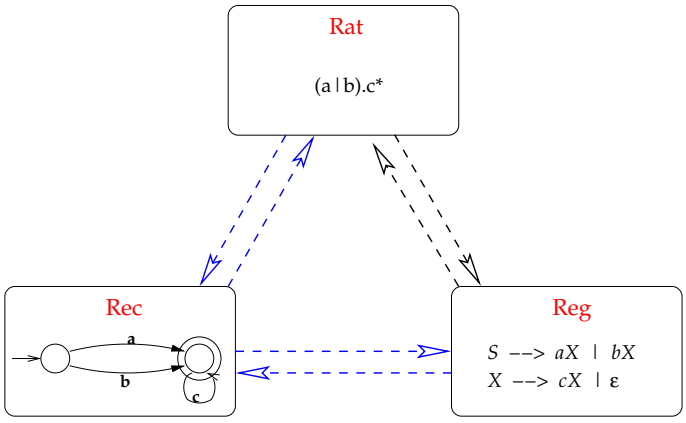
4.1 Le théorème de Kleene

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
– $\mathcal{L}_{\text{Rec}}$ est la classe des langages **rec**onnaissables par un automate à nombre fini d'états ;
– $\mathcal{L}_{\text{Rat}}$ est la classe des langages que l'on peut décrire avec une expression **rat**ionnelle ;
– $\mathcal{L}_{\text{Reg}}$ est la classe des langages engendrés par une grammaire **rég**ulière.
On symbolise en général ce résultat sous la forme du triangle représenté à la figure 4.1.

FIG. 4.1 – Triangle de Kleene



La démonstration du thèorème 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. (§ 4.2.2)
$\mathcal{L}_{\text{Reg}} \subset \mathcal{L}_{\text{Rec}}$ Algorithme très proche du précédent, toujours basé sur l'identité entre symbole non-terminal et état. (§ 4.2.3)
$\mathcal{L}_{\text{Rat}} \subset \mathcal{L}_{\text{Rec}}$ Algorithme basé sur la décomposition syntaxique de l'expression rationnelle, et la composition d'automates correspondants. (§ 4.3.1)

$\mathcal{L}_{\text{Rec}} \subset \mathcal{L}_{\text{Rat}}$ 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. (§ 4.3.2)

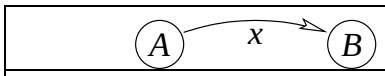
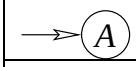
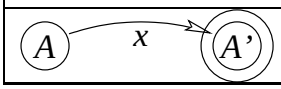
4.2 Grammaires et automates

4.2.1 Principe

Rappel une grammaire **régulière** (dite aussi linéaire) est une grammaire dont toutes les règles de production sont sous l'une des formes suivantes[1] : $A \to xB$
$A \to x$
$A \to \varepsilon$

Note On peut toujours proposer une définition sans ε-production, ou plutôt sans autre ε-production qu'une règle $S \to \varepsilon$, 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 4.1 résume cette correspondance, qui donne les bases des algorithmes dans les deux sens.

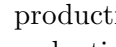
	$A \to xB$
	Axiome $= A$
	$B \to \varepsilon$
 (A' nouvel état)	$A \to x$

TAB. 4.1 – Correspondances automate ↔ grammaire régulière

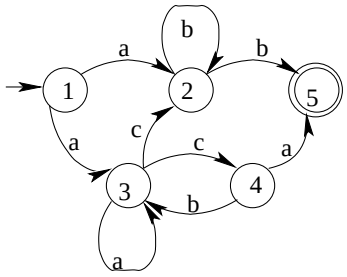
L'application de l'algorithme est illustrée à la figure 4.2 avec un automate non déterministe.

4.2.2 Automates → 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 4.1.

Si l'automate contient des transitions vides, on peut bien sûr s'en débarrasser (algorithme déjà vu), ou bien les traduire en productions singulières ( devient $A \to B$). Mais il faut ensuite supprimer les productions singulières (qui ne sont pas permises dans

[1] Où, 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.



$$
\begin{array}{lcl}
S_1 & \rightarrow & \mathrm{a}S_2 \\
 & | & \mathrm{a}S_3 \\
S_2 & \rightarrow & \mathrm{b}S_2 \\
 & | & \mathrm{b}S_5 \\
S_3 & \rightarrow & \mathrm{c}S_2 \\
 & | & \mathrm{c}S_4 \\
 & | & \mathrm{a}S_3 \\
S_4 & \rightarrow & \mathrm{b}S_3 \\
 & | & \mathrm{a}S_5 \\
S_5 & \rightarrow & \varepsilon
\end{array}
$$

FIG. 4.2 – Exemple Rec → Reg (automate non déterministe)

une grammaire régulière), avec un algorithme qui ressemble beaucoup à l'algorithme de suppression des ε-productions dans un automate.

