

1.1 Rappel : théorème de Kleene

1.1.1 Le théorème de Kleene

Ce que nous appellerons (de façon légèrement impropre) le **théorème de Kleene** dans ce chapitre est un résultat fondamental d'équivalence entre classes de langages, que l'on peut résumer ainsi :

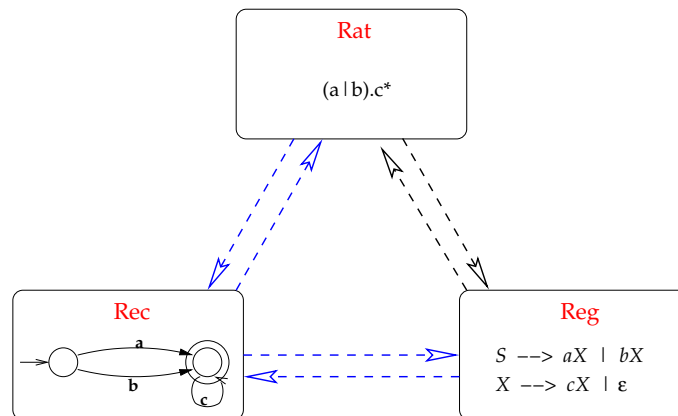
$$\mathcal{L}_{\text{Rec}} = \mathcal{L}_{\text{Rat}} = \mathcal{L}_{\text{Reg}}$$

où

- \mathcal{L}_{Rec} est la classe des langages **re**connaisables par un automate à nombre fini d'états ;
- \mathcal{L}_{Rat} est la classe des langages que l'on peut décrire avec une expression **rationnelle** ;
- \mathcal{L}_{Reg} est la classe des langages engendrés par une grammaire **régulière**.

On symbolise en général ce résultat sous la forme du triangle représenté à la figure 1.1.

FIG. 1.1 – Triangle de Kleene



La démonstration du théorème peut se faire de manière *constructive* : par exemple, pour montrer que tout langage rationnel est reconnaissable, il suffit d'exhiber un algorithme qui prenant une expression rationnelle quelconque en entrée, produit en sortie un automate qui reconnaît le même langage. Outre la difficulté de définir l'algorithme, il faut pour que la démonstration soit valide, d'une part garantir que l'algorithme fournit une réponse pour toute entrée possible, et d'autre part démontrer que l'automate fourni reconnaît bien le même langage.

Nous ne verrons pas ici ces deux derniers aspects de la démonstration (qui sont assez techniques), nous nous contenterons de donner (sous forme d'exemples) les algorithmes pour certaines des flèches pointillées de la figure (en bleu). Les algorithmes manquants existent, mais ils sont théoriquement inutiles si les deux autres équivalences sont établies.

Plus précisément, nous définirons les algorithmes permettant de démontrer :

$\mathcal{L}_{\text{Rec}} \subset \mathcal{L}_{\text{Reg}}$ Algorithme construisant une grammaire régulière à partir d'un automate, en identifiant les non-terminaux de la grammaire et les états de l'automate. (§ 1.1.2.2)

$\mathcal{L}_{\text{Reg}} \subset \mathcal{L}_{\text{Rec}}$ Algorithme très proche du précédent, toujours basé sur l'identité entre symbole non-terminal et état. (§ 1.1.2.3)

$\mathcal{L}_{\text{Rat}} \subset \mathcal{L}_{\text{Rec}}$ Algorithme basé sur la décomposition syntaxique de l'expression rationnelle, et la composition d'automates correspondants. (§ 1.1.3.1)

$\mathcal{L}_{\text{Rec}} \subset \mathcal{L}_{\text{Rat}}$ Algorithme de Mac Naughton et Yamada, qui construit itérativement, en partant de l'automate initial, un *automate fini généralisé* qui finit par ne contenir qu'une transition étiquetée par une expression rationnelle équivalente. (§ 1.1.3.2)

1.1.2 Grammaires et automates

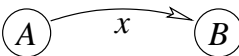


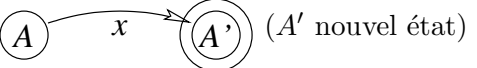
1.1.2.1 Principe

Rappel une grammaire **régulière** (dite aussi linéaire) est une grammaire dont toutes les règles de production sont sous l'une des formes suivantes¹ :

$$\begin{aligned} A &\rightarrow xB \\ A &\rightarrow x \\ A &\rightarrow \varepsilon \end{aligned}$$

Note On peut toujours proposer une définition sans ε -production, ou plutôt sans autre ε -production qu'une règle $S \rightarrow \varepsilon$, où S est l'axiome, et S est inaccessible.

