IF114 – Automates finis et applications

Frédéric Herbreteau
ENSEIRB-MATMECA (Bordeaux INP)

27 février 2024

Table des matières

Table des figures

Remerciements

Je tiens à remercier Laurent Bienvenu, Astrid Casadei, Sylvain Lombardy, Yves Métivier, David Renault, Nasser Saheb et Hayssam Soueidan pour leurs précieuses remarques qui ont contribué à améliorer substantiellement ce document. Un grand merci également aux étudiant(e)s qui ont suivi cet enseignement, et qui, au fil des ans, ont permis d'améliorer ce document par leurs remarques.

Bibliographie

Le cours et les exercices sont inspirés, et pour certains issus, des deux ouvrages suivants :

— *Introduction to Automata Theory, Languages, and Computation.* John E. Hopcroft, Rajeev Motwani et Jeffery D. Ullman, Prentice Hall, 2007 (3ème édition).

(Voir http://infolab.stanford.edu/~ullman/ialc.html)

— *Introduction à la calculabilité.* Pierre Wolper, InterEditions, 2006 (3ème edition).

Chapitre 1

Automates finis et langages

En informatique, il est souvent nécessaire de vérifier qu'un texte est conforme à un modèle. Par exemple, un compilateur vérifie qu'un programme respecte bien la syntaxe du langage de programmation. Un formulaire électronique valide les données saisies par l'utilisateur (dates, nombres, etc). Cette tâche s'appelle l'**analyse syntaxique** (voir chapitre 7). Des algorithmes sont nécessaires pour analyser et (in-)valider un texte. Ces algorithmes sont le plus souvent décrits sous la forme de machines à états : les **automates**. Un automate lit séquentiellement un texte qu'il accepte ou rejette suivant qu'il est conforme ou non au modèle attendu. L'automate est souvent lui-même calculé automatique à partir d'une description du modèle. Par exemple, l'automate utilisé par un compilateur de langage C est calculé automatiquement à partir d'une **grammaire** (voir chapitre 5) qui décrit les programmes C syntaxiquement corrects. Les outils `grep` et `sed` calculent un automate fini pour les motifs qu'ils doivent identifier à partir d'**expressions régulières** (voir chapitre 3).

Plus généralement, les automates sont des algorithmes qui permettent de résoudre une variété particulière de problèmes : les **problèmes de décision**. Ceux-ci admettent pour réponse «oui» ou «non». Un problème de décision est donc énoncé sous la forme d'une **question générique** s'appliquant à un ensemble d'éléments E. La question partitionne l'ensemble E en une partie E_{oui} pour lesquels la réponse à la question est «oui» ou **positive**, et une partie E_{non} pour la quelle la réponse est «non» ou **négative**.

Exemple 1.1 «Déterminer si un nombre naturel est pair» est un problème de décision appliqué à l'ensemble $E = \mathbb{N}$. La partie $E_{oui} = \{2n \mid n \in \mathbb{N}\}$ est l'ensemble des nombres pairs, et la partie $E_{non} = \{2n + 1 \mid n \in \mathbb{N}\}$ est l'ensemble des nombres impairs. ♦

Une **instance** d'un problème de décision pose la question pour un élément particulier x de E. Toute instance **admet une réponse**, qui est positive lorsque $x \in E_{oui}$ et négative lorsque $x \in E_{non}$.

Exemple 1.2 Reprenons l'exemple 1.1. Une instance de ce problème consiste à poser la question de la parité pour un entier naturel donné, par exemple «32 est-il un nombre pair ?». Cette question admet une réponse : 32 est pair puisque $32 \in E_{oui}$. ♦

En informatique, il existe d'autres problèmes d'intérêt que les problèmes de décision. Cependant, tout problème peut s'exprimer sous la forme d'un problème de décision. Cette approche permet de définir une théorie du calcul dans un cadre simple et unique. La **calculabilité** d'un langage L s'intéresse à l'existence d'un automate (d'un type particulier) capable de

résoudre le problème de décision «$x \in L$? »pour toute entrée x. La calculabilité est étudiée en 2ème année dans le cours «calculabilité et complexité ».

> ▷ La notion de problème est **indépendante de la notion d'algorithme**. Un algorithme peut résoudre un problème, mais il ne le définit pas. Par ailleurs, un problème est généralement résolu par plusieurs algorithmes.

1.1 Les langages

Ce cours s'intéresse donc à des problème de décision qui demandent si une entrée x appartient ou non à un certain ensemble E. Il est nécessaire de pouvoir décrire cet ensemble E. L'entrée x sera soumise à un automate qui la lira **séquentiellement**. On encode donc x par une séquence de symboles (un mot). L'ensemble E est donc constitué de mots : c'est un langage. L'automate fini aura donc pour rôle de décider si le mot x appartient au langage E.

Définition 1.1 (Alphabet) Un **alphabet** est un ensemble **fini** et **non vide** de symboles.♦

Exemple 1.3 Les ensembles $\{a, b, c\}$, $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ et $\{\text{do}, \text{ré}, \text{mi}, \text{fa}, \text{sol}, \text{la}, \text{si}\}$ sont des alphabets. ♦

Un alphabet est généralement désigné par la lettre grecque Σ, et ses dérivées : Σ_1, Σ', etc. Nous pouvons donc maintenant construire des séquences de symboles.

