligatures æ et œ, ainsi que quelques symboles comme l'apostrophe ' ou le tiret -). Mais la notion de langage en informatique n'est pas restreinte à l'ensemble de mots de langues naturelles. En particulier, un langage peut avoir un nombre infini de mots (ce n'est pas le cas pour une langue comme le français ou l'anglais, qui ont seulement un nombre fini de mots).

Voici d'autres exemples d'alphabets, de mots et de langages. On note souvent A l'alphabet, parfois aussi Σ .

■ **Exemple 1.1** Si l'alphabet est $A = \{0, 1\}$, un mot sur cet alphabet peut représenter, par exemple, le codage en base 2 d'un nombre entier naturel. Par exemple, 111101111 est une suite de lettres, c'est donc un mot (qui vaut 2015 suivant le codage habituel en base 2). Sur cet alphabet, l'ensemble de tous les mots se note $\{0, 1\}^*$. On verra plus loin une justification de cette notation.

C'est un langage infini. Ses premiers mots, par ordre de longueur croissante, sont ε , 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, ... Deux remarques :

— Le mot ε est spécial : c'est le mot qui ne contient aucune lettre. On l'appelle le *mot vide*. Il

correspond à la chaîne vide que l'on note "" dans plusieurs langages de programmation.

- 0 et 1 sont à la fois des lettres, et des mots d'une seule lettre. En français, on a aussi des mots qui n'ont qu'une seule lettre, par exemple 'y' et 'a' (comme dans la phrase « Il y a très longtemps... »).

■ **Exemple 1.2** Sur le même alphabet, on peut considérer un langage plus restreint : l'ensemble de toutes les suites de 32 bits exactement. Par exemple,

00000000000000000000000000000000

est l'un de ces mots. Ce langage, cette fois, est fini : il contient 2^{32} mots. ■

■ **Exemple 1.3** Sur l'alphabet ASCII, on peut considérer un programme C comme un mot. Un tel mot peut être très long, mais comporte un nombre fini de caractères. Par exemple, le programme

```
1  int main(int argc, char *argv[])
2  {
3      return 0;
4  }
```

est un mot de 52 caractères (du premier, i, au dernier, le retour à la ligne). Certains de ces caractères, de l'alphabet ASCII, ne sont pas présents dans l'alphabet utilisé pour écrire les mots du français, comme par exemple l'étoile '*', ou le caractère *newline*, qu'on note '\n' dans les chaînes de caractères en C (mais qui n'est réellement qu'un seul caractère qui se visualise comme un retour à la ligne). L'ensemble des programmes C est donc un langage. Comme il y a des programmes C aussi longs que l'on veut, ce langage est infini (il contient un nombre infini de « mots », chacun d'eux représentant un programme). Ce langage ne contient pas le mot vide, car il faut une fonction main dans tout programme C. Le mot le plus court de ce langage est

```
1  int main(){return 0;}
```

qui a 21 lettres. ■

■ **Exemple 1.4** On peut aussi considérer des langages plus abstraits. Même si cela n'est pas évident à ce stade, on verra plus tard que cela a un intérêt. Par exemple, le langage vide \emptyset . C'est le langage qui ne contient aucun mot. Il ne faut pas le confondre avec le langage $\{\epsilon\}$, qui contient un seul mot : le mot vide. ■

1.2 Intérêt des langages formels

Pourquoi s'intéresser aux langages formels ? L'informatique est la science du traitement de l'information. Mais en informatique, les structures que l'on manipule sont souvent plus complexes que des mots. Il suffit de penser aux graphes, par exemple. Cependant, le mot est la structure la plus simple que l'on peut imaginer. Elle permet de modéliser de façon naturelle des structures de données classiques. Par exemple, on représente habituellement par un mot le contenu d'une pile.

Il y a plusieurs autres raisons de s'intéresser aux mots. C'est particulièrement naturel dans le traitement des langues, par exemple. Mais un mot peut aussi représenter des structures complexes. Prenons l'exemple d'un graphe.

Le graphe de la FIGURE 1.1 peut-être représenté par un mot. Par exemple, avec un codage évident qui liste d'abord les sommets puis les arêtes, dans un ordre arbitraire, par le mot

{1,2,3,4,5,6,7},{1,4}{4,5}{5,2}{2,1}{2,3}{1,6}{6,3}

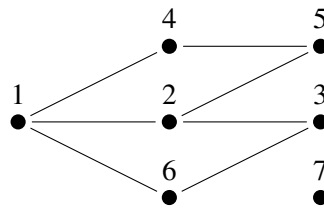


FIGURE 1.1 – Un graphe fini

sur l'alphabet contenant les chiffres, les accolades $\{$ et $\}$, et la virgule.

Les mots sont aussi utilisés en traitement de texte. Par exemple, de nombreux algorithmes ont été développés pour rechercher efficacement un motif dans un texte. Parmi les algorithmes classiques, le plus connu est celui de Knuth, Morris et Pratt, et sa variation donnée par Simon. De tels algorithmes ont de nombreux intérêts, par exemple en bio-informatique où l'alignement de longues séquences d'ADN est un problème important.

Parce que les programmes sont eux-mêmes des mots, il n'est pas non plus étonnant qu'ils soient à la base des algorithmes de compilation qui seront vus dans le cours d'analyse syntaxique.

Les mots et les outils qui ont été développés pour manipuler les langages ont de nombreuses autres applications. Un autre exemple d'importance est le domaine de la vérification automatique de logiciel.

Parce que les mots sont la structure de donnée la plus simple, il est important de bien comprendre les langages de mots, et développer des algorithmes pour ces langages, qui pourront ensuite être étendus à d'autres structures plus complexes, comme les arbres, par exemple.

1.3 Vocabulaire et opérations sur les mots

Voici un résumé de ce qu'on a vu plus haut, complété par quelques notations.

On a déjà vu qu'un alphabet est un ensemble fini de symboles, appelées lettres, et qu'un mot est une suite de lettres. Au lieu de noter cette suite en séparant les lettres, on la note en les collant les unes aux autres. Par exemple, sur l'alphabet $\{0, 1\}$, on écrit 1110 et non 1, 1, 1, 0 ou $(1, 1, 1, 0)$.

Le nombre de lettres d'un mot u est sa *longueur*, notée $|u|$. Par exemple, $|abc| = 3$. Il existe un seul mot de longueur 0, c'est le *mot vide*. Comme il est de longueur 0, il n'a pas de lettre. En informatique, c'est la chaîne vide qu'on désigne "" dans certains langages de programmation. Dans ce cours, on désigne le mot vide par ε . On a donc $|\varepsilon| = 0$.

Définition 1.3.1 — Monoïde libre. L'ensemble de tous les mots formés sur un alphabet A se note A^* . On dit que A^* est le *monoïde libre* sur A .

Un « début » de mot s'appelle un *préfixe* de ce mot. Par exemple, le mot *informatique*, qui a 12 lettres, a 13 préfixes. Les premiers, par longueur croissante, sont ε , i, in, inf, info, inform, etc. Symétriquement, un *suffixe* est un bloc de lettres consécutives à la fin d'un mot. Par exemple, les premiers suffixes du mot *informatique*, par ordre de longueur croissante, sont ε , e, ue, que, ique, tique, ... Enfin, un bloc de lettres consécutives dans un mot s'appelle un *facteur*. Parmi les facteurs, il y a les préfixes et les suffixes mais il y en a d'autres. Par exemple, *forma* est un facteur du mot informatique.

On peut *concaténer* 2 mots u et v , c'est-à-dire les coller l'un à la suite de l'autre. On obtient alors le mot uv . On peut parfois l'écrire $u \cdot v$ pour montrer qu'il s'agit de la concaténation de u et v . Par exemple, la concaténation de 1010 et de 11 est le mot $1010 \cdot 11$, ou 101011. On dit

aussi *produit* de deux mots à la place de *concaténation* de deux mots.

Définition 1.3.2 — Concaténation. La concaténation de deux mots u et v est le mot uv obtenu en faisant suivre la suite des lettres de u par la suite des lettres de v .

On peut remarquer que la concaténation de mots est une opération associative :

$$(u \cdot v) \cdot w = u \cdot (v \cdot w)$$


Cette propriété nous dispense de parenthéser les produits de mots : on écrit par exemple uvw . On remarque aussi que concaténer le mot vide à un mot produit le même mot :

$$\varepsilon \cdot u = u = u \cdot \varepsilon.$$

Formellement, le mot vide ε est l'élément neutre pour l'opération de concaténation.

On définit finalement ce que sont les langages.

Définition 1.3.3 — Langage. Un *langage* sur un alphabet A est un ensemble de mots dont les lettres appartiennent à A .

 **Remarque** Un langage peut être fini ou infini.

Première partie

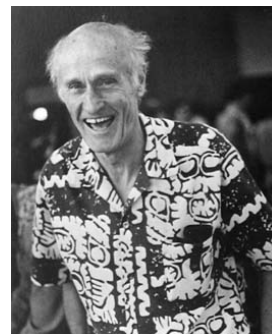
Langages rationnels

2. Expressions et Langages Rationnels

En informatique, on a besoin de décrire précisément, mathématiquement, les langages qu'on va manipuler. Idéalement, on voudrait

- des langages assez intéressants pour pouvoir exprimer des propriétés utiles,
- des langages faciles à décrire,
- des langages qu'on pourrait traiter automatiquement, par des algorithmes.

Une telle classe de langages a été définie par S. Kleene. Il s'agit des *langages rationnels*, aussi appelés *langages réguliers*. Un langage rationnel est défini en utilisant une *expression rationnelle*, qu'on appelle aussi *expression régulière*. Ces expressions sont un peu comme les expressions arithmétiques, mais au lieu de manipuler des entiers, elles manipulent des lettres pour définir des langages grâce à des opérateurs.



2.1 Opérations sur les langages

Avant de définir le langage représenté par une expression rationnelle, nous allons d'abord définir, puis expliquer plusieurs opérations sur les langages. L'objectif est toujours de pouvoir définir des langages à l'aide de nos briques de base.

Définition 2.1.1 — Opérations rationnelles. Il y a 3 opérations rationnelles : l'union, le produit et l'étoile.

1. La première opération est très simple : il s'agit de l'union de deux langages. Si L_1 et L_2 sont deux langages, alors, leur union $L_1 \cup L_2$ est le langage suivant :

$$L_1 \cup L_2 = \{u \mid u \in L_1 \text{ ou } u \in L_2\}.$$

2. La seconde opération généralise le produit de concaténation aux langages. Si L_1 et L_2 sont deux langages, alors, leur produit de concaténation $L_1 \cdot L_2$ est le langage suivant :

$$L_1 \cdot L_2 = \{u_1 \cdot u_2 \mid u_1 \in L_1 \text{ et } u_2 \in L_2\}.$$

3. Enfin, l'étoile d'un langage L est définie de la façon suivante :

$$\begin{aligned} L^* &= \{u_1 \cdot u_2 \cdot u_3 \cdots u_n \mid n \in \mathbb{N}, u_1, u_2, \dots, u_n \in L\} \\ &= \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \dots \cup L^n \cup \dots \\ &= \bigcup_{k=0}^{\infty} L^k. \end{aligned}$$

La première opération, l'union de deux langages, est très simple. Le langage $L_1 \cup L_2$ est simplement composé des mots qui sont soit dans L_1 , soit dans L_2 . Par exemple, $\{ab, ba\} \cup \{aaa, ab\} = \{aaa, ab, ba\}$.

La seconde opération est le produit, ou la concaténation de deux langages. Le langage $L_1 \cdot L_2$, noté aussi $L_1 L_2$, est composé de toutes les concaténations d'un mot de L_1 par un mot de L_2 . On a par exemple

- $\{a\}\{b\} = \{ab\}$.
- $\{a\}\{b, c\} = \{ab, ac\}$.
- $\{a, b\}\{b, c, dd\} = \{ab, ac, add, bb, bc, bdd\}$.
- Sur l'alphabet binaire $A = \{0, 1\}$, le langage $A^*\{0\}$ est l'ensemble des mots qui représentent un entier pair.

■ **Exemple 2.1** Un exemple plus compliqué : sur un alphabet A , prenons P le langage de tous les mots de longueur *paire* sur cet alphabet, et I le langage de tous les mots de longueur *impaire* sur cet alphabet. Par exemple, si $A = \{a, b\}$, alors $P = \{\varepsilon, aa, ab, ba, bb, aaaa, \dots\}$ et $I = \{a, b, aaa, aab, aba, abb, \dots\}$.