Le principe de correspondance entre automates et grammaires régulières est très intuitif : il correspond à l'observation que chaque transition dans un automate produit exactement un symbole, de même que chaque dérivation dans une grammaire régulière. Le tableau 1.1 résume cette correspondance, qui donne les bases des algorithmes dans les deux sens.

	$A \rightarrow xB$
	Axiome = A
	$B \rightarrow \varepsilon$
	$A \rightarrow x$

TAB. 1.1 – Correspondances automate \leftrightarrow grammaire régulière

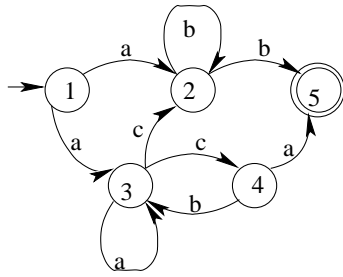
L'application de l'algorithme est illustrée à la figure 1.2 avec un automate non déterministe.

1.1.2.2 Automates \rightarrow grammaires régulières

On peut partir d'un automate quelconque (non déterministe, non complet), il suffit de considérer une à une toutes les transitions et de produire les règles correspondantes d'après le tableau 1.1.

Si l'automate contient des transitions vides, on peut bien sûr s'en débarrasser (algorithme déjà vu), ou bien les traduire en productions singulières ($A \xrightarrow{\varepsilon} B$ devient $A \rightarrow B$).

¹Où, conformément aux conventions habituelles, A et B sont des non-terminaux, et x est un symbole terminal. La définition donnée ici correspond à une grammaire linéaire/régulière **gauche**. Définition analogue possible à droite.



S_1	\rightarrow	aS_2
		aS_3
S_2	\rightarrow	bS_2
		bS_5
S_3	\rightarrow	cS_2
		cS_4
		aS_3
S_4	\rightarrow	bS_3
		aS_5
S_5	\rightarrow	ε

 FIG. 1.2 – Exemple $\text{Rec} \rightarrow \text{Reg}$ (automate non déterministe)

Mais il faut ensuite supprimer les productions singulières (qui ne sont pas permises dans une grammaire régulière), avec un algorithme qui ressemble beaucoup à l’algorithme de suppression des ε -productions dans un automate.

1.1.2.3 Grammaires régulières \rightarrow automates

Partant d’une grammaire régulière, il suffit de créer un état pour chaque non-terminal, et de “traduire” chaque règle de production en utilisant le même tableau de correspondance. Le seul cas un peu particulier concerne les règles de la forme $A \rightarrow x$, pour lesquelles il suffit de remarquer que ce sont des règles terminales (la dérivation s’arrête nécessairement dès qu’une production de cette forme est déclenchée). Il faut créer un nouvel état, terminal (A' dans le tableau). Pour comprendre cette correspondance, on peut observer que la dérivation $A \rightarrow x$ est équivalente à une dérivation avec les deux règles $A \rightarrow xA'$, et $A' \rightarrow \varepsilon$.

1.1.3 Automates et expressions rationnelles

1.1.3.1 Expression rationnelle \rightarrow Automate

On peut montrer (voir section “Propriétés de fermeture”) que la *réunion*, la *concaténation*, et l’*étoile* peuvent être définis sur les automates; il est donc possible, par exemple, de construire un automate qui reconnaît $L_1 \cup L_2$ par la réunion de l’automate qui reconnaît L_1 et de l’automate qui reconnaît L_2 . Ces considérations permettent de définir facilement un algorithme de “traduction” d’une expression rationnelle quelconque en un automate reconnaissant le même langage.