Définition 1.2 (Mot) Un **mot** sur un alphabet Σ est une séquence finie de symboles de Σ.♦

Le nombre de symboles constituant le mot définit sa longueur.

Définition 1.3 (Longueur d'un mot) La **longueur** d'un mot $w = w_1 \dots w_n$, avec $w_i \in \Sigma$, est définie par $|w| = n$. ♦

Enfin, nous considérons une séquence particulière qui ne contient aucun symbole :

Définition 1.4 (Mot vide) Le **mot vide** noté ε est le mot de longueur nulle : $|\varepsilon| = 0$. ♦

Exemple 1.4

1. «*a*», «*bb*», «*abc*» sont des exemples de mots sur l'alphabet $\{a, b, c\}$.
2. «123», «0», «2003» sont des mots sur l'alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
3. «do fa sol la» est un exemple de mot sur l'alphabet $\{\text{do}, \text{ré}, \text{mi}, \text{fa}, \text{sol}, \text{la}, \text{si}\}$.
4. «*abcd* » n'est pas un mot sur l'alphabet $\{a, b, c\}$
5. «$-12.45e + 37$ n'est pas un mot sur l'alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. En ajoutant les symboles $\{-, ., e, +\}$ à l'alphabet des chiffres, nous obtenons un nouvel alphabet sur lequel $-12.45e + 37$ est un mot ♦

Deux mots peuvent servir à en former un troisième en les accolant :

Définition 1.5 (Concaténation de mots) La **concaténation** de deux mots $w = w_1 \dots w_n$ sur Σ, et $w' = w'_1 \dots w'_k$ sur Σ', est définie par le mot $w \cdot w' = w_1 \dots w_n w'_1 \dots w'_k$ sur $\Sigma \cup \Sigma'$.
La longueur de $w.w'$ est donnée par : $|w.w'| = |w| + |w'| = n + k$. ♦

Notons que le mot vide est l'élément neutre pour l'opération de concaténation : pour tout mot w, nous avons : $w \cdot \varepsilon = \varepsilon \cdot w = w$.

Définition 1.6 (Sous-facteur, préfixe, suffixe) Un mot w' est un **sous facteur** de w s'il existe deux mots u et v tels que $w = u \cdot w' \cdot v$. Un sous facteur w' est propre si $w' \neq \varepsilon$.

Nous appelons **préfixe** de w tout sous facteur w' tel que $u = \varepsilon$, *i.e.* $w = w' \cdot v$. Symétriquement, un **suffixe** de w est un sous facteur w' tel que $v = \varepsilon$, *i.e.* $w = u \cdot w'$. ⧫

L'instance x d'un problème de décision sera donc représentée par un mot. Un problème de décision est alors défini comme l'ensemble E de ses instances x, donc comme un ensemble de mots.

Définition 1.7 (Langage) Un **langage** sur un alphabet Σ est un ensemble fini ou infini de mots sur Σ. ⧫

Nous notons Σ^* l'ensemble des mots sur Σ. Un langage sur Σ est donc une partie de Σ^*.

Exemple 1.5

1. $\{a \cdot w \,|\, w \in \{a, b, c\}^*\}$ est le langage des mots sur $\{a, b, c\}$ commençant par le symbole a.
2. $\{w \cdot 0, w \cdot 2, w \cdot 4, w \cdot 6, w \cdot 8 \,|\, w \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^*\}$ est le langage qui encode l'ensemble des nombres pairs en base décimale.
3. Soit $\Sigma = \{\text{do}, \text{ré}, \text{mi}, \text{fa}, \text{sol}, \text{la}, \text{si}\}$, $\{w \in \Sigma^* \,|\, \forall i \in [1..|w|],\, w_i \neq \text{mi}\}$ est le langage des séquences finies de notes sans «mi». ⧫

> ▷ Attention à ne pas confondre le langage vide $\emptyset$, et le langage (non vide) contenant uniquement le mot vide : $\{\varepsilon\}$.

Les langages que nous venons d'introduire fournissent un bon modèle pour les problèmes. Chaque instance x est modélisée par un mot w. Le problème est alors le langage L formé par l'ensemble E de ses instances. Notons que **la description**, **et non pas sa définition**, de l'ensemble E_{oui} des instances positives et la description E_{non} de l'ensemble des instances négatives, dépendent de l'encodage choisi.

Exemple 1.6 Reprenons le problème de parité de l'exemple 1.1. Nous pouvons choisir l'encodage décimal proposé dans le point 2 de l'exemple 1.5. Dans ce cas, E_{oui} est décrit par le langage L des mots de la forme $w \cdot 0$, $w \cdot 2$, $w \cdot 4$, $w \cdot 6$ et $w \cdot 8$ où w est un mots sur l'alphabet des chiffres $\{0, \ldots, 9\}$. Inversement, E_{non} est décrit par le complémentaire de L : l'ensemble des mots de la forme $w \cdot 1$, $w \cdot 3$, $w \cdot 5$, $w \cdot 7$ et $w \cdot 9$.