Quels sont les produits PP , PI , IP et II ? Pour PP , on remarque d'une part que lorsqu'on concatène deux mots de longueur paire, on obtient encore un mot de longueur paire. Cela se traduit par $PP \subseteq P$. Inversement, tout mot de longueur paire peut se décomposer en un produit de deux mots de longueur paire. Par exemple, si le mot u est de longueur paire, $u = \varepsilon \cdot u$ est bien le produit de deux mots de longueur paire. On en conclut que $P \subseteq P \cdot P$. Finalement, on a $PP = P$.

De la même façon, la concaténation de deux mots de longueur impaire est un mot de longueur paire. Donc $II \subseteq P$. Inversement, a-t-on $P \subseteq II$, c'est-à-dire, un mot de longueur paire se décompose-t-il en une concaténation de deux mots de longueur impaire ? Cela fonctionne effectivement dès que le mot a au moins une lettre. Un mot de longueur paire qui a au moins une lettre s'écrit $a \cdot u$, avec a sa première lettre et u le reste du mot, qui est bien de longueur impaire. Par contre, le mot vide ne se décompose pas en une concaténation de 2 mots de longueur impaire. On a donc $P \setminus \{\varepsilon\} \subseteq II$. On en déduit que $I \cdot I = P \setminus \{\varepsilon\}$.

Enfin, on montre de la même façon que $IP = PI = I$: la concaténation d'un mot de longueur paire et d'un mot de longueur impaire est un mot de longueur impaire, et inversement, n'importe quel mot de longueur impaire se décompose comme une concaténation d'un mot de longueur paire et d'un mot de longueur impaire. ■

Une fois que nous avons défini la concaténation de deux langages, on remarque que, comme pour les mots, les parenthèses sont inutiles lorsqu'on concatène plusieurs langages, car $(L_1 L_2) L_3 = L_1 (L_2 L_3)$: c'est le langage composé des mots qui sont une concaténation d'un mot de L_1 , d'un mot de L_2 et d'un mot de L_3 .

On peut donc définir la puissance d'un langage sans parenthéser :

Définition 2.1.2 — Puissance d'un langage.

$$L^n = \underbrace{L \cdot L \cdots L}_{n \text{ fois}}.$$

Par convention,

$$L^0 = \{\varepsilon\}.$$

Le langage L^n est donc l'ensemble de tous les mots qui peuvent se décomposer comme un produit de n mots de L .

- Exercice 2.1** 1. Soit $n \geq 1$ un entier et L est un langage qui contient ε . Justifier que $L \subseteq L^n$.
2. Donner un exemple de langage L montrant qu'on peut avoir $L^2 \subseteq L$ et $L^2 \neq L$.
3. Donner un exemple de langage L tel qu'on n'a ni $L \subseteq L^2$, ni $L^2 \subseteq L$. ■

On vient de définir la *puissance* L^n d'un langage L . Cela permet de définir son étoile. L'étoile d'un langage est l'ensemble des mots qui se décomposent comme une concaténation de mots de L , mais on ne fixe pas le nombre de mots dans cette concaténation : il peut y en avoir 0 (auquel cas on obtient le mot vide), seulement 1 (auquel cas on obtient un mot de L), ou plus. Par exemple, pour $L = \{a\}$, on a $\{a\}^* = \{\varepsilon, a, a^2, a^3, \dots\} = \{a^n \mid n \geq 0\}$. De même, si $L = A$ est le langage des lettres, alors L^* est l'ensemble de tous les mots sur l'alphabet A . Cela justifie *a posteriori* la notation A^* pour le monoïde libre.

- Exercice 2.2** À l'aide des opérations sur les langages, exprimer l'ensemble P des mots de longueur paire et l'ensemble I des mots de longueur impaire sur l'alphabet $A = \{a, b\}$, en partant des trois langages de base $\{\varepsilon\}$, $\{a\}$ et $\{b\}$. ■

2.2 Syntaxe des expressions rationnelles.

Les expressions sont définies de la façon suivante :

- Définition 2.2.1 — Expressions rationnelles.** 1. On part de « briques de base ». Ces briques sont les expressions rationnelles suivantes :
- (a) L'expression \emptyset ,
 - (b) L'expression ε ,
 - (c) Toutes les expressions a , où a est une lettre de l'alphabet.
2. À l'aide d'expressions rationnelles déjà construites, on peut construire des expressions plus complexes : si e_1 et e_2 sont des expressions rationnelles, alors
- (a) $e_1 + e_2$,
 - (b) $e_1 \cdot e_2$,
 - (c) e_1^*
- sont des expressions rationnelles.

On dit aussi *expressions régulières* au lieu d'*expressions rationnelles*.

Comme pour les expressions arithmétiques, on écrit souvent $e_1 e_2$ au lieu de $e_1 \cdot e_2$. De même également, le produit \cdot est plus prioritaire que l'addition $+$, et l'étoile $*$ plus prioritaire que le produit.

Ainsi par exemple, si l'alphabet est $A = \{a, b, c\}$, on peut construire les expressions suivantes.

- bc
- $a + bc$, qui doit se comprendre comme $a + (bc)$, puisque le produit est plus prioritaire que l'addition,
- $(a + b)c$, qui ne désigne pas la même chose,
- ab^* , qui doit se comprendre comme $a(b^*)$, puisque l'étoile est plus prioritaire que le produit.

- $(ab)^*$, qui ne désigne pas la même expression.
- $(aa)^*$,
- $a((b+c)^*ab)^*$, qui est une expression plus complexe que les précédentes.

2.3 Sémantique des expressions rationnelles

Chaque expression désigne un langage. La *sémantique* d'une expression est le langage que l'expression désigne.

1. On interprète ainsi les « briques de base » :
 - (a) L'expression \emptyset désigne simplement le langage vide.
 - (b) L'expression ε désigne le langage qui ne contient que le mot vide. C'est donc le singleton $\{\varepsilon\}$.
 - (c) L'expression a désigne le langage qui n'a qu'un seul mot : le mot a . C'est donc le singleton $\{a\}$.
2. Concernant les constructions $+$, \cdot et $*$, supposons que e_1 représente un langage L_1 et e_2 représente un langage L_2 . Alors
 - (a) l'expression $e_1 + e_2$ représente le langage $L_1 \cup L_2$, union de L_1 et de L_2 .
 - (b) l'expression $e_1 \cdot e_2$ représente le langage $L_1 \cdot L_2$.
 - (c) l'expression e_1^* représente le langage L_1^* .

Notation 2.1. On note $L(e)$ le langage désigné par l'expression e .

Par exemple, $L(ba^*)$ est le langage $\{ba^n \mid n \geq 0\}$, dont les premiers mots par ordre de taille sont b , ba , baa , $baaa$,...

Exercice 2.3 Trouver des expressions rationnelles pour les langages suivants sur l'alphabet $A = \{a, b, c\}$.

1. L'ensemble des mots de longueur 2.
2. L'ensemble des mots de longueur n .
3. L'ensemble de tous les mots.
4. L'ensemble de tous les mots sauf le mot vide.
5. L'ensemble des mots dont toutes les lettres sont identiques (soit que des a , soit que des b , soit que des c).
6. L'ensemble des mots qui commencent par un a .
7. L'ensemble des mots qui ont au moins un a .
8. L'ensemble des mots qui n'ont pas de a .
9. L'ensemble des mots qui ont un a et un b .
10. L'ensemble des mots qui ont un préfixe ab .
11. L'ensemble des mots qui ont un suffixe ab .
12. L'ensemble des mots qui ont un facteur ab .
13. L'ensemble des mots qui n'ont pas comme préfixe ab .
14. L'ensemble des mots qui n'ont pas comme facteur ab . ■

Solution 1. L'ensemble des mots de longueur 2 est décrit par l'expression $aa + ab + ac + ba + bb + bc + ca + cb + cc$, qui peut s'écrire $(a + b + c)(a + b + c)$, ou plus brièvement $(a + b + c)^2$. Comme l'alphabet est $A = \{a, b, c\}$, on admet A comme une expression rationnelle pour abréger $a + b + c$. L'ensemble des mots de longueur 2 est donc décrit par l'expression A^2 .

2. L'ensemble des mots de longueur n est décrit par l'expression A^n . En effet, $A^n =$

$\underbrace{A \cdot A \cdots A}_{n \text{ fois}}$ est, par définition de la concaténation de langages, l'ensemble des mots qui sont la concaténation de n lettres.

3. L'ensemble de tous les mots sur l'alphabet $A = \{a, b, c\}$ est décrit par l'expression $(a + b + c)^*$, qui s'écrit aussi A^* .
4. L'ensemble de tous les mots sauf le mot vide est $A^* \setminus \{\varepsilon\}$. Mais ceci n'est pas une expression rationnelle car on a utilisé l'opération \setminus , et on n'a droit qu'à $+$, \cdot et $*$. On peut écrire ce langage AA^* (en distinguant la première lettre), ou A^*A (en distinguant la dernière).

Plus généralement, on définit e^+ comme

$$e^+ = ee^* = e^*e.$$

Il ne faut bien sûr pas confondre ce symbole $^+$ en exposant avec l'opérateur binaire $+$ qui représente l'union.

Si e est une expression, e^* représente la somme (infinie) $\varepsilon + e + e^2 + e^3 + \dots$, donc $e^+ = ee^* = e(\varepsilon + e + e^2 + e^3 + \dots) = e + e^2 + e^3 + e^4 \dots$ désigne le langage des mots qui sont concaténations d'au moins un mot du langage représenté par e . On utilise cette notation aussi au niveau des langages :

$$L^+ = L \cup L^2 \cup L^3 \cup \dots \cup L^n \cup \dots$$

Par définition de l'étoile, on a

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \dots \cup L^n \cup \dots$$

et donc $L^* = \{\varepsilon\} \cup L^+$, ou au niveau des expressions,

$$e^* = \varepsilon + e^+.$$

5. L'ensemble des mots ne contenant que des a est a^* . L'ensemble des mots dont toutes les lettres sont identiques est donc $a^* + b^* + c^*$.
6. L'ensemble des mots qui commencent par a est décrit par l'expression aA^* .
7. Expression pour l'ensemble des mots qui ont au moins un a : A^*aA^* .
8. Expression pour l'ensemble des mots qui n'ont pas de a : $(b + c)^*$. À nouveau, on ne peut pas écrire $A^* \setminus (A^*aA^*)$, car cette expression utilise \setminus .
9. Une expression possible pour l'ensemble des mots qui ont un a et un b est $A^*aA^*bA^* + A^*bA^*aA^*$. Si on ne mettait que la première partie de cette expression, $A^*aA^*bA^*$, on oublierait par exemple le mot ba .
10. L'ensemble des mots qui ont un préfixe ab est abA^* .
11. L'ensemble des mots qui ont un suffixe ab est A^*ab .
12. L'ensemble des mots qui ont un facteur ab est A^*abA^* .
13. L'ensemble des mots qui *n'ont pas* un préfixe ab est $A^* \setminus abA^*$, qui n'est pas une expression valide. Une expression valide est

$$(b + c)A^* + a(a + c)A^* + a + \varepsilon.$$

En détail, un mot de ce langage :

- soit commence par b ou par c , et dans ce cas on peut avoir n'importe quel suffixe ensuite : ce cas est décrit par l'expression $(b + c)A^*$.

- soit commence par a . S'il y a une lettre ensuite, elle ne peut pas être b car cela formerait le préfixe interdit ab . Dans Ce cas correspond à l'expression $a(a+c)A^*$.
- soit commence par un a et n'a pas d'autre lettre après : ce cas correspond à l'expression a .
- soit est le mot vide : ε .

14. Pour ce cas plus compliqué, voici plusieurs expressions qui conviennent. L'expression

$$((b+c)^*(a^*c)^*)^*a^* \quad (2.1)$$

proposée en cours est bien correcte. Une expression alternative et un peu plus simple à analyser est

$$(b+c+a^+c)^*a^*. \quad (2.2)$$

On rappelle que a^+ signifie aa^* , qui désigne le langage des mots non vides n'ayant que des a . On voit que chacune de ces expressions interdit à une lettre a d'être suivie par un b : dans l'expression (2.1), a apparaît dans les sous expressions

- a^*c , qui représentent des mots de la forme $aaaa \cdots a\underline{a}c$: les a de ces mots sont suivis soit de a soit de c (pour le dernier a , souligné en gras ci-dessus),
- a^* en fin d'expression : les a correspondants sont suivis de a , sauf la dernière lettre.