4.2.3 Grammaires régulières → 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 $A' \rightarrow \varepsilon$.

4.3 Automates et expressions rationnelles

4.3.1 Expression rationnelle → 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 4.1 donné plus haut (mais orienté cette fois-ci).

Traduction récursive d'une expression rationnelle en un automate

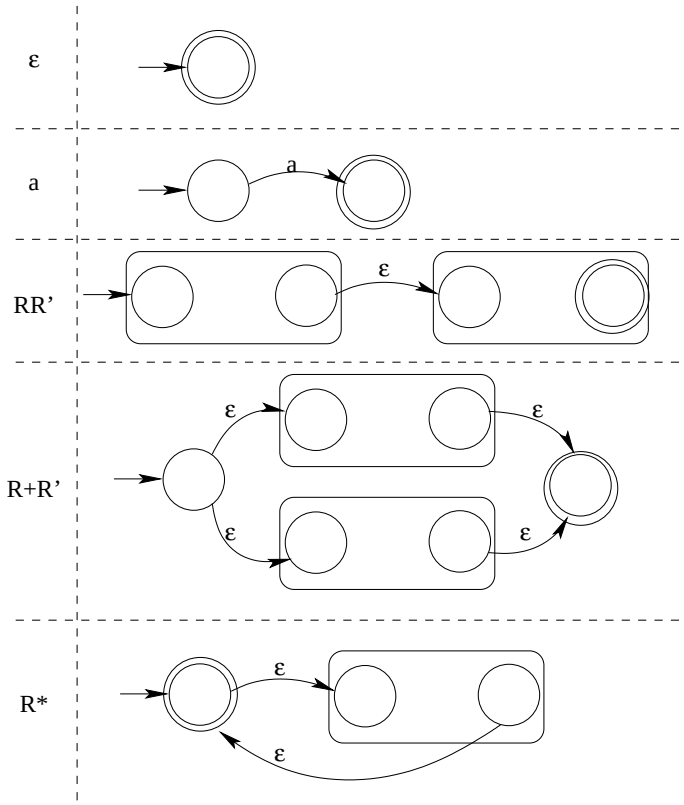
1. Au mot vide ε, on associe l'automate $\langle X, \{q_0\}, \{q_0\}, \{q_0\}, \emptyset \rangle$
2. À l'expression rationnelle x ($x \in X$), on associe l'automate
$$\langle X, \{q_0, q_1\}, \{q_0\}, \{q_1\}, \{(q_0, x, q_1)\} \rangle$$

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. 4.2 – D'une expression rationnelle vers un automate

4.3.2 Automate → expression rationnelle

Il s'agit de l'algorithme le plus sophistiqué de la série présentée ici. Nous reprenons pour le présenter un extrait de polycopié proposé par Alexis Nasr.

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

L'algorithme est divisé en deux étapes. Lors de la première étape, l'automate est transformé en un automate d'un autre type, appelé *automate (fini) généralisé*. Cet automate est ensuite transformé (itérativement) en expression régulière lors d'une seconde étape.

Un automate généralisé est un automate dont les transitions sont étiquetées par des expressions rationnelles (plus le symbole $\emptyset$, voir plus loin) et non pas simplement par des symboles ou ε. L'automate généralisé lit le mot à reconnaître par blocs de symboles.

Les automates généralisés que nous allons manipuler vérifient les contraintes suivantes :

- L'état initial possède une transision vers tous les autres états (éventuellement une transition "non passsante", étiquetée par $\emptyset$) ;
- Aucun état n'a de transition vers l'état initial
- Il existe un et un seul état d'acceptation,
  - distinct de l'état initial,
  - qui n'a aucune transition vers les autres états
  - qui est atteint par tous les autres états
- *Cf. pages suivante — copie d'un autre polycopié.*

– A l'exception de l'état d'acceptation et de l'état initial, tous les états possèdent une transition et une seule vers tous les autres états.