Si maintenant nous représentons les nombres en base binaire, *i.e.* sur l'alphabet $\{0, 1\}$, E_{oui} est représenté par le langage L contenant l'ensemble des mots de la forme $w \cdot 0$ où w est un mot sur $\{0, 1\}$. Inversement, E_{non} est représenté par l'ensemble des mots sur $\{0, 1\}$ de la forme $w \cdot 1$. ⧫

(a) A_1 (b) A_2

FIGURE 1.1 – Deux automates finis.

1.2 Automates finis

Nous savons maintenant comment représenter un problème de décision par un langage sur un alphabet choisi : l'ensemble E de ses instances x, chacune d'elle représentée par un mot w sur cet alphabet. L'ensemble des instances positives E_{oui} est lui aussi représenté par un langage L, tout comme l'ensemble des instances négatives E_{non} qui est représenté par le complémentaire de L. Un algorithme qui résout un problème donné prend en entrée une instance particulière x, et doit décider si $x \in E_{oui}$, c'est à dire si $w \in L$. Pour cela, l'algorithme doit lire certains symboles du mot w qui lui permettent de prendre sa décision. Un **modèle d'algorithme**, ou **modèle de calcul**, doit donc permettre de lire un mot w en entrée, de mémoriser certaines informations (par exemple, le symbole lu, ou encore l'égalité des deux derniers symboles lus, etc), et prendre des décisions pour, *in fine*, décider si $w \in L$.

Nous introduisons maintenant les automates finis comme modèle de calcul. Un automate fini lit le mot d'entrée w, symbole par symbole, de gauche à droite et sans pouvoir revenir en arrière. L'automate est une machine à états qui en représentent la mémoire. L'état de l'automate change pour refléter l'information qu'il mémorise en fonction des symboles de w qu'il lit. Il possède certains états particuliers. Les états **initiaux** correspondent aux connaissances élémentaires qu'il possède avant de lire w. Les états **accepteurs** correspondent aux connaissances permettant de décider si $w \in L$. C'est à dire, si après avoir lu w l'automate se trouve dans un état accepteur alors w est accepté, c'est à dire $w \in L$.

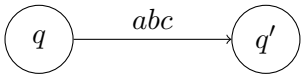
Définition 1.8 (Automate fini) Un **automate fini** $A = (Q, \Sigma, \delta, I, F)$ est défini par :
- Q est un ensemble fini d'états.
- Σ est l'alphabet de A.
- $\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ est la relation de transition.
- $I \subseteq Q$ est l'ensemble des états initiaux.
- $F \subseteq Q$ est l'ensemble des états accepteurs. ♦

Exemple 1.7 L'automate fini en figure 1.1(a) possède 2 états $Q = \{q_0, q_1\}$; q_0 est initial (*i.e.* $I = \{q_0\}$) et q_1 est accepteur (*i.e.* $F = \{q_1\}$). Son alphabet est $\Sigma = \{0, 1\}$. Enfin, sa relation de transition est $\delta = \{(q_0, 0, q_1), (q_0, 1, q_0), (q_1, 0, q_1), (q_1, 0, q_0)\}$.

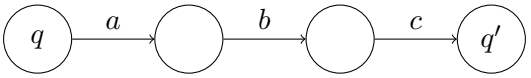
Graphiquement, les états initiaux sont représentés par une flèche entrante issue d'aucun état. Les états accepteurs sont représentés par un double cercle, mais aussi parfois par une flèche sortante pointant vers aucun état.

La figure 1.1(b) représente un automate à 2 états $Q = \{q_0, q_1\}$ ayant $I = \{q_0\}$ pour unique état initial, et $F = \{q_1\}$ pour seul état accepteur. Son alphabet est $\Sigma = \{0, 1\}$ et sa relation de transition est $\delta = \{(q_0, 0, q_0), (q_0, 0, q_1), (q_0, 1, q_0)\}$. ♦

La définition des automates finis stipule que les transitions sont étiquetées soit par un symbole de Σ, soit par le mot vide ε. Plus généralement, les transitions d'un automate fini peuvent être étiquetées par un mot. En effet, une transition :



est aisément transformée en une séquence de transitions, chacune étiquetée par un symbole :



Nous nous permettrons donc dans la suite de noter $q \xrightarrow{w} q'$ l'existence d'une séquence de transitions dans A, menant de q à q' et étiquetée par w.

> ▷ Notons qu'en présence d'une transition étiquetée ε l'automate change d'état sans lire aucun symbole du mot d'entrée. Nous en verrons les conséquences au chapitre 2.

Certaines propriétés de la relation de transition δ d'un automate fini conduisent à des classes particulières d'automates finis. La relation de transition δ peut être vue comme une fonction de transition $Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ qui associe à un état et un symbole donnés, un ensemble d'états successeurs. Si cette fonction est **totale**, alors l'automate est complet :

Définition 1.9 (Automate fini complet) Un automate fini $A = (Q, \Sigma, \delta, I, F)$ est **complet** si pour tout $q \in Q$ et pour tout $s \in \Sigma$ il existe $q' \in Q$ tel que $(q, s, q') \in \delta$. ⧫

Un automate fini complet possède une transition pour chaque lettre de l'alphabet depuis tout état. Il peut donc traiter tout mot d'entrée entièrement, sans jamais être «bloqué » avant d'avoir lu l'intégralité du mot d'entrée. Cependant, cela ne suffit pas pour pouvoir programmer l'automate. Il faut de plus que celui-ci soit déterministe. C'est à dire que connaissant l'état de l'automate et la lettre lu, il doit être possible de déterminer l'état suivant de manière unique.