Le même raisonnement fonctionne pour la seconde expression.

On vient donc de montrer que le langage décrit par ces expressions n'est formé que de mots ne contenant pas le facteur ab .

Pour s'assurer que ces expressions sont correctes, il faut aussi montrer qu'elles décrivent le langage de tous ces mots. Considérons donc un mot u sans facteur ab . On veut montrer que ce mot est dans le langage de l'expression (2.2), par exemple.

On peut écrire le mot u en mettant en évidence tous les blocs de a . Chacun de ces blocs est

- soit suivi d'un c (car un b est interdit, et un a ferait partie du bloc),
- soit le dernier.

Le mot se décompose donc en facteurs qui sont

- soit des b ,
- soit des c ,
- soit des blocs de a suivis d'un c ,

avec en fin de mot un éventuel bloc de a . Cela correspond bien à l'expression $(b+c+a^+c)^*a^*$. L'expression $(b+c+a^*c)^*a^*$ est une troisième expression correcte. ■

Les exercices suivants sont du même type, pour des langages plus compliqués.

Exercice 2.4 Trouver des expressions rationnelles pour les langages suivants sur l'alphabet $A = \{a, b\}$.

1. L'ensemble de tous les mots qui ont comme facteur aba .
2. L'ensemble de tous les mots qui n'ont **pas** comme facteur bab .
3. L'ensemble de tous les mots qui ont comme facteur aba mais qui n'ont **pas** comme facteur bab . ■

Exercice 2.5 Trouver une expression rationnelle sur l'alphabet des chiffres

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

pour les langages suivants :

- L'ensemble des mots qui représentent un nombre impair.
- L'ensemble des mots qui représentent un nombre divisible par 5.
- L'ensemble des mots qui représentent un nombre divisible par 3.



3. Les Automates Finis

3.1 Limites des expressions rationnelles

Les dernières questions des exercices 2.4 et 2.5 du chapitre précédent font ressortir plusieurs **questions ou remarques naturelles** :

1. Il n'est pas évident de trouver une expression rationnelle pour certains langages, ou même, de savoir s'il en existe une.
2. Quand on connaît une expression rationnelle e , il n'est pas évident de trouver une autre expression rationnelle qui décrit l'ensemble des mots qui **ne sont pas décrits** par e .
3. Lorsqu'on connaît deux expressions rationnelles, disons e_1 et e_2 , il est aussi difficile de trouver une expression rationnelle pour l'ensemble des mots qui satisfont à la fois e_1 **et** e_2 .

En fait, pour les deux derniers points, il n'est pas clair qu'on peut exprimer la « négation » d'une expression e , ni la « conjonction » de deux expressions e_1 et e_2 . Par contre, on a déjà à notre disposition la disjonction : par définition de $+$, l'expression $e_1 + e_2$ décrit exactement l'ensemble des mots qui satisfont *soit* e_1 , *soit* e_2 .

Dans ce chapitre, nous allons voir comment répondre à ces trois remarques. L'objet principal que nous allons introduire s'appelle *automate fini*.

3.2 Les automates finis

3.2.1 Qu'est-ce qu'un automate ?

Un automate est une machine qui prend en entrée des mots, et émet un « verdict » pour chacun d'entre eux. Chaque mot peut être :

- soit accepté,
- soit rejeté par l'automate.

De même qu'une expression rationnelle décrit un langage, un automate représente donc également un langage : l'ensemble de mots qui sont acceptés par l'automate. Avant de définir formellement les automates, nous allons voir plusieurs exemples pour illustrer la notion d'acceptation, de calcul et de langage accepté.

Un premier exemple d'*automate* est présenté sur la FIGURE 3.2. Sur cette figure, il y a trois **états**, indiqués par des cercles en bleu. On peut donner des noms aux états. Ici, on les a appelés 1, 2 et 3 (on peut bien sûr choisir d'autres noms, comme q_1 , q_2 , q_3 par exemple). Un état peut avoir l'une ou l'autre des deux caractéristiques suivantes :

- il peut être ou non **initial**. Les états initiaux sont indiqués par une flèche entrante. Sur la FIGURE 3.2, il n'y a qu'un seul état initial : l'état 1.
- il peut être ou non **final**. On dit aussi : **acceptant**. Les états finaux sont indiqués par une flèche sortante. Sur la FIGURE 3.2, il y a deux états finaux : les états 1 et 2.

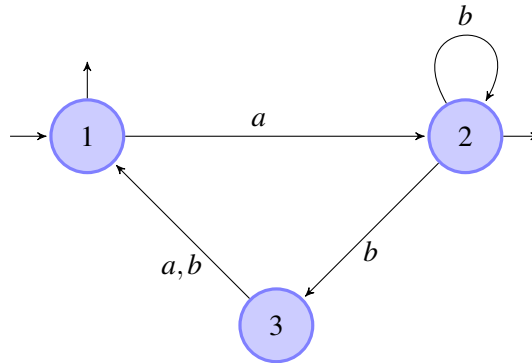


FIGURE 3.1 – Un premier exemple d'automate fini

Il existe une autre façon de représenter les états finaux : au lieu d'ajouter une flèche sortante, on double le cercle. Le même automate peut donc se représenter comme sur la FIGURE 3.2.

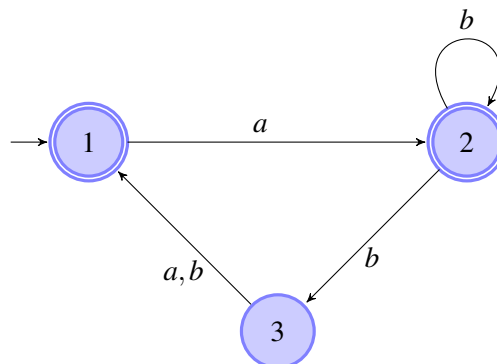


FIGURE 3.2 – Une autre représentation de l'automate de la FIGURE 3.1

Pour résumer, un état peut être à la fois initial et final (comme l'état 1), seulement initial, seulement final (comme l'état 2), ou ni initial ni final.

Un automate a aussi des « transitions ». Sur les figures précédentes, elles sont représentées par des flèches entre les états, et elles sont étiquetées par une ou plusieurs lettres de l'alphabet, ici $\{a, b\}$ (il se pourrait d'ailleurs que l'alphabet soit $\{a, b, c\}$ mais que c n'apparaisse sur aucune transition). Les transitions de cet automate sont les suivantes :

- $1 \xrightarrow{a} 2$,
- $2 \xrightarrow{b} 2$,
- $2 \xrightarrow{b} 3$,
- $3 \xrightarrow{a} 1$, et
- $3 \xrightarrow{b} 1$.

R **Remarque — Boucles et transitions multiples.** Concernant la FIGURE 3.1, on peut faire deux remarques :

- La transition $2 \xrightarrow{b} 2$ va d'un état à lui-même, on l'appelle une **boucle**.

- Sur la figure, la flèche $3 \xrightarrow{a,b} 1$, qui comporte 2 lettres, est juste une abréviation pour 2 transitions, chacune étiquetée par une seule lettre : $3 \xrightarrow{a} 1$ et $3 \xrightarrow{b} 1$.

Pour résumer, un automate est un objet mathématique fini, proche d'un graphe orienté, et que l'on peut implémenter par une structure de donnée. La définition suivante formalise ce que nous avons vu.

Définition 3.2.1 — Automate. Un automate (fini) $\mathcal{A} = (A, Q, I, F, \delta)$ est donné par

- un alphabet A ,
- un ensemble d'états Q ,
- un ensemble d'états initiaux $I \subseteq Q$,
- un ensemble d'états finaux (aussi appelés acceptants) $F \subseteq Q$,
- un ensemble de transitions $\delta \subseteq Q \times A \times Q$.

Pour illustrer cette définition, l'automate de la FIGURE 3.2 est donné par

- $A = \{a, b\}$,
- $Q = \{1, 2, 3\}$,
- $I = \{1\}$,
- $F = \{1, 2\}$,
- $\delta = \{(1, a, 2), (2, b, 2), (2, b, 3), (3, a, 1), (3, b, 1)\}$.

Une transition peut se noter de plusieurs façons, au choix :

- soit par une flèche étiquetée, comme $1 \xrightarrow{a} 2$,
- soit par un triplet, comme $(1, a, 2)$.
- soit par un couple, comme $((1, a), 2)$.

Notation 3.1. Lorsqu'on a une suite de transitions d'un état p_0 jusqu'à un état p_n de la forme

$$p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} p_n,$$

on peut l'écrire de façon condensée

$$p_0 \xrightarrow{a_1 \dots a_n} p_n,$$

ou, si on ne veut pas préciser le mot $a_1 \dots a_n$, on écrit encore plus simplement

$$p_0 \xrightarrow{*} p_n.$$

Cela signifie donc qu'il existe un chemin dans le graphe de l'automate allant de p_0 à p_n . On dit que p_n est accessible depuis p_0 . Cette notation s'utilise en particulier lorsque $p_0 = p_n$, car un mot possible pour aller de p_0 à lui-même est le mot vide.

3.2.2 Mots et langage acceptés

Un automate accepte certains mots et rejette les autres. Pour comprendre comment, on a besoin de définir la notion de "calcul", ou "run".

Définition 3.2.2 — Calcul, ou Run. Un calcul (ou run) d'un automate sur un mot $a_1 a_2 \dots a_n$, où les a_i sont des lettres, est un chemin formé de transitions consécutives qui part d'un état initial de l'automate, donc de la forme

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \dots \xrightarrow{a_n} q_n$$

■ **Exemple 3.1** Pour l'automate de la FIGURE 3.1, un calcul doit donc partir de l'état 1 (qui est le seul état initial). Voici quelques exemples de calculs :

- $1 \xrightarrow{a} 2$ est un calcul sur le mot a .
- $1 \xrightarrow{a} 2 \xrightarrow{b} 2 \xrightarrow{b} 2$ est un calcul sur le mot abb .
- $1 \xrightarrow{a} 2 \xrightarrow{b} 2 \xrightarrow{b} 3$ est un autre calcul sur le mot abb .
- $1 \xrightarrow{a} 2 \xrightarrow{b} 3 \xrightarrow{b} 1$ est encore un autre calcul sur le mot abb .
- 1 est un calcul sur le mot vide. ■

Définition 3.2.3 — Calcul acceptant, ou Run acceptant. Un calcul est appelé **acceptant** s'il se termine dans un état final.

Par exemple, sur le mot abb et avec l'automate de la FIGURE 3.1,

- Le calcul $1 \xrightarrow{a} 2 \xrightarrow{b} 2 \xrightarrow{b} 2$ est acceptant (car il se termine dans l'état final 2).
- Le calcul $1 \xrightarrow{a} 2 \xrightarrow{b} 2 \xrightarrow{b} 3$ **n'est pas** acceptant (car il se termine dans l'état **non** final 3).
- Le calcul $1 \xrightarrow{a} 2 \xrightarrow{b} 3 \xrightarrow{b} 1$ est acceptant (car il se termine dans l'état final 1).

R **Remarque** Il ne faut pas confondre état *initial* et état *non final* : un état est initial si on peut y faire *débuter* un calcul. Un état est non final si un calcul qui s'y *termine* n'est pas acceptant.

Cette notion de calcul (ou run) acceptant, nous permet de définir si un mot est accepté par l'automate.

Définition 3.2.4 — Mot accepté (ou reconnu). Un mot u est **accepté** par un automate \mathcal{A} s'il existe au moins un calcul acceptant de \mathcal{A} sur le mot u . On dit aussi qu'un mot est **reconnu** par l'automate.

Par exemple, le mot abb est accepté, car il y a un calcul sur ce mot qui est acceptant. En fait, on a même deux calculs acceptants. On remarque aussi qu'il y a un calcul non acceptant (celui qui se termine dans l'état 3), mais cela ne change rien au fait que le mot abb est accepté, puisqu'il y a au moins un autre calcul acceptant.