Un exemple d'automate généralisé est représenté dans la figure 2.11

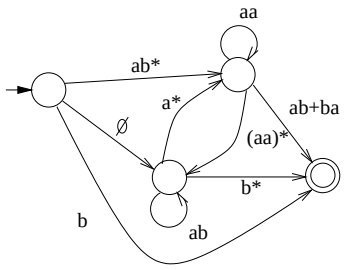


FIG. 2.11 – Un automate généralisé

Il est facile de construire un automate généralisé à partir d'un automate, il suffit pour cela d'ajouter un nouvel état initial possèdant une transition-ε vers l'ancien état initial et un nouvel état d'acceptation vers lequel il existe une transition-ε partant des anciens états d'acceptation. S'il existe plusieurs transitions entre deux états, elles sont remplacées par une transition unique étiquetée par l'union des étiquettes des différentes transitions. Finalement, des transitions étiquetées $\emptyset$ sont ajoutées entre les états qui ne sont reliés par aucune transition. Cet ajout ne modifie pas le langage reconnu par l'automate car une transition étiquetée $\emptyset$ ne peut jamais être franchie. L'automate généralisé correspondant à l'automate de la figure 2.1 est représenté dans la figure 2.12.

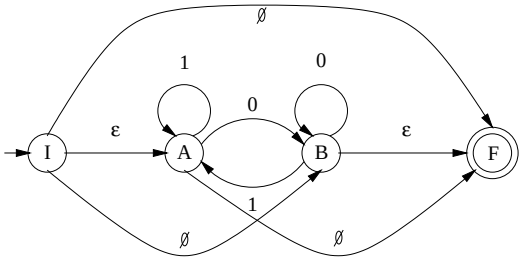


FIG. 2.12 – Transformation d'un automate en automate généralisé

Il reste maintenant à construire une expression régulière à partir d'un automate généralisé. Cette construction s'effectue de manière itérative en diminuant d'un, à chaque itération, le nombre d'états de l'automate généralisé. A l'issue de cette étape, on obtient un automate généralisé comportant deux états (l'état initial et l'état d'acceptation) et une transition entre les deux. L'expression régulière étiquetant cette transition dénote le langage de l'automate initial.

Il nous reste à décrire l'étape de réduction de l'automate généralisé qui consiste donc à transformer un automate généralisé G comportant k états (avec $k > 2$) en un automate G' comportant $k-1$ états. Le principe consiste à choisir un état de G, que nous appellerons q_e à l'éliminer et à

"réarranger" le reste des transitions de façon à ce que G et G' reconnaissent le même langage. Le principe du réarrangement est le suivant : s'il existe dans G une transition de l'état q_1 vers l'état q_e étiquetée R_1 et une transition de q_e vers lui-même étiqueté R_2 et finalement une transition de q_e vers q_2 étiquetée R_3, alors on crée une nouvelle transition, allant de q_1 vers q_2 étiquetée avec l'expression régulière suivante :

$$R_1 R_2^* R_3 + R_4$$

où R_4 est l'étiquette de la transition qui existait dans G entre q_1 et q_2. Cette élimination d'état est représentée dans la figure 2.13.

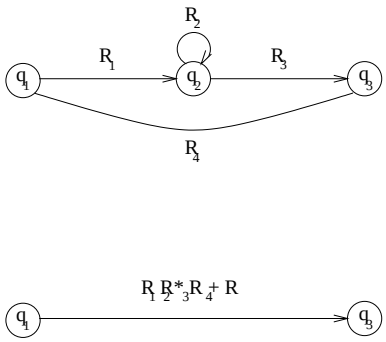


FIG. 2.13 – Elimination d'un état d'un automate généralisé

L'application de cette méthode à l'automate de la figure 2.12 est représentée dans la figure 2.14. Dans un premier temps l'état A est éliminé et dans un second temps l'état B. L'expression régulière résultante est $1^*0(0 + 11^*0)^*$ que l'on peut transformer en $1^*0((\varepsilon + 11^*)0)^*$ puis en $1^*0(1^*0)^*$ qui est équivalente à l'expression régulière $(1 + 0)^*0$.

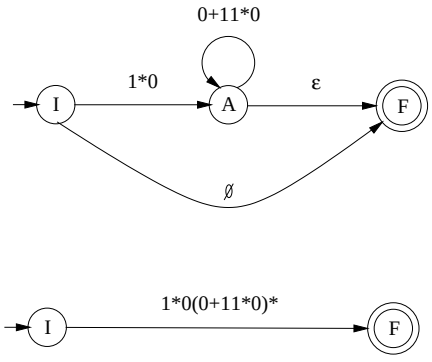


FIG. 2.14 – Elimination des états A et B