Définition 1.10 (Automate fini déterministe) Un automate fini $A = (Q, \Sigma, \delta, I, F)$ est **déterministe** si et seulement si :

- il possède un unique état initial : $I = \{q_0\}$;
- il n'existe pas de transition étiquetée ε : $(Q \times \{\varepsilon\} \times Q) \cap \delta = \emptyset$;
- et pour tout état $q \in Q$, pour tout $s \in \Sigma$, il existe **au plus** un état $q' \in Q$ tel que $(q, s, q') \in \delta$. ⧫

Intuitivement, si un automate est déterministe, alors à partir d'un état q nous pouvons déterminer l'état q' dans lequel se trouve l'automate lors de la lecture d'un symbole s. Ceci n'est pas le cas s'il existe $q', q'' \in Q$ tels que $(q, s, q') \in \delta$ et $(q, s, q'') \in \delta$ puisqu'il peut se trouver aussi bien en q' qu'en q''. De même si $(q, \varepsilon, q') \in \delta$, l'automate se trouve aussi bien dans q que dans q', son état ne peut pas être déterminé. Enfin, si l'automate possède plusieurs états initiaux, nous ne pouvons pas déterminer à partir de quel état débuter le calcul.

Les automates finis déterministes peuvent être programmés [1] : ce sont des algorithmes. Les automates finis déterministes et complets sont plus simples à programmer puisqu'ils peuvent traiter l'intégralité du mot d'entrée, sans bloquer.

Les **automates finis déterministes** sont abrégés AFD, et les **automates finis non-déterministes** sont abrégés AFN. Ces derniers sont étudiés en détail au chapitre 2.

Exemple 1.8 L'automate en figure 1.1(a) est déterministe et complet puisque en chacun de ses états, et pour chaque symbole de l'alphabet, il existe un et seulement un état en relation par δ (qui est donc une fonction totale). Par ailleurs, il n'a aucune transition ε. Enfin, il n'a qu'un seul état initial.

Par contre, l'automate en figure 1.1(b) est non-déterministe puisque en l'état q_0, et pour le symbole 0, il existe deux transitions dans δ : $q_0 \xrightarrow{0} q_0$ et $q_0 \xrightarrow{0} q_1$. Par ailleurs, il n'est pas complet puisque depuis l'état q_1, il n'existe pas de transition (*i.e.* ni pour 0, ni pour 1). ♦

▷ Lorsque l'on considère un automate avec un seul état initial q_0, en particulier un automate fini déterministe, on s'autorisera à écrire directement $A = (Q, \Sigma, \delta, q_0, F)$.

1.2.1 Langage accepté et langage reconnu

Un automate fini est appliqué à un mot d'entrée donné, encodé dans l'alphabet de l'automate. C'est à dire que l'automate, à partir d'un état initial, lit le mot d'entrée lettre à lettre et il franchit, pour chacune d'elles, une transition de même étiquette. Ceci définit une exécution de l'automate fini.

Définition 1.11 (Exécution) Une exécution d'un automate fini $A = (Q, \Sigma, \delta, I, F)$ sur un mot d'entrée $w \in \Sigma^*$ est une séquence finie de transitions :

$$q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \cdots q_n \xrightarrow{a_n} q_{n+1} \tag{1.1}$$

telle que : $q_0 \in I$, $w = a_0 a_1 \dots a_n$, et pour tout $i \in [0; n]$: $(q_i, a_i, q_{i+1}) \in \delta$. ♦

Notons que pour tout mot d'entrée, toute exécution d'un automate fini est finie. En d'autres termes, toute exécution d'un automate fini termine. Conformément à notre notation pour les séquences de transitions, nous noterons $q_0 \xrightarrow{w} q$ l'existence d'une exécution [2] pour le mot w aboutissant en q.

Exemple 1.9 Considérons l'automate A_1 de la figure 1.1(a) :
- $q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_1$ est une exécution sur le mot d'entrée 00
- $q_0 \xrightarrow{1} q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_0$ est une exécution sur le mot d'entrée 1101
- $q_1 \xrightarrow{1} q_0 \xrightarrow{0} q_1$ n'est pas une exécution car la séquence débute en $q_1 \notin I$
- $q_0 \xrightarrow{0} q_1 \xrightarrow{0} q_0 \xrightarrow{1} q_0$ n'est pas une exécution car $(q_1, 0, q_0) \notin \delta$

1. À l'heure actuelle, nous ne disposons pas d'ordinateurs non-déterministes.
2. C'est à dire une séquence de transitions issue d'un état initial q_0.