Par contre, les mots aa et b ne sont pas acceptés, car il n'y a aucun calcul sur ces mots (et donc aucun calcul acceptant). Un mot n'est pas accepté (on dit aussi pas reconnu, ou rejeté) lorsqu'aucun calcul sur ce mot n'est acceptant. C'est le cas en particulier lorsqu'il n'y a aucun calcul sur le mot, parce que des transitions manquent pour lire le mot depuis un état initial.

Définition 3.2.5 — Langage accepté (ou reconnu). Le langage accepté par un automate \mathcal{A} , noté $L(\mathcal{A})$, est l'ensemble des mots acceptés par l'automate \mathcal{A} .

■ **Exemple 3.2** On peut vérifier que le langage accepté par l'automate de la FIGURE 3.1 est

$$\varepsilon + a[b + b(a + b)a]^*(\varepsilon + ba + bb).$$

Il n'est pas évident *a priori* que cette expression est bien correcte. On verra plus tard un *algorithme* permettant d'exprimer à l'aide d'une expression rationnelle le langage accepté par un automate donné en entrée de l'algorithme. ■

La remarque qui termine l'exemple précédent dit que tout langage accepté par un automate peut se décrire par une expression rationnelle. Qu'a-t-on donc gagné avec les automates, par rapport à ce qu'on avait déjà avec les expressions rationnelles ?

Un premier point est qu'il est parfois plus facile de raisonner en pensant avec les expressions, et parfois plus facile de raisonner en pensant avec les automates.

Exercice 3.1 — Construction d'automates. Construire des automates permettant de reconnaître les langages suivants sur l'alphabet $\{a, b\}$:

1. Le langage des mots qui **ne** contiennent **pas** le facteur bab .
2. Le langage des mots qui **ne** contiennent **pas** le facteur bab , mais qui contiennent le facteur aba .

L'exercice 3.1 demande une construction d'automates, plus simple que si on demandait la construction directe d'expressions rationnelles (essayez !). Cela est dû en particulier au fait que, si un automate \mathcal{A} est mis sous une bonne forme, on peut facilement calculer un automate pour le complémentaire du langage $L(\mathcal{A})$, c'est-à-dire calculer un automate \mathcal{B} tel que $L(\mathcal{B}) = A^* \setminus L(\mathcal{A})$. Au contraire, il est beaucoup plus difficile de calculer directement, à partir d'une expression e , le complémentaire de l'ensemble des mots décrits par e .

Cependant, pour pouvoir calculer cet automate \mathcal{B} , il faut que l'automate \mathcal{A} ait deux propriétés : il doit être *complet* et *déterministe*. Avant d'introduire ces deux notions, voyons comment construire les automates de l'exercice 3.1.

Solution — (de l'exercice 3.1). 1. On va d'abord construire un automate qui reconnaît le langage des mots qui contiennent le facteur bab . Une expression rationnelle pour ce langage est $(a+b)^*bab(a+b)^*$. On pourrait faire l'automate suivant :

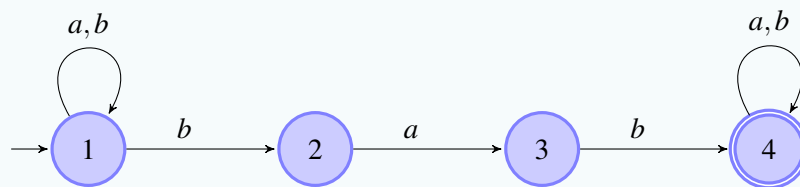


FIGURE 3.3 – Un premier automate qui reconnaît le langage $(a+b)^*bab(a+b)^*$

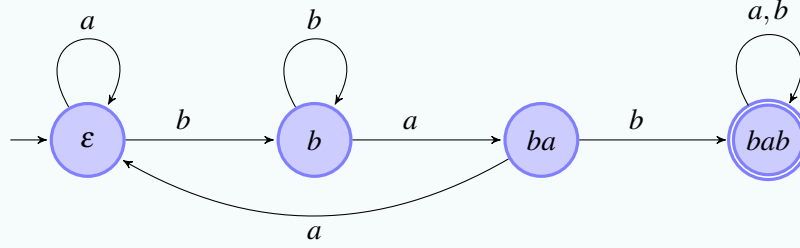
Il est facile de voir pourquoi cet automate accepte bien les mêmes mots que ceux décrits par $(a+b)^*bab(a+b)^*$: si un mot u contient le facteur bab , alors u peut s'écrire $u = xbaby$, pour $x, y \in A^*$. Dans ce cas, il y a bien un calcul acceptant sur u : on reste dans l'état 1 en lisant x , puis on prend les transitions $1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4$, et on reste enfin dans l'état 4 en lisant y . Inversement, si un mot est accepté, il doit étiqueter un chemin allant de l'état initial 1 à l'état final 4, donc passer par les transitions $1 \xrightarrow{b} 2 \xrightarrow{a} 3 \xrightarrow{b} 4$, donc contenir le facteur bab .

Il n'est pas clair qu'on peut utiliser cet automate pour calculer un automate reconnaissant le *complémentaire* de son langage accepté (qui est le langage qui nous intéresse).

On remarque que dans cet automate

- il n'y a pas de transition étiquetée par la lettre b sortant de l'état 2.
- il y a deux transitions étiquetées a sortant de l'état 1.

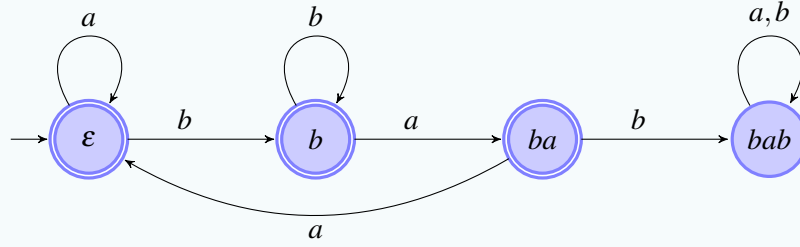
On va construire un automate pour lequel, depuis n'importe quel état, il sort exactement une transition par a et une transition par b . L'idée est de mémoriser dans chaque état du nouvel automate le plus long suffixe du mot lu qui est aussi un préfixe du mot bab . Cette information est indiquée dans les noms d'états sur l'automate de la FIGURE 3.4. Une fois qu'on se trouve dans l'état bab , on reste dans cet état, car on a détecté le facteur bab dans le mot d'entrée.

FIGURE 3.4 – Un second automate qui reconnaît le langage $(a + b)^*bab(a + b)^*$

Cet automate a la bonne propriété que depuis chaque état p et pour chaque lettre x , il y a exactement une transition étiquetée x qui sort de l'état p . Par conséquent, chaque mot a **exactement un calcul** dans l'automate (puisque à chaque lettre lue, il y a exactement un choix possible pour l'état suivant). En résumé :

- Chaque mot a un unique calcul dans l'automate.
- Si ce calcul se termine dans l'unique état final (nommé bab), le mot est accepté.
- Si ce calcul se termine dans l'un des autres états, le mot est rejeté.

Construisons l'automate \mathcal{B} de la façon suivante : on reprend le même automate que \mathcal{A} , sauf pour les états finaux. Si $\mathcal{A} = (A, Q, I, F, \delta)$, alors $\mathcal{B} = (A, Q, I, Q \setminus F, \delta)$. Autrement dit, un état est final dans \mathcal{B} si et seulement s'il n'est **pas** final dans \mathcal{A} . L'automate \mathcal{B} est dessiné sur la FIGURE 3.5.

FIGURE 3.5 – Un automate qui reconnaît le langage $A^* \setminus (a + b)^*bab(a + b)^*$

Par construction, un mot est reconnu par \mathcal{B} si et seulement si l'unique calcul sur ce mot dans \mathcal{B} est acceptant, si et seulement si l'unique calcul sur ce mot dans \mathcal{A} n'est pas acceptant, si et seulement si le mot n'est pas reconnu par \mathcal{A} . Donc $L(\mathcal{B}) = A^* \setminus L(\mathcal{A})$ est le langage des mots qui **ne** contiennent **pas** le facteur bab .

En analysant l'automate \mathcal{B} , on peut en déduire une expression rationnelle pour ce langage, par exemple

$$(a + b^+aa)^*(\varepsilon + b^+ + b^+a). \quad (3.1)$$

Trouver une telle expression sans utiliser les automates est plus difficile. Il faut noter que $A^* \setminus (a + b)^*bab(a + b)^*$ n'est pas une expression rationnelle valide, puisqu'on utilise explicitement l'opérateur de complément $A^* \setminus$.

2. On peut utiliser les automates de la question précédente. On a un automate qui reconnaît les mots n'ayant pas le facteur bab . C'est l'automate de la FIGURE 3.5, dont on peut renommer les états, et supprimer l'état bab qui ne sert à rien (car une fois qu'on

est dans cet état, on ne peut plus atteindre un état final). On obtient l'automate \mathcal{A}_1 de la FIGURE 3.6.

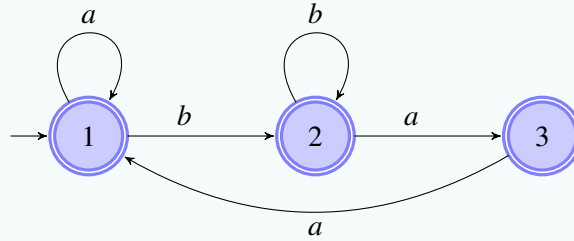


FIGURE 3.6 – Un automate \mathcal{A}_1 qui reconnaît le langage $A^* \setminus (a+b)^* bab(a+b)^*$

On a aussi un automate qui reconnaît les mots ayant le facteur aba : c'est celui de la figure 3.4, en échangeant les rôles de a et b , et en renommant les états. On obtient l'automate \mathcal{A}_2 de la FIGURE 3.7.

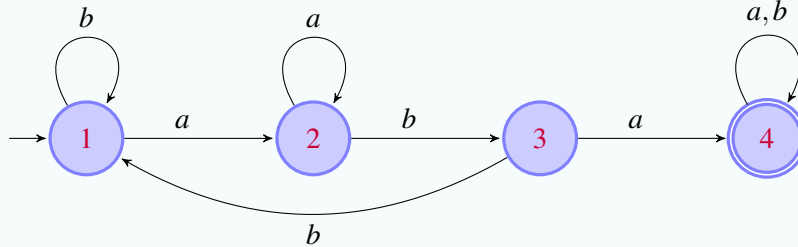


FIGURE 3.7 – Un automate \mathcal{A}_2 qui reconnaît le langage $(a+b)^* aba(a+b)^*$

On a distingué les états du premier automate (en noir) de ceux du second (en rouge). On veut construire un automate capable de vérifier qu'un mot est accepté à la fois dans l'automate de la FIGURE 3.6 et dans celui de la FIGURE 3.7. L'idée est de mémoriser dans l'état courant du nouvel automate à la fois l'état courant du premier automate, et l'état courant du second automate. Par exemple, le mot $aaba$ a comme run

- $1 \xrightarrow{a} 1 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{a} 3$ dans l'automate \mathcal{A}_1 .
- $1 \xrightarrow{a} 2 \xrightarrow{a} 2 \xrightarrow{b} 3 \xrightarrow{a} 4$ dans l'automate \mathcal{A}_2 .

On va construire un automate qui effectue le calcul dans les 2 automates à la fois. On veut par exemple que pour le mot $aaba$, le calcul soit

- $\begin{pmatrix} 1 \\ 1 \end{pmatrix} \xrightarrow{a} \begin{pmatrix} 1 \\ 2 \end{pmatrix} \xrightarrow{a} \begin{pmatrix} 1 \\ 2 \end{pmatrix} \xrightarrow{b} \begin{pmatrix} 2 \\ 3 \end{pmatrix} \xrightarrow{a} \begin{pmatrix} 3 \\ 4 \end{pmatrix}$ dans le nouvel automate.

Un état de ce nouvel automate qui doit simuler à la fois \mathcal{A}_1 et \mathcal{A}_2 sera composé d'un état de \mathcal{A}_1 et d'un état de \mathcal{A}_2 . Pour rendre la figure plus lisible, au lieu d'écrire $\begin{pmatrix} 1 \\ 2 \end{pmatrix}$

comme nom d'état, par exemple, on écrit $1/2$. Le calcul commence dans l'état $1/1$. À partir de cet état, si on lit un a , on doit aller dans l'état 1 dans l'automate \mathcal{A}_1 et dans l'état 2 dans l'automate \mathcal{A}_2 , donc dans l'état $1/2$ dans l'automate que nous construisons. Après avoir construit toutes les transitions possibles, on obtient l'automate de la FIGURE 3.8.