Voici cet algorithme, spécifié d’abord sous forme mathématique, puis sous la forme d’un tableau de correspondance graphique, dans le même esprit que le tableau 1.1 donné plus haut (mais orienté cette fois-ci).

Traduction récursive d’une expression rationnelle en un automate

1. Au mot vide ε , on associe l’automate $\langle X, \{q_0\}, \{q_0\}, \{q_0\}, \emptyset \rangle$
2. À l’expression rationnelle x ($x \in X$), on associe l’automate

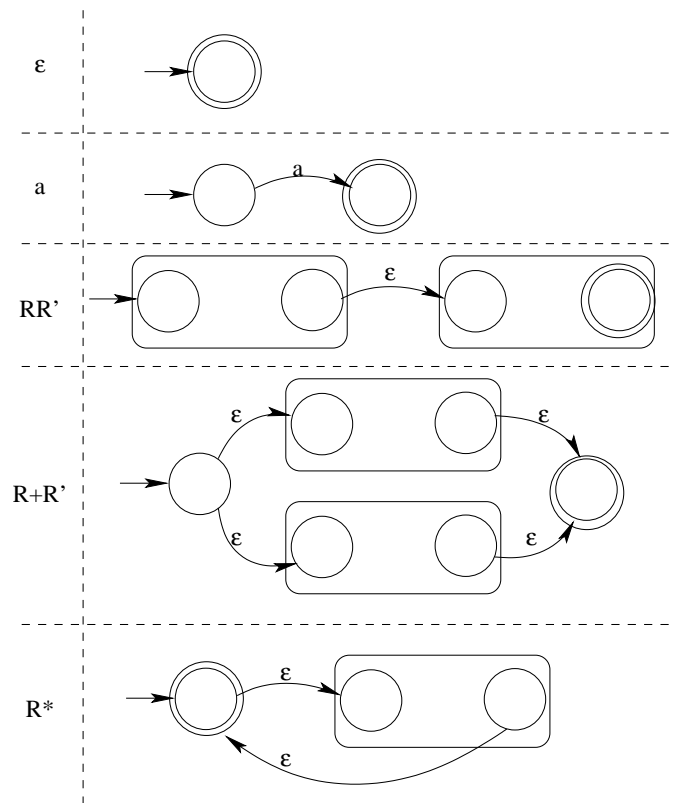
$$\langle X, \{q_0, q_1\}, \{q_0\}, \{q_1\}, \{(q_0, x, q_1)\} \rangle$$

3. Soit R une expression rationnelle, associée à l'automate $\langle X, Q_R, I_R, F_R, \delta_R \rangle$;
à R^* , on associe l'automate $\langle X, Q_R \cup \{Q_0\}, \{Q_0\}, \{Q_0\}, \delta'_R \rangle^2$,
où $\delta'_R = \delta_R \cup \bigcup_{q \in I_R} (Q_0, \varepsilon, q) \cup \bigcup_{q \in F_R} (q, \varepsilon, Q_0)$
4. Soient R et S deux expressions rationnelles auxquelles ont été associés respectivement $\langle X, Q_R, I_R, F_R, \delta_R \rangle$ et $\langle X, Q_S, I_S, F_S, \delta_S \rangle$, dont on suppose que tous les états sont distincts ($Q_S \cap Q_R = \emptyset$).
(a) À RS on associe l'automate

$$\left\langle X, Q_R \cup Q_S, I_R, F_S, \delta_R \cup \delta_S \cup \bigcup_{q \in F_R} \bigcup_{q' \in I_S} (q, \varepsilon, q') \right\rangle$$

- (b) À $R|S$ on associe l'automate

$$\left\langle X, Q_R \cup Q_S, Q_0, F_R \cup F_S, \delta_R \cup \delta_S \cup \bigcup_{q \in I_R \cup I_S} (Q_0, \varepsilon, q) \right\rangle$$



TAB. 1.2 – D'une expression rationnelle vers un automate

² Q_0 est un nouvel état t.q. $Q_0 \notin Q$.