— $q_0 \xrightarrow{1101} q_0$ est une exécution puisque la séquence de transitions correspondante existe (voir ci-dessus)
— q_0 est une exécution sur le mot d'entrée ε (on part de l'état initial q_0 et on ne lit aucun symbole)

Considérons maintenant l'automate A_2 de la figure 1.1(b) :
— $q_0 \xrightarrow{0} q_0 \xrightarrow{0} q_0$ et $q_0 \xrightarrow{0} q_0 \xrightarrow{0} q_1$ sont deux exécutions sur le mot d'entrée 00 (et ce sont les deux seules)
— $q_0 \xrightarrow{1} q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_0$ est une exécution sur le mot d'entrée 1101, mais $q_0 \xrightarrow{1} q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_1$ n'en est pas une puisque $(q_1, 1, q_1) \notin \delta$. Tous les choix offerts par le non-déterminisme ne sont donc pas équivalents.
— q_0 est une exécution sur le mot d'entrée ε (on part de l'état initial q_0 et on ne lit aucun symbole) ⧫

> ▷ Un automate fini déterministe admet donc au plus une exécution pour tout mot d'entrée. A contrario, un automate fini complet admet au moins une exécution pour tout mot d'entrée. Finalement, un automate fini déterministe et complet admet exactement une exécution pour tout mot d'entrée.

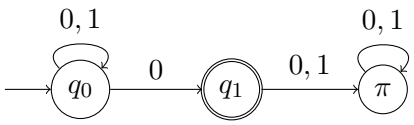
Une exécution d'un automate fini (voir équation 1.1) pour un mot d'entrée w aboutit dans un état donné, ici q_{n+1}. C'est alors que l'automate **rend une décision** : soit $q_{n+1} \in F$ est accepteur et alors w est accepté. Soit au contraire $q_{n+1} \notin F$ n'est pas accepteur, et le mot w est rejeté.

> ▷ C'est en cela que les automates finis représentent des algorithmes : ils ne modélisent pas seulement des exécutions, mais ils effectuent un calcul en acceptant ou en rejetant le mot d'entrée. Ils rendent donc une réponse à un problème de décision, et y constituent donc une solution algorithmique.

Définition 1.12 (Langage accepté par un automate) Un langage L sur l'alphabet Σ est **accepté** par un automate fini $A = (Q, \Sigma, \delta, I, F)$ si pour tout mot $w \in L$, il existe une exécution acceptante de A sur w. ⧫

Exemple 1.10 D'après l'exemple 1.9, l'automate A_1 de la figure 1.1(a) accepte notamment le mot 00. L'automate A_2 de la figure 1.1(b) accepte lui aussi le mot 00. En effet, bien que l'exécution $q_0 \xrightarrow{0} q_0 \xrightarrow{0} q_0$ n'aboutisse pas dans un état accepteur (*i.e.* $q_0 \notin F = \{q_1\}$), il existe une exécution : $q_0 \xrightarrow{0} q_0 \xrightarrow{0} q_1$ pour le mot 00 qui termine dans un état accepteur.

Notons que ces deux automates acceptent plus généralement tous les mots se terminant par 0, c'est à dire les encodages binaires des entiers pairs. ⧫



⧫

FIGURE 1.2 – eqcomplet(A_2) pour l'automate A_2 en figure 1.1(b).

Définition 1.13 (Langage reconnu par un automate) Le langage **langage reconnu** par un automate fini $A = (Q, \Sigma, \delta, I, F)$ est l'ensemble des mots w pour lesquels il existe une exécution de A qui aboutit dans un état accepteur :

$$\mathcal{L}(A) = \{w \in \Sigma^* \mid q_0 \xrightarrow{w} q_F \quad \text{avec} \quad q_0 \in I \quad \text{et} \quad q_F \in F\}$$ ⧫

Un langage L est donc accepté par un automate A si tout les mots de L sont acceptés par A. Cependant, A peut également accepter des mots qui n'appartiennent pas à L. Si deux langages L_1 et L_2 sont acceptés par A, alors $L_1 \cup L_2$ est accepté par A. Le langage reconnu par A est le plus grand langage accepté par A (pour l'inclusion). Ainsi, tout mot de $\mathcal{L}(A)$ est accepté par A, et tout qui n'appartient pas à $\mathcal{L}(A)$ est rejeté par A.

Exemple 1.11 L'automate A_1 de la figure 1.1(a) accepte notamment le langage $\{0\}$ constitué du seul mot 0, ainsi que le langage de tous les mots contenant deux 1 et terminés par 0 (notons qu'il s'agit d'un langage contenant une infinité de mots).

Aucun de ces deux langages n'est cependant reconnu par A puisque A accepte le mot 10 qui ne leur appartient pas. Le langage reconnu par A_1 est l'ensemble des mots sur l'alphabet $\{0, 1\}$ qui se terminent par 0 (l'encodage en binaire des entiers pairs). ⧫

1.2.2 Complétude et complémentation

Nous montrons maintenant que tout langage reconnu par un automate fini est également reconnu par un automate fini complet.

Définition 1.14 (Complétude) Soit $A = (Q, \Sigma, \delta, I, F)$ un automate fini. La **complétude** de A est l'automate fini eqcomplet(A) $= (Q', \Sigma, \delta', I, F)$ défini par :

— $Q' = Q \cup \{\pi\}$ où $\pi \notin Q$ est un nouvel état,

— et $\delta' = \delta \cup \{(q, s, \pi) \mid q \in Q,\ s \in \Sigma \text{ t.q. } \forall q' \in Q\, (q, s, q') \notin \delta\} \cup \{(\pi, s, \pi) \mid s \in \Sigma\}$. ⧫

L'automate eqcomplet(A) correspond à l'automate A auquel un nouvel état π, nommé **état puits**, a été ajouté. Lorsqu'il n'existait pas de transition issue de q, de symbole s dans A, une transition $q \xrightarrow{s} \pi$ a également été ajoutée. Ainsi, alors que la lecture du symbole s bloquait A dans l'état q, elle fait arriver eqcomplet(A) dans l'état π où il reste désormais «coincé» (comme dans un puits) quelle que soit la lettre lue.