Pour qu'un mot soit accepté, il faut que ses calculs dans chacun des automates \mathcal{A}_1 et \mathcal{A}_2 soient tous les deux acceptants. Les états finaux sont donc ceux pour lesquels les

deux composantes sont des états finaux : 1/4, 2/4, et 3/4.

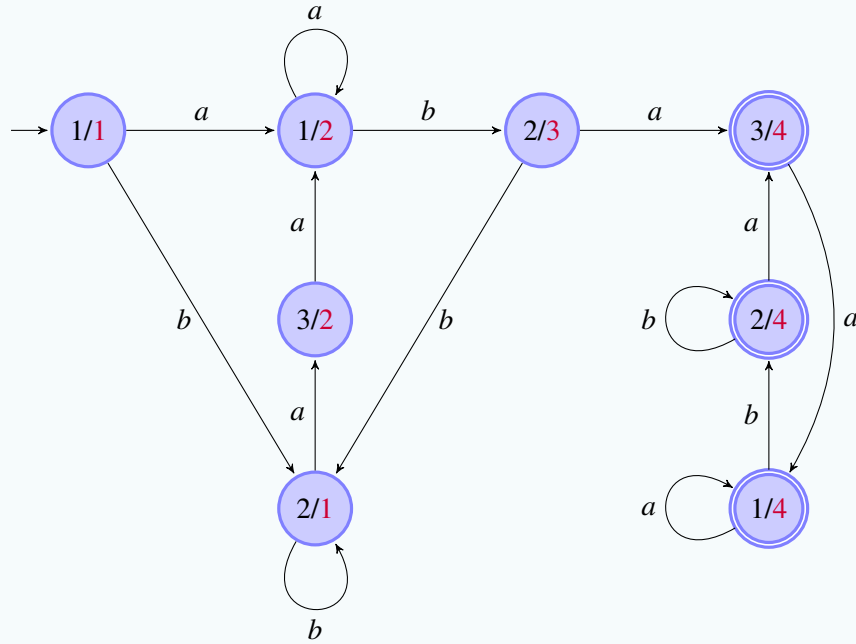


FIGURE 3.8 – Un automate reconnaissant $[A^* \setminus (a+b)^*bab(a+b)^*] \cap (a+b)^*aba(a+b)^*$

Obtention d'une expression rationnelle

En analysant cet automate, on peut obtenir une expression rationnelle. Les mots contenant *aba* sont décrits par l'expression $(a+b)^*aba(a+b)^*$, et les mots ne contenant pas *bab* par l'expression $A^* \setminus (a+b)^*bab(a+b)^*$. Cette dernière expression n'est pas une expression rationnelle car elle utilise le complément $A^* \setminus$. De même, le langage des mots qui contiennent *aba* mais pas *bab* correspond à l'expression $[A^* \setminus (a+b)^*bab(a+b)^*] \cap [(a+b)^*aba(a+b)^*]$, mais ce n'est pas une expression rationnelle, d'une part pour la même raison, et aussi parce qu'elle utilise l'intersection.

Pour obtenir une expression rationnelle, qui n'utilise que les opérateurs $+$, \cdot , $*$, on analyse les chemins acceptants dans l'automate de la FIGURE 3.8. Un chemin acceptant doit aller de l'état 1/1 à l'un des états acceptants. Pour cela, il doit passer par l'état 1/2. La première fois qu'on passe par l'état 1/2, c'est soit après avoir lu *a*, soit après avoir lu une suite non vide de *b* suivie de deux *a*. Cela nous donne le début d'expression

$$(a + b^+aa). \quad (3.2)$$

Ensuite, on peut éventuellement boucler autour de l'état 1/2, en utilisant

- soit la boucle *a*,
- soit le chemin $1/2 \xrightarrow{b} 2/3 \xrightarrow{b^+} 2/1 \xrightarrow{a} 3/2 \xrightarrow{a} 1/2$.

Ces chemins qui bouclent autour de l'état 1/2 peuvent être décrits par l'expression

$$(a + bb^+aa)^*. \quad (3.3)$$

Ensuite, on ne revient plus sur l'état 1/2, et on atteint l'état 3/4 par le mot

$$ba \quad (3.4)$$

En combinant les expressions (3.2), (3.3), (3.4), on obtient finalement une expression pour le langage des mots contenant aba comme facteur, mais pas bab :

$$(a + b^+aa)(a + bb^+aa)^*ba \cdot (a^+b^+a)^*(\epsilon + a^+ + a^+b^+).$$

Exercice 3.2 Sur l'alphabet $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, trouver un automate fini reconnaissant exactement, les écritures en base 10 des entiers divisibles par 3. ■

Solution L'idée est de mémoriser dans les états de l'automate le reste de la division par 3 de l'entier lu en entrée. On note $(k \bmod 3)$ le reste de la division de k par 3. Ce reste vaut 0, 1 ou 2. On aura donc 3 états.

Il faut bien distinguer un mot sur l'alphabet $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, qui est une suite de lettres de cet alphabet, de la valeur que ce mot représente. Pour un mot u , on note $\text{val}(u)$ la valeur du nombre représenté par u . Par exemple, si $u = 123$, la valeur $\text{val}(u)$ est l'entier 123. Si $u = 0000123$ (qui est un mot différent du précédent : il est plus long), la valeur $\text{val}(u)$ est encore l'entier 123.

L'objectif est de mettre en place les transitions entre les états 0, 1 et 2 de telle façon que lorsqu'on lit un mot u , on arrive dans l'état $(\text{val}(u) \bmod 3)$. Supposons que le mot u mène effectivement dans cet état. Lorsqu'on lit la lettre suivante, qu'on appelle $a \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, la valeur du mot ua est

$$\text{val}(ua) = 10 \times \text{val}(u) + \text{val}(a).$$

Lorsqu'on regarde le reste modulo 3, comme $(10 = 1 \bmod 3)$, cela donne :

$$\begin{aligned} (\text{val}(ua) \bmod 3) &= ((\text{val}(u) + \text{val}(a) \bmod 3)) \\ &= ((\text{val}(u) \bmod 3) + (\text{val}(a) \bmod 3)). \end{aligned}$$

En résumé, on se trouve après avoir lu le mot u dans l'état $q = (\text{val}(u) \bmod 3)$, et la lecture de la lettre a fait passer

- de l'état q
- à l'état $q + (\text{val}(a) \bmod 3)$.

Pour obtenir l'état suivant, il suffit donc d'ajouter, modulo 3, la valeur de la lettre lue à la valeur de l'état courant. Cela permet d'obtenir l'automate voulu, représenté en FIGURE 3.9. Le seul état final est 0, puisqu'un entier est divisible par 0 si et seulement si le reste de sa division par 3 vaut 0.

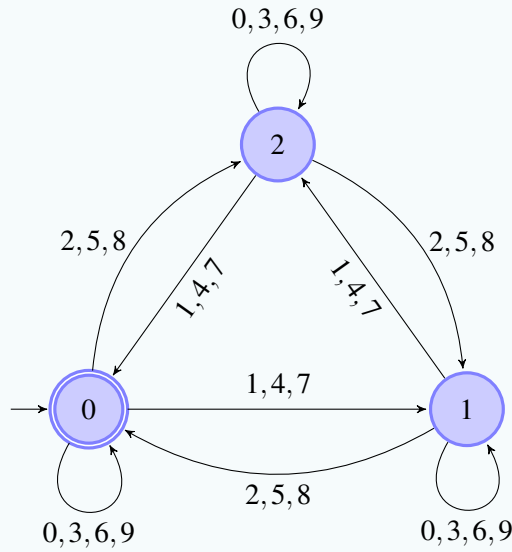


FIGURE 3.9 – Un automate qui reconnaît les représentations en base 10 des multiples de 3

Cette méthode fonctionnerait aussi pour d'autres entiers que 3 et pour d'autres bases que 10. On pourrait par exemple construire un automate reconnaissant les représentations en base 2 des entiers divisibles par 7. On verra au Chapitre 5 un algorithme pour extraire d'un automate une expression rationnelle décrivant le langage reconnu par l'automate. Essayez de le faire sur cet automate. ■

3.3 Automates Complets et Automates Déterministes

3.3.1 Automates Complets

Cette section présente deux propriétés importantes des automates. Si un automate \mathcal{A} possède ces deux propriétés, on peut calculer un automate \mathcal{B} qui reconnaît le langage complémentaire de celui reconnu par \mathcal{A} , comme on l'a fait dans l'exercice 3.1.

Définition 3.3.1 — Automate complet. Un automate $\mathcal{A} = (A, Q, I, F, \delta)$ est **complet** si pour chaque état $q \in Q$ et pour chaque lettre $a \in A$, il existe au moins une transition étiquetée par a sortant de l'état q .

Par exemple, l'automate de la FIGURE 3.1 n'est pas complet, car :

- de l'état 1, il n'y a pas de transition étiquetée b ,
- de l'état 2, il n'y a pas de transition étiquetée a .

Si un automate \mathcal{A} n'est pas complet, on peut facilement construire un automate complet \mathcal{B} qui reconnaît le même langage que \mathcal{A} .

Proposition 3.3.1 Soit $\mathcal{A} = (A, Q, I, F, \delta)$ un automate et soit $\mathcal{B} = (A, Q \cup \{q_\perp\}, I, F, \delta')$, où

- q_\perp est un nouvel état,
- les transitions de \mathcal{B} , données par δ' , sont les suivantes :

$$\delta' = \delta \cup \{(p, a, q_\perp) \mid p \in Q, a \in A, \text{ et il n'y a aucune transition de la forme } (p, a, q) \text{ dans } \delta\}.$$

Alors l'automate \mathcal{B} est complet et reconnaît le même langage que \mathcal{A} .

L'automate \mathcal{B} de la Proposition 3.3.1 s'appelle le **complété** de \mathcal{A} .

- Exercice 3.3** 1. Appliquer la construction de la Proposition 3.3.1 à l'automate de la FIGURE 3.1.
2. Justifier la correction de la construction de la Proposition 3.3.1. ■

3.3.2 Automates déterministes

On peut implémenter un automate, vue comme une machine lisant des lettres et changeant d'état : chaque état peut correspondre à une variable globale. Par exemple, l'automate de la FIGURE 3.7 qui reconnaît les mots de $(a+b)^*aba(a+b)^*$ correspondrait au code C suivant :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int c, state = 1;
7
8      printf("\n%d ", state);
9
10     while((c = getchar()) != EOF)
11     {
12         switch (state)
13         {
14             case 1 :
15                 state = (c == 'a') ? 2 : 1;
16                 break;
17             case 2 :
18                 state = (c == 'a') ? 2 : 3;
19                 break;
20             case 3 :
21                 state = (c == 'a') ? 4 : 1;
22                 break;
23             default :
24                 break;
25         }
26         if (c == 'a' || c == 'b')
27             printf("-%c-> %d ", c, state);
28     }
29
30     printf("\n\n");
31
32     if (state == 4)
33         printf("Mot d'entrée accepté.\n");
34     else
35         printf("Mot d'entrée rejeté.\n");
36
37     return EXIT_SUCCESS;
38 }
39
```

On peut tester ce code une fois compilé par