Exemple 1.12 L'automate A_2 en figure 1.1(b) n'est pas complet puisqu'il n'y a pas de transitions issues de l'état q_1. L'automate eqcomplet(A_2) est représenté en figure 1.2.

In fine eqcomplet(A) rejette tout mot dont la lecture l'amène dans π rendant ainsi le même verdict que A. En effet, l'état puits π n'est pas accepteur et aucune transition ne permet d'aller vers un autre état que π.

Lemme 1.1 (Correction de la complétude) Pour tout automate fini A, $\mathsf{eqcomplet}(A)$ est un automate fini complet tel que $\mathcal{L}(A) = \mathcal{L}(\mathsf{eqcomplet}(A))$. ⧫

Preuve. La complétude de $\mathsf{eqcomplet}(A)$ vient de la définition de δ' en définition 1.14. En effet, pour tout $q \in Q$ et $s \in \Sigma$, soit il existe $q' \in Q$ tel que $(q, s, q') \in \delta$ donc $(q, s, q') \in \delta'$, soit $(q, s, \pi) \in \delta'$. D'autre part, $(\pi, s, \pi) \in \delta'$ pour tout $s \in \Sigma$.

Nous montrons maintenant que $\mathcal{L}(A) = \mathcal{L}(\mathsf{eqcomplet}(A))$. Supposons que $w \in \mathcal{L}(A)$ avec $w = w_1 w_2 \cdots w_n$ et $w_i \in \Sigma$ pour tout $i \in [1; n]$. Alors, il existe une exécution acceptante dans A :

$$q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} \cdots \xrightarrow{w_n} q_n \qquad \text{avec } q_0 \in I \text{ et } q_n \in F$$

Sachant que $\delta \subseteq \delta'$, cette exécution est également acceptante dans $\mathsf{eqcomplet}(A)$.

Considérons maintenant $w \in \mathcal{L}(\mathsf{eqcomplet}(A))$. Il existe donc une exécution acceptante dans $\mathsf{eqcomplet}(A)$:

$$q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} \cdots \xrightarrow{w_n} q_n \qquad \text{avec } q_0 \in I \text{ et } q_n \in F$$

Supposons que l'un des q_i soit égal à π, l'état puits introduit dans $\mathsf{eqcomplet}(A)$. Alors, puisque pour tout $s \in \Sigma$, $\pi \xrightarrow{s} \pi$ est l'unique transition issue de π et d'étiquette s, il vient que tous les états $q_{i+1}, \ldots, q_n$ sont eux aussi égaux à π. Or $\pi \notin F$, ce qui contredit $q_n \in F$. Nous en déduisons que tous les états de cette exécution sont distincts de π, et donc par la définition 1.14, qu'il s'agit également d'une exécution acceptante de A. ■

Nous considérons maintenant un automate fini $A = (Q, \Sigma, \delta, q_0, F)$ **déterministe et complet**. Nous montrons que le complémentaire de $\mathcal{L}(A)$ est lui aussi reconnu par un automate fini, et nous donnons une construction d'un automate qui le reconnaît.

Définition 1.15 (Complémentaire) Soit $A = (Q, \Sigma, \delta, q_0, F)$ un automate fini déterministe et complet. Le **complémentaire** de A est l'automate $\overline{A} = (Q, \Sigma, \delta, q_0, F')$ où :

— $F' = Q \setminus F$. ⧫

La complémentation d'un automate fini déterministe et complet consiste donc simplement à inverser les états accepteurs et non accepteurs de l'automate. La conséquence de cette inversion est que les mots acceptés par A sont rejetés par $\overline{A}$ et inversement, ceux rejetés par A sont acceptés par $\overline{A}$.

Exemple 1.13 Considérons l'automate fini déterministe et complet A_1 en figure 1.1(a). $\overline{A_1}$ est l'automate isomorphe tel que l'état q_0 est accepteur, et l'état q_1 est non accepteur. Visuellement, les mots acceptés par $\overline{A_1}$ sont donc le mot vide ε d'une part (car q_0 est accepteur) et tous les mots se terminant pas 1 d'autre part. $\overline{A}$ accepte donc les encodages binaires des entiers naturels impairs (plus ε). ⧫

Lemme 1.2 (Correction de la complémentation) Pour tout automate fini déterministe et complet $A = (Q, \Sigma, \delta, q_0, F)$, nous avons[3] $\mathcal{L}(\overline{A}) = \overline{\mathcal{L}(A)}$ ⧫

Preuve. Soit $w \in \mathcal{L}(\overline{A})$ avec $w = w_1 w_2 \ldots w_n$ avec $w_i \in \Sigma$ pour tout $i \in [1; n]$. Alors, il existe une exécution acceptante de $\overline{A}$:

$$q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} \cdots \xrightarrow{w_n} q_n \qquad \text{avec } q_n \in F'$$

3. $\overline{\mathcal{L}(A)} = \Sigma^* \setminus \mathcal{L}(A)$ est le complément de $\mathcal{L}(A)$ défini en section 3.1 page 29.