```
> echo -n abbbabbabaaa | ./a.out
```

```
1 -a-> 2 -b-> 3 -b-> 1 -b-> 1 -a-> 2 -b-> 3 -b-> 1 -a-> 2 -a-> 2 -a-> 2
```

```
Mot d'entrée rejeté.
```

et

```
> echo -n abbbababaaa | ./a.out
```

```
1 -a-> 2 -b-> 3 -b-> 1 -b-> 1 -a-> 2 -b-> 3 -a-> 4 -a-> 4 -a-> 4
```

```
Mot d'entrée accepté.
```

Le code affiche le chemin correspondant au mot d'entrée (le premier est rejeté, le second est accepté). Ce qui rend le programme simple est que pour tout état de l'automate et pour chacune des lettres a et b , il y a exactement une transition sortante. Si l'automate n'était pas complet, cela ne serait pas beaucoup plus compliqué : on ajouterait un état puits 0, ce qui correspond à compléter l'automate.

Par contre, un automate comme celui de la FIGURE 3.3 serait plus compliqué à traduire en C, parce que de l'état 1 sortent **deux** transitions étiquetées par b . Cela motive la définition suivante.

Définition 3.3.2 — Automate déterministe. On dit qu'un automate $\mathcal{A} = (A, Q, I, F, \delta)$ est **déterministe** si les deux propriétés suivantes sont vérifiées :

1. $|I| = 1$, c'est-à-dire qu'il y a un seul état initial.
2. Pour tout état $p \in Q$ et toute lettre $a \in A$, il y a au plus une transition étiquetée a qui sort de l'état p .

Par exemple, l'automate de la FIGURE 3.3 n'est pas déterministe parce qu'il y a **deux** transitions étiquetées par b sortant de l'état 1. De même, l'automate de la FIGURE 3.1 n'est pas déterministe parce qu'il y a **deux** transitions étiquetées par b sortant de l'état 2.

En revanche, l'automate de la FIGURE 3.4 est déterministe et d'ailleurs aussi complet. L'automate de la FIGURE 3.6 est déterministe (mais pas complet, parce qu'il n'y a pas de transition b sortant de l'état 3).

On a vu dans la Section 3.3.1 qu'il est facile de compléter un automate : il suffit d'ajouter un état, dans le cas le pire. On peut se poser la même question pour les automates non-déterministes : lorsqu'on part d'un automate \mathcal{A} non-déterministe, peut-on toujours construire un automate *déterministe* \mathcal{B} qui reconnaît le même langage que \mathcal{A} ? Cette question est moins facile que la précédente, mais on verra dans le Chapitre 4 qu'on peut effectivement déterminer tout automate. Cependant, cette phase de détermination peut être coûteuse : si l'automate \mathcal{A} a n états, l'automate \mathcal{B} peut avoir $2^n - 1$ états, dans le cas le pire. Pour donner une idée, si $n = 30$, alors $2^n - 1 = 1073741823$, soit plus d'un milliard.

L'intérêt des automates à la fois déterministes et complets est qu'ils peuvent être facilement complétés, au sens où si on a un tel automate \mathcal{A} , on peut facilement calculer un automate \mathcal{B} qui reconnaît le *complémentaire* de $L(\mathcal{A})$. Rappelons que le *complémentaire* (ou *complément*) d'un ensemble L de mots de A^* est l'ensemble $A^* \setminus L$ des mots qui **ne** sont **pas** dans L . Nous avons en fait déjà fait ce type de construction dans la solution de l'exercice 3.1.

Proposition 3.3.2 — Complémentation d'un automate complet et déterministe. Soit $\mathcal{A} = (A, Q, I, F, \delta)$ un automate complet et déterministe. Soit $\mathcal{B} = (A, Q, I, Q \setminus F, \delta)$. C'est-à-dire, \mathcal{B} est obtenu à partir de \mathcal{A} en rendant finaux les états non finaux dans \mathcal{A} , et vice-versa. Alors, \mathcal{B} reconnaît le langage complémentaire de celui reconnu par \mathcal{A} :

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

R Remarque

1. Il ne faut bien sûr pas confondre *compléter* et *complémenter*.
2. Si l'automate \mathcal{A} n'est pas complet, la Proposition 3.3.2 n'est plus vraie. En effet, si l'automate n'est pas complet, il est possible qu'un mot n'ait aucun calcul dans l'automate \mathcal{A} . Mais par définition de \mathcal{B} , ce mot n'aura aucun calcul dans \mathcal{B} , et donc sera rejeté à la fois par \mathcal{A} et par \mathcal{B} . Un exemple est donné par l'automate suivant :

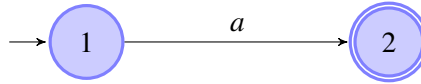


FIGURE 3.10 – Automate incomplet

Cet automate reconnaît le langage $\{a\}$. Si on applique la construction de la Proposition 3.3.2, l'état 1 devient final et l'état 2 devient non final, on obtient un automate qui reconnaît le langage $\{\varepsilon\}$, qui n'est pas le complémentaire de $\{a\}$.

3. Si l'automate \mathcal{A} n'est pas déterministe, la Proposition 3.3.2 n'est plus vraie. En effet, si l'automate n'est pas déterministe, il est possible qu'un mot ait deux calculs dans l'automate \mathcal{A} : le premier acceptant, le second rejetant. Par définition de \mathcal{B} , ce mot aura aussi deux calculs dans \mathcal{B} , le premier rejetant, le second acceptant. Il sera donc accepté à la fois par \mathcal{A} et par \mathcal{B} , puisqu'il a un calcul acceptant dans chacun des automates. Un exemple est donné par l'automate suivant :

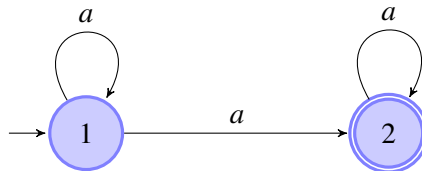


FIGURE 3.11 – Automate non-déterministe

Cet automate reconnaît le langage a^+ . Si on applique la construction de la Proposition 3.3.2, l'état 1 devient final et l'état 2 devient non final, on obtient un automate qui reconnaît le langage a^* , qui n'est pas le complémentaire de a^+ .

4. Détermination des automates

4.1 Motivation et Présentation Informelle

Nous avons déjà vu une motivation d'obtenir des automates déterministes : une fois un automate déterministe complété (ce qui est très simple à faire : au pire on doit ajouter un état et quelques transitions) on peut facilement calculer un automate qui reconnaît le complémentaire du langage reconnu par l'automate de départ.

De plus, implémenter un automate non-déterministe n'est pas si facile qu'implémenter un déterministe, comme cela a été fait pour l'automate de la FIGURE 3.7. Une façon de le faire serait de se souvenir à chaque instant de tous les états dans lesquels l'automate est susceptible de se trouver. Cette remarque permet de concevoir un algorithme de détermination. Cette section présente cet algorithme de façon informelle.

Prenons à nouveau un exemple : l'automate de la FIGURE 4.1 reconnaît le langage des mots sur l'alphabet $\{a, b\}$ qui ont un a en avant-dernière position. Il n'est pas déterministe (pourquoi ?).

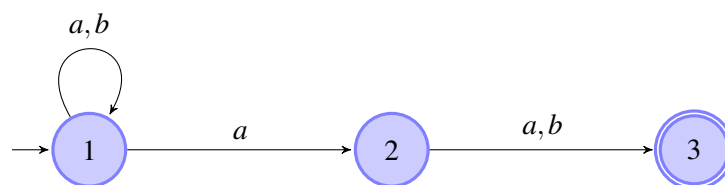


FIGURE 4.1 – Automate reconnaissant les mot de $\{a, b\}^*$ avec un a en avant-dernière position

Représentons l'ensemble des calculs sur le mot *babaa* :

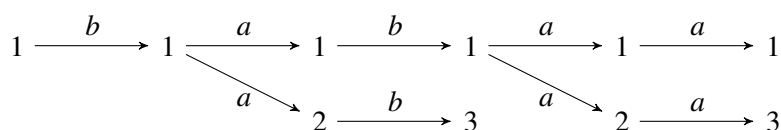


FIGURE 4.2 – Ensemble des calculs de l'automate de la FIGURE 4.1 sur le mot *babaa*

L'idée principale pour construire un automate déterministe qui reconnaît le même langage

est que cet automate retienne, dans ses états, l'ensemble des états de l'automate original dans lesquels l'automate original peut se trouver. Cela revient à regrouper en un ensemble les états apparaissant sur la même ligne verticale dans la FIGURE 4.2.

On voudrait ainsi que le calcul de cet automate sur le mot *babaa* soit le suivant :

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

FIGURE 4.3 – Le calcul correspondant de l'automate déterministe associé

L'information extraite de ce calcul est qu'après avoir lu *babaa*, on se trouve soit dans l'état 1 soit dans l'état 3. Comme l'état 3 est acceptant dans l'automate original, *babaa* devra être accepté.

Au lieu de faire cette construction seulement pour un mot, on peut la faire entièrement sur l'automate. Cette construction nous donne l'automate suivant :

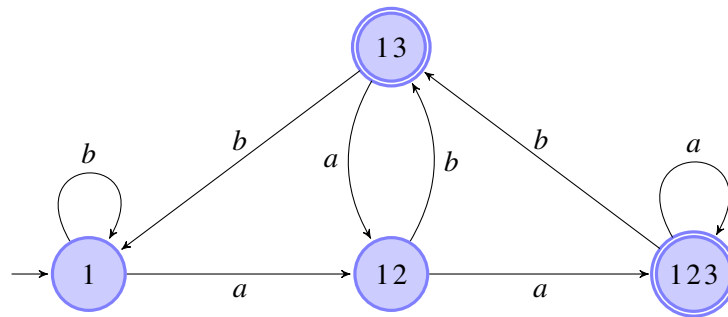


FIGURE 4.4 – Automate déterminisé de l'automate de la FIGURE 4.1

Cet automate retient dans ses états le sous-ensemble de l'ensemble $\{1,2,3\}$ des états originaux dans lequel pourrait être l'automate de la FIGURE 4.1. Pour alléger la notation, on a enlevé les accolades et les virgules, c'est-à-dire qu'on a écrit dans un état 123 plutôt que $\{1,2,3\}$, ou encore 13 plutôt que $\{1,3\}$.

Comment cet automate a-t-il été construit ?

- Comme l'automate original n'a qu'un seul état initial, 1, on part de l'état initial $\{1\}$, noté 1 sur la figure. Détaillons les transitions qui sortent de cet état :
 - Si on lit un *b* à partir de l'état 1 dans l'automate original, on reste en *b*, ce qui explique la boucle autour de l'état 1 dans le nouvel automate.
 - Par contre, si depuis l'état 1 on lit un *a*, alors dans l'automate original, on a le choix d'aller soit en 1 soit en 2. Cela explique la transition $\{1\} \xrightarrow{a} \{1,2\}$ dans le nouvel automate.
- Détaillons maintenant les transitions partant de l'état $\{1,2\}$. Être dans l'état $\{1,2\}$ du nouvel automate correspond à être soit dans l'état 1, soit dans l'état 2 dans l'automate original. À partir de l'un de ces états :
 - Si on lit un *a* à partir de $\{1,2\}$, c'est soit qu'on se trouvait en 1, et on peut se retrouver en 1 ou en 2, soit qu'on se trouvait en 2, et on se retrouve en 2 après la lecture de *a*. Au final, partant de 1 ou de 2 dans l'automate original et lisant un *a*, on peut se retrouver dans 1, 2 ou 3. Ceci explique la transition $\{1,2\} \xrightarrow{a} \{1,2,3\}$.
 - De même, si on lit un *b* à partir de $\{1,2\}$, c'est soit qu'on se trouvait en 1, et on reste en 1 après lecture du *b*, soit qu'on se trouvait en 2, et on progresse en 3. Au final, partant de 1 ou de 2 dans l'automate original et lisant un *b*, on peut se retrouver dans 1 ou 3. Ceci explique la transition $\{1,2\} \xrightarrow{b} \{1,3\}$.

Exercice 4.1 — Détermination. Construire un automate **déterministe** reconnaissant le même langage que

1. l'automate de la FIGURE 3.2,
2. l'automate de la FIGURE 3.3.

Si vous avez fait la question 1 de cet exercice en suivant la démarche expliquée plus haut, vous devez avoir obtenu un automate avec plus d'états que l'automate original. Pour la question 2, obtenez-vous le même automate que celui de la FIGURE 3.4 (au nom des états près) ?