Par définition de $\overline{A}$, cette exécution est aussi une exécution de A, mais $q_n \notin F$, et A rejette w.

Soit maintenant $w \in \mathcal{L}(A)$ avec $w = w_1 w_2 \dots w_n$ avec $w_i \in \Sigma$ pour tout $i \in [1; n]$. Alors, il existe une exécution acceptante de A :

$$q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} \cdots \xrightarrow{w_n} q_n \qquad \text{avec } q_n \in F$$

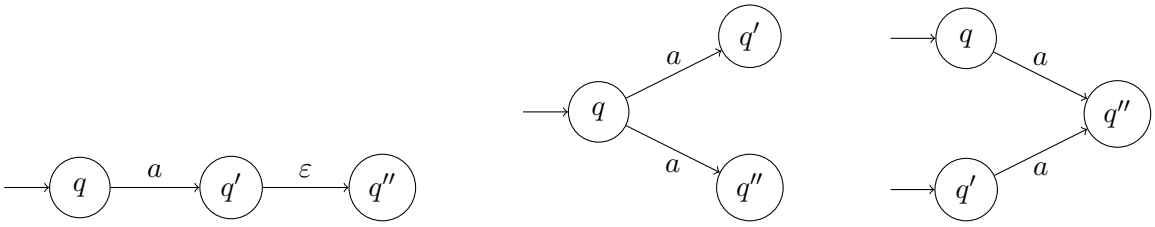
Par la définition 1.15, $q_n \notin F'$ donc $\overline{A}$ rejette w. ■

Chapitre 2

Non-déterminisme et déterminisation

Nous avons introduit au chapitre 1 une distinction entre automates finis **déterministes** et automates finis **non-déterministes** (définition 1.10 page 11). La notion de non-déterminisme provient de l'impossibilité de déterminer l'état courant de l'automate, à partir de son état initial et des symboles lus.

Exemple 2.1 Considérons les trois automates ci-dessous :



(a) A_N (b) A'_N (c) A''_N

FIGURE 2.1 – Trois cas de non-déterminisme.

Lorsque A_N, en figure 2.1(a), lit le symbole a à partir de l'état q, il peut en résulter deux exécutions :

$$q \xrightarrow{a} q' \qquad \text{et} \qquad q \xrightarrow{a} q' \xrightarrow{\varepsilon} q''$$

Pour un observateur qui ne connaîtrait que l'état de départ q, et le symbole lu a, il serait alors totalement impossible de déterminer si l'état d'arrivée de A_N est q' ou q''.

L'automate A'_N, en figure 2.1(b), est lui aussi non-déterministe puisqu'à partir de q, et en lisant a, il existe deux exécutions possibles :

$$q \xrightarrow{a} q' \qquad \text{et} \qquad q \xrightarrow{a} q''$$

Là encore, un observateur qui ne connaîtrait que l'état de départ q, et le symbole lu a, ne peut pas déterminer si le nouvel état de A'_N est q' ou q''.

Enfin, il n'est pas possible de déterminer l'état initial de A''_N : il peut s'agir de q ou de q'. Il existe deux exécutions possibles en lisant a :

$$q \xrightarrow{a} q'' \qquad \text{et} \qquad q' \xrightarrow{a} q''$$

C'est en cela que A_N, A'_N et A''_N sont non-déterministes : il est parfois impossible de déterminer l'état de l'automate durant une exécution. Il est cependant possible de déterminer un ensemble d'états parmi lesquels peut se trouver l'automate : $\{q', q''\}$ pour A_N et A'_N après lecture de a, et $\{q, q'\}$ pour A''_N avant lecture de a. Notons que les causes du non-déterminisme sont différentes et qu'elles peuvent bien sûr se combiner. ♦

Pour certains langages, nous pouvons contruire des automates non-déterministes et des automates déterministes qui les reconnaissent. Existe-t-il un langage qui est reconnu par un automate fini non-déterministe, mais par aucun automate fini déterministe ? Nous montrons en section 2.1 qu'il n'en est rien : tout langage reconnu par un AFN est également reconnu par un AFD. Puis en section 2.2 nous présentons un algorithme qui calcule, à partir d'un AFN A_N donné, un AFD A_D tel que $\mathcal{L}(A_D) = \mathcal{L}(A_N)$.