Le nombre d'états obtenus pour l'automate déterministe peut être plus grand que celui de l'automate de départ. Par exemple, l'automate de la FIGURE 4.4 a 4 états, alors que l'automate non-déterministe dont il provient, celui de la FIGURE 4.1, n'en a que 3. L'exercice suivant illustre le fait que l'automate déterminisé peut avoir plus d'un état supplémentaire.

Exercice 4.2 1. Donner une expression rationnelle pour le langage des mots sur l'alphabet $\{a, b\}$ qui ont un a en avant-avant-dernière position.
 2. Construire un automate (éventuellement non-déterministe) qui reconnaît ce langage.
 3. Si l'automate obtenu à la question 2 n'est pas déterministe, construire à partir de cet automate un automate déterministe qui reconnaît le même langage.

4.2 Un algorithme de détermination

Dans cette section, nous allons formaliser la méthode utilisée dans l'exercice en un algorithme. L'objectif est, à partir d'un automate non-déterministe \mathcal{A} , de construire un automate **déterministe** \mathcal{B} tel que

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

L'idée est la même que dans l'exemple de la Section 4.1 : l'automate \mathcal{B} va « retenir » l'ensemble des états dans lesquels l'automate \mathcal{A} peut se trouver après avoir lu un mot. Si $\mathcal{A} = (A, Q, I, F, \delta)$, l'ensemble des états de \mathcal{B} sera donc l'ensemble des sous-ensembles de Q . Cet ensemble est noté 2^Q (si Q a n éléments, alors 2^Q en a 2^n , ce qui rend la notation cohérente). Par exemple, si $Q = \{1, 2, 3\}$, on a $2^Q = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.

Construction de l'automate déterminisé

On formalise la construction faite précédemment sur un exemple.

Définition 4.2.1 — Automate déterminisé. Soit $\mathcal{A} = (A, Q, I, F, \delta)$. L'automate déterminisé de \mathcal{A} est l'automate \mathcal{B} suivant :

- L'ensemble d'états de \mathcal{B} est 2^Q .
- L'automate \mathcal{B} a un unique état initial : I .
- Un état $P \subseteq Q$ de \mathcal{B} est final ssi $P \cap F \neq \emptyset$.
- Les transitions de \mathcal{B} sont définies ainsi : pour P et R deux sous-ensembles de Q et pour a une lettre, on a

$$P \xrightarrow{a} R$$

lorsque

$$R = \{q \in Q \mid \exists p \in P, (p, a, q) \in \delta\}.$$

L'automate \mathcal{B} s'appelle le *déterminisé* de l'automate \mathcal{A} .

Proposition 4.2.1 — Détermination. L'automate \mathcal{B} construit à la Définition 4.2.1 est un automate déterministe et complet qui reconnaît le même langage que l'automate \mathcal{A} .

Démonstration. L'automate \mathcal{B} est bien déterministe et complet par construction : pour toute lettre a et tout état P de \mathcal{B} , il y a un unique état R atteint depuis P en lisant a : c'est l'état $R = \{q \in Q \mid \exists p \in P, (p, a, q) \in \delta\}$, c'est-à-dire l'ensemble des états qu'on peut atteindre dans \mathcal{A} depuis un état de P par la lettre a . Il ne reste donc plus qu'à montrer que $L(\mathcal{A}) = L(\mathcal{B})$.

Admettons provisoirement l'observation suivante : pour tout mot w et tout état $r \in Q$, il existe un chemin $q \xrightarrow{w} r$ dans \mathcal{A} avec $q \in I$ si et seulement si il existe un chemin $I \xrightarrow{w} R$ dans \mathcal{B} avec $r \in R$. Cette observation permet de montrer que $L(\mathcal{A}) = L(\mathcal{B})$. Cela se fait en 2 étapes :

1. **Tout mot de $L(\mathcal{A})$ est aussi dans $L(\mathcal{B})$.** En effet, prenons un mot w de $L(\mathcal{A})$. Il existe donc un chemin acceptant dans \mathcal{A} sur w , donc des états $q \in I$ et $r \in F$ tels que $q \xrightarrow{w} r$ dans \mathcal{A} . D'après l'observation ci-dessus, il existe un chemin $I \xrightarrow{w} R$ dans \mathcal{B} avec $r \in R$. Comme r est un état final de \mathcal{A} et $r \in R$, l'état R est final dans \mathcal{B} (d'après la définition de \mathcal{B}), donc le chemin $I \xrightarrow{w} R$ est acceptant, donc w est reconnu par \mathcal{B} , donc $w \in L(\mathcal{B})$.
2. **Tout mot de $L(\mathcal{B})$ est aussi dans $L(\mathcal{A})$.** En effet, prenons un mot w de $L(\mathcal{B})$. Il existe donc un chemin acceptant dans \mathcal{B} sur w , donc un état R final tel que $I \xrightarrow{w} R$ dans \mathcal{B} . Comme R est un état final de \mathcal{B} , il contient un état $r \in F$. D'après l'observation, il existe un calcul $q \xrightarrow{w} r$ dans \mathcal{A} avec $q \in I$. Comme r est final, ce calcul est acceptant, donc w est reconnu par \mathcal{A} : on a bien $w \in L(\mathcal{A})$.

Il reste à prouver l'observation : pour tout mot w et tout $r \in Q$, il existe un chemin $q \xrightarrow{w} r$ avec $q \in I$ dans \mathcal{A} si et seulement si il existe un chemin $I \xrightarrow{w} R$ dans \mathcal{B} avec $r \in R$. Cela se montre par récurrence sur la longueur du mot w . Si $w = \varepsilon$ et si $q \xrightarrow{w} r$ dans \mathcal{A} avec $q \in I$, alors $r = q$ (puisque w est le mot vide) donc $r \in I$. Il suffit de choisir $R = I$.

Soit maintenant w un mot non vide et supposons l'observation vraie pour tout mot de longueur inférieure. Soit $w = ua$ avec $a \in A$, c'est-à-dire que u est le préfixe de w de longueur $|w| - 1$. On va appliquer l'hypothèse de récurrence à u . On traite séparément les deux sens de l'implication :

- Supposons qu'il existe un chemin $q \xrightarrow{w} r$ dans \mathcal{A} avec $q \in I$. On isole la dernière transition de ce chemin : $q \xrightarrow{u} p \xrightarrow{a} r$. D'après l'hypothèse de récurrence, il existe P avec $p \in P$ tel que $I \xrightarrow{u} P$. Comme on a une transition $p \xrightarrow{a} r$ dans \mathcal{A} , on a une transition $P \xrightarrow{a} R$ dans \mathcal{B} avec $r \in R$ (d'après la définition des transitions de \mathcal{B}). Finalement, on a bien construit un chemin $I \xrightarrow{u} P \xrightarrow{a} R$, soit $I \xrightarrow{w} R$ avec $r \in R$, ce qui conclut ce sens.
- Inversement, supposons qu'il existe un calcul $I \xrightarrow{w} R$ dans \mathcal{B} avec $r \in R$, et montrons qu'il existe un état $q \in I$ tel que dans \mathcal{A} , on ait $q \xrightarrow{w} r$. On isole la dernière transition du calcul $I \xrightarrow{w} R$: on obtient $I \xrightarrow{u} P \xrightarrow{a} R$. Comme $r \in R$ et comme $P \xrightarrow{a} R$ est une transition de \mathcal{B} , par définition de ces transitions, on doit avoir un état $p \in P$ tel que $p \xrightarrow{a} r$ dans \mathcal{A} . On a maintenant dans \mathcal{B} un chemin $I \xrightarrow{u} P$ avec $p \in P$: par l'hypothèse de récurrence appliquée au mot u , il existe $q \in I$ tel que dans l'automate \mathcal{A} , on a un calcul $q \xrightarrow{u} p$. En recollant ce calcul et la transition $p \xrightarrow{a} r$, on obtient bien un calcul $q \xrightarrow{w} r$ dans \mathcal{A} .

Cela conclut la preuve de la proposition. ■

Exercice 4.3 1. Appliquer la construction sur l'automate de la FIGURE 3.1.

2. Sur cet automate, y a-t-il des états inutiles ? Si oui, proposer un algorithme pour éviter de construire ces états.

3. Prouver la Proposition 4.2.1 : l'automate \mathcal{B} ainsi construit reconnaît le même langage que \mathcal{A} . ■

- Exercice 4.4**
1. Si l'automate \mathcal{A} a n états, combien d'états possède son déterminisé, dans le cas le pire ?
 2. On considère le langage L_n des mots sur l'alphabet $\{a, b\}$ dont la $n^{\text{ième}}$ lettre avant la fin est un a . Donner un automate non déterministe reconnaissant L_n et ayant $n + 1$ états.
 3. Expliquer intuitivement pourquoi un automate déterministe qui reconnaît L_n doit avoir $O(2^n)$ états. ■

4.3 Propriétés de Clôture des Automates

Le fait que tout automate soit équivalent à un automate déterministe a une conséquence importante. On dit qu'un langage est *reconnaissable* s'il existe un automate qui reconnaît ce langage.

Proposition 4.3.1 Si $L \subseteq A^*$ est un langage reconnaissable, alors son complémentaire $A^* \setminus L$ est aussi reconnaissable.

Démonstration. Cela provient directement des Propositions 4.2.1 et 3.3.2. ■

Par ailleurs, si L_1 est un langage reconnu par un automate $\mathcal{A}_1 = (A, Q_1, I_1, F_1, \delta_1)$ et L_2 un langage reconnu par $\mathcal{A}_2 = (A, Q_2, I_2, F_2, \delta_2)$, le langage $L_1 \cup L_2$ est reconnu par l'automate \mathcal{A} obtenu en « juxtaposant » \mathcal{A}_1 et \mathcal{A}_2 . Formellement, on renomme les états de Q_2 pour avoir $Q_1 \cap Q_2 = \emptyset$, et l'automate \mathcal{A} est défini par $\mathcal{A} = (A, Q_1 \cup Q_2, I_1 \cup I_2, F_1 \cup F_2, \delta_1 \cup \delta_2)$.

Comme par ailleurs $L_1 \cap L_2 = A^* \setminus (A^* \setminus L_1 \cup A^* \setminus L_2)$, l'intersection de deux langages reconnaissables est encore un langage reconnaissable. En résumé, on a donc le résultat suivant :

Théorème 4.3.2 Si L_1 et L_2 sont des langages reconnaissables, alors les langages suivants sont aussi reconnaissables :

- $A^* \setminus L_1$,
- $L_1 \cup L_2$,
- $L_1 \cap L_2$.

Pour l'union et l'intersection, il existe une autre construction (qui évite le complémentaire pour l'intersection), obtenu en faisant le « produit » des automates, comme cela a été vu dans la solution de l'exercice 3.1 (cf. FIGURE 3.8). Attention cependant, cette construction peut être adaptée pour l'union seulement lorsque les automates sont complets.

Exercice 4.5 Soit $\mathcal{A}_1 = (A, Q_1, I_1, F_1, \delta_1)$ et $\mathcal{A}_2 = (A, Q_2, I_2, F_2, \delta_2)$ deux automates sur le même alphabet A . On construit les ensembles suivants :

- $Q = Q_1 \times Q_2$,
 - $I = I_1 \times I_2$,
 - $F_\cup = \{(q_1, q_2) \in Q \mid q_1 \in F_1 \text{ ou } q_2 \in F_2\}$,
 - $F_\cap = \{(q_1, q_2) \in Q \mid q_1 \in F_1 \text{ et } q_2 \in F_2\}$,
 - $\delta = \{((p_1, p_2), a, (q_1, q_2)) \in Q \times A \times Q \mid (p_1, a, q_1) \in \delta_1 \text{ et } (p_2, a, q_2) \in \delta_2\}$.
1. Montrer que l'automate $\mathcal{A}_\cap = (A, Q, I, F_\cap, \delta)$ reconnaît l'intersection des langages reconnus par \mathcal{A}_1 et \mathcal{A}_2 .

2. Montrer que si \mathcal{A}_1 et \mathcal{A}_2 sont complets, l'automate $\mathcal{A}_\cup = (A, Q, I, F_\cup, \delta)$ reconnaît l'union des langages reconnus par \mathcal{A}_1 et \mathcal{A}_2 .
3. Montrer que la propriété de la question 2 n'est plus toujours vraie si on ne suppose pas \mathcal{A}_1 et \mathcal{A}_2 complets. ■

Solution 1. On montre d'abord que $L(\mathcal{A}_1) \cap L(\mathcal{A}_2) \subseteq L(\mathcal{A}_\cap)$. Soit $w = a_1 \cdots a_n$ (avec chaque a_i une lettre) un mot dans le langage $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$. Il existe donc un calcul acceptant dans \mathcal{A}_1 sur w :

$$p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} p_2 \cdots \xrightarrow{a_n} p_n$$

(donc avec $p_0 \in I_1$ et $p_n \in F_1$), et un calcul acceptant dans \mathcal{A}_2 sur w :

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \cdots \xrightarrow{a_n} q_n.$$

(donc avec $q_0 \in I_2$ et $q_n \in F_2$). D'après la définition des transitions de \mathcal{A}_\cap , on a un calcul

$$(p_0, q_0) \xrightarrow{a_1} (p_1, q_1) \xrightarrow{a_2} p_2 \cdots \xrightarrow{a_n} (p_n, q_n)$$

dans \mathcal{A}_\cap , avec $(p_0, q_0) \in I$ et $(p_n, q_n) \in F_\cap$, donc le mot w est reconnu par \mathcal{A}_\cap .

Montrons ensuite l'inclusion inverse : $L(\mathcal{A}_\cap) \subseteq L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$. Supposons que $w = a_1 \cdots a_n$ (avec chaque a_i une lettre) est un mot du langage $L(\mathcal{A}_\cap)$. Il existe donc un calcul acceptant de \mathcal{A}_\cap sur w :

$$(p_0, q_0) \xrightarrow{a_1} (p_1, q_1) \xrightarrow{a_2} p_2 \cdots \xrightarrow{a_n} (p_n, q_n). \quad (4.1)$$

avec $(p_0, q_0) \in I$ et $(p_n, q_n) \in F_\cap$. Par définition de I , on a donc $p_0 \in I_1$ et $q_0 \in I_2$. D'après la définition de F_\cap , on a de même $p_n \in F_1$ et $q_n \in F_2$. D'après la définition des transitions de \mathcal{A}_\cap , on a un calcul de \mathcal{A}_1

$$p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} p_2 \cdots \xrightarrow{a_n} p_n$$

et un calcul de \mathcal{A}_2

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \cdots \xrightarrow{a_n} q_n,$$

tous deux acceptants, donc $w \in L(\mathcal{A}_1)$ et $w \in L(\mathcal{A}_2)$. Finalement, on a bien $w \in L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$.

2. La preuve est essentiellement la même. Le fait que \mathcal{A}_1 et \mathcal{A}_2 sont complets est utilisé pour montrer que $L(\mathcal{A}_1) \cup L(\mathcal{A}_2) \subseteq L(\mathcal{A}_\cup)$: on prend un mot $w = a_1 \cdots a_n$ (avec chaque a_i une lettre) dans le langage $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$. On sait qu'il est soit reconnu par \mathcal{A}_1 , soit reconnu par \mathcal{A}_2 . Si c'est le premier cas, par exemple, on obtient un calcul acceptant dans \mathcal{A}_1 de la forme

$$p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} p_2 \cdots \xrightarrow{a_n} p_n.$$

Si \mathcal{A}_2 n'était pas complet, il se pourrait qu'il n'y ait aucun calcul sur w dans \mathcal{A}_2 ce qui poserait problème pour continuer cette preuve. Mais comme \mathcal{A}_2 est complet, on sait qu'il existe un calcul dans \mathcal{A}_2 sur w (pas forcément acceptant), de la forme

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \cdots \xrightarrow{a_n} q_n.$$

À partir de ces deux calculs, on fabrique le calcul de \mathcal{A}_\cup

$$(p_0, q_0) \xrightarrow{a_1} (p_1, q_1) \xrightarrow{a_2} p_2 \cdots \xrightarrow{a_n} (p_n, q_n).$$

Comme p_n est final dans \mathcal{A}_1 , ce calcul est acceptant dans \mathcal{A}_\cup .

3. Si les automates ne sont pas complets, l'automate \mathcal{A}_\cup ne reconnaît pas forcément l'union des langages \mathcal{A}_1 et \mathcal{A}_2 . Par exemple, si on prend pour l'un des automates un automate qui reconnaît A^* , et pour l'autre un automate avec un seul état initial et non final, et sans aucune transition (il reconnaît donc \emptyset), on trouve pour \mathcal{A}_\cup un automate qui reconnaît \emptyset alors que l'union des langages reconnus par \mathcal{A}_1 et \mathcal{A}_2 est A^* . ■

5. Automates et Expressions Rationnelles

L'objectif de ce chapitre est de montrer le résultat suivant.

Théorème 5.0.3 Expressions rationnelles et automates décrivent **les mêmes langages**. Plus précisément

- on peut convertir algorithmiquement une expression rationnelle en automate qui reconnaît le langage décrit par l'expression,
- vice-versa, on peut convertir algorithmiquement un automate en expression rationnelle qui décrit le langage reconnu par l'automate.

Nous savons déjà que les langages rationnels « de base », c'est-à-dire les singletons $\{\varepsilon\}$ et $\{a\}$ pour $a \in A$, ainsi que le langage vide \emptyset sont reconnaissables (rappel : *reconnaissable* signifie reconnaissable par automate). Nous savons aussi que l'union de deux langages reconnaissables est encore un langage reconnaissable (voir le Théorème 4.3.2).

Ken Thompson (à gauche sur la photo, avec Dennis Ritchie) a montré que si L_1 et L_2 sont reconnaissables, alors il en est aussi de même de L_1^* et de $L_1 \cdot L_2$. Cela montre que tout langage rationnel est aussi reconnaissable. En effet, les langages de base sont reconnaissables, et l'ensemble des langages reconnaissables est stable par union, produit et étoile, c'est-à-dire par les 3 opérations rationnelles. Pour montrer ce résultat, Ken Thompson a utilisé des automates dont les transitions peuvent aussi être étiquetées par ε . Une telle transition est franchissable sans lire de lettre. Il est facile de montrer que tout langage reconnu par un automate avec de telles ε -transitions peut aussi être reconnu par un automate sans ε -transition. Autrement dit, ajouter des ε -transitions ne permet pas de décrire plus de langages. Par ailleurs, la transformation d'un automate avec ε -transitions en un automate sans ε -transition peut se réaliser algorithmiquement.



5.1 Des expressions vers les automates

5.1.1 La construction de Thompson

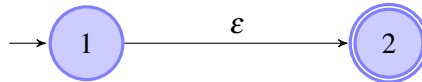
Yes we can : la remarque de Ken Thompson est qu'on peut construire pour chaque expression rationnelle un automate avec ε -transition qui reconnaît le même langage que l'expression. Cette construction maintient une propriété importante : les automates construits ont un unique état initial sur lequel n'arrive aucune transition et un unique état final duquel ne part aucune transition.

On commence d'abord par construire de tels automates pour les expressions \emptyset, a et ε . On les obtient très facilement :

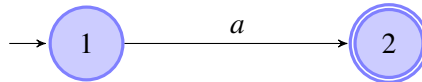
— Pour \emptyset :



— Pour ε :

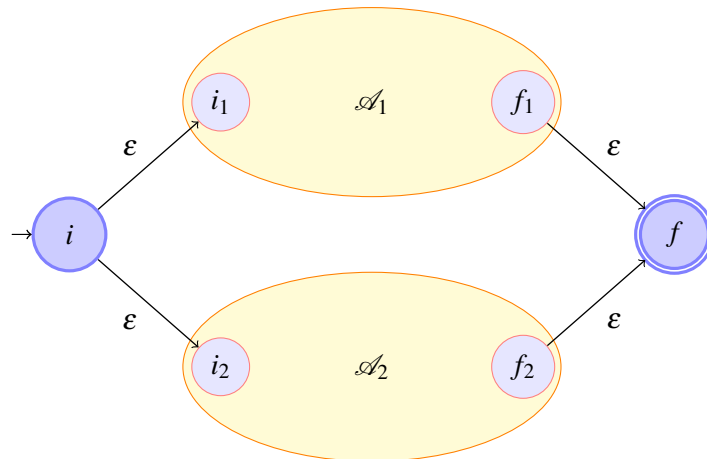


— Pour a :

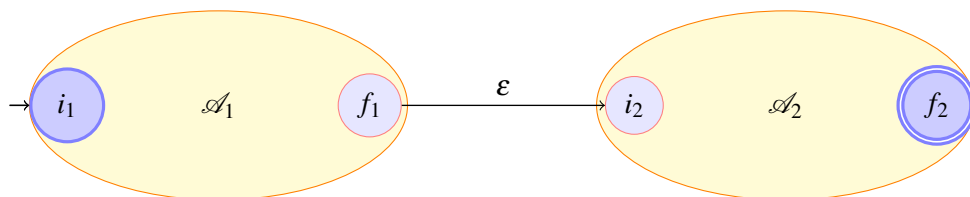


En supposant qu'on sait faire de tels automates pour e_1 et e_2 , on le fait pour $e_1 \cdot e_2$, $e_1 + e_2$, e_1^* (ce qui conduit à un algorithme récursif simple).

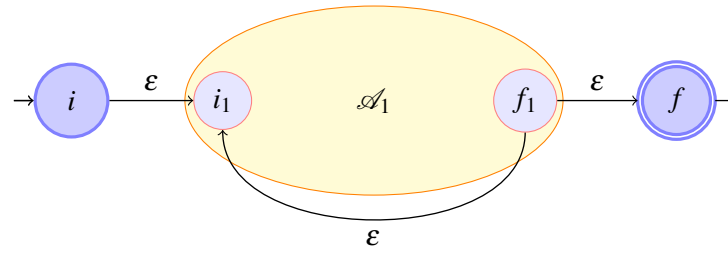
— Pour l'union :



— Pour le produit :



— Pour l'étoile :



5.1.2 La construction de Glushkov

La construction de Thompson est simple, mais elle présente l'inconvénient d'introduire beaucoup d' ε -transitions. Glushkov a trouvé une autre idée pour convertir les expressions en automates sans ε -transition. Cette construction est également simple. L'idée est la suivante : on va construire un l'automate qui mémorise, de façon non déterministe, la position dans laquelle on peut se trouver dans l'expression rationnelle après avoir lu le mot d'entrée. Par exemple, si l'expression rationnelle est aba , il y a 3 positions, ce qu'on peut indiquer en numérotant les lettres intervenant dans l'expression : $a_1b_2a_3$. La première position n'est accessible qu'après avoir lu la lettre a . A partir de cette position 1, pour atteindre la position 2, on doit lire un b . Enfin, pour atteindre la position 3, on doit lire un a . La même idée fonctionne avec n'importe quelle expression. Par exemple, pour $a(ba + c)^*$, on peut à nouveau numéroté les lettres pour rendre apparentes les 3 positions : $a_1(b_2a_3 + c_4)^*$. La première position n'est accessible qu'en lisant la lettre a . A partir de cette position, on peut atteindre soit la position 2 en lisant un b , soit la position 4 en lisant un c . À partir de la position 2, on ne peut qu'atteindre la position 3, en lisant un a . Enfin, de la position 3 comme de la position 4, on peut atteindre la position 2 par un b ou la position 4 par un c (du fait que l'on se trouve « à la fin d'une étoile », et qu'on peut revenir au début de la sous-expression). Cela donne l'idée d'un algorithme pour convertir une expression en automate.



5.2 Des automates vers les expressions

Il y a plusieurs algorithmes pour convertir un automate en expression : algorithme de McNaughton et Yamada, algorithme de J. Brzozowski et E. McCluskey, algorithme basé sur le lemme d'Arden. On présente ce dernier résultat parce qu'il est assez facile à mettre en œuvre. Le lemme d'Arden est le résultat suivant.

Proposition 5.2.1 Soient U, V, X trois langages sur un alphabet A tels que

- Le langage U ne contient pas le mot vide.
- On a

$$X = UX + V. \quad (5.1)$$

Alors $X = U^*V$.

Le Lemme d'Arden permet donc de résoudre des équations dont les inconnues (ici, X) et les constantes (ici, U, V) sont des langages. Si on admet cette proposition, on peut l'utiliser pour calculer, à partir d'un automate, une expression rationnelle qui décrit le langage reconnu par l'automate.

6. Minimisation des Automates

Ce chapitre présente l'une des propriétés les plus importantes des automates.

Bibliographie

- [Aut94] J. M. AUTEBERT. *Théorie des langages et des automates*. Masson, 1994.
- [Car08] O. CARTON. *Langages formels, Calculabilité et Complexité*. Vuibert, 2008.

Index

- Alphabet, 5, 7
- Automate, 21
 - complet, 28
 - déterminisé, 35
 - déterministe, 30
- Calcul, 21
 - acceptant, 22
- Concaténation
 - de langages, 11
 - de mots, 8
- Détermination, 33, 35
- Étoile, 12
- Langage, 5, 8
 - accepté, ou reconnu, 22
- Lettre, 5, 7
- Longueur, 7
- Monoïde libre, 7
- Mot, 5, 7
 - accepté, ou reconnu, 22
 - vide, 5, 7
- Opérations Rationnelles, 11
- Produit de concaténation, *voir* Concaténation
- Reconnaissable, 37
- Run, *voir* Calcul
 - Run acceptant, *voir* Calcul acceptant
- Union, 11