2.1 Élimination du non-déterminisme

Considérons l'automate fini non-déterministe A_N de la figure 2.2. A_N accepte en particulier

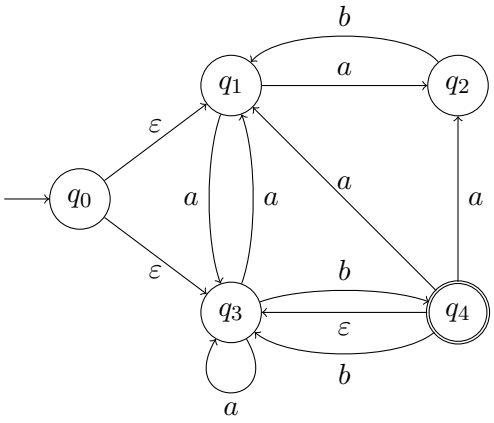


FIGURE 2.2 – L'automate non-déterministe A_N.

le mot *aabb*, notamment sur l'exécution suivante :

$$q_0 \xrightarrow{\varepsilon} q_3 \xrightarrow{a} q_3 \xrightarrow{a} q_3 \xrightarrow{b} q_4 \xrightarrow{\varepsilon} q_3 \xrightarrow{b} q_4$$

Nous remarquons que cette exécution contient deux transitions instantanées qui sont toutes les deux indispensables à A_N pour accepter *aabb* (sur cette exécution) puisque d'une part depuis q_0 il n'y a aucune transition d'étiquette a, et d'autre part sans la transition $q_4 \xrightarrow{\varepsilon} q_3$, l'exécution se terminerait en q_3 qui n'est pas accepteur. En toute généralité, il peut être nécessaire qu'un automate fini non-déterministe franchisse une ou plusieurs transitions instantanées avant et après avoir lu chaque symbole du mot d'entrée afin de l'accepter. La figure 2.3 présente l'arbre des exécutions de A_N sur le mot *aabb* où nous avons donné l'occasion à A_N de franchir autant de transitions instantanées que possible (les symboles de *aabb* sont séparés par des ε). Nous voyons en particulier que pour toutes les exécutions acceptantes de A_N sur *aabb*, il est nécessaire de commencer par une transition instantanée : soit $q_0 \xrightarrow{\varepsilon} q_1$ soit $q_0 \xrightarrow{\varepsilon} q_3$. Rappelons également que ε permet de boucler sur tout état.

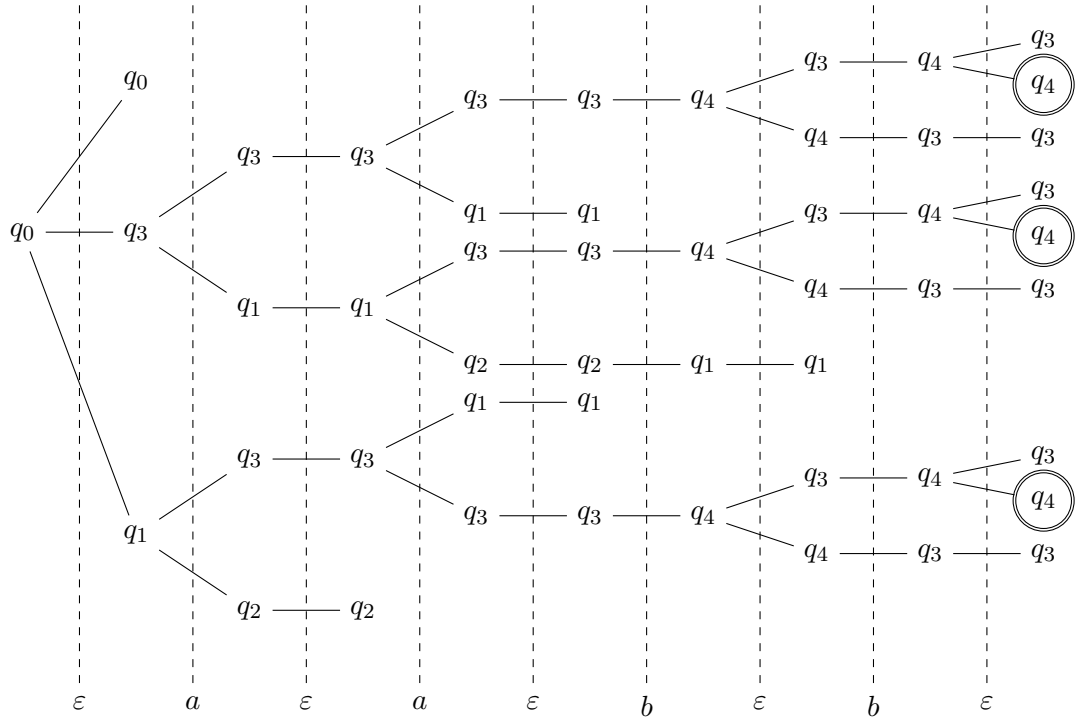


FIGURE 2.3 – Arbre des exécution de l'AFN A_N en figure 2.2 sur le mot *aabb*.

Nous construisons la séquence de transitions d'ensemble d'états en ensemble d'états de A_N en regroupant les états suivant leur distance à la racine q_0 dans l'arbre :

$$\{q_0\} \xrightarrow{\varepsilon} \{q_0, q_1, q_3\} \xrightarrow{a} \{q_1, q_2, q_3\} \xrightarrow{\varepsilon} \{q_1, q_2, q_3\} \xrightarrow{a} \{q_1, q_2, q_3\} \xrightarrow{\varepsilon} \\ \{q_1, q_2, q_3\} \xrightarrow{b} \{q_1, q_4\} \xrightarrow{\varepsilon} \{q_1, q_3, q_4\} \xrightarrow{b} \{q_3, q_4\} \xrightarrow{\varepsilon} \{q_3, q_4\} \quad (2.1)$$

Dans cette séquence, la transition $\{q_0\} \xrightarrow{\varepsilon} \{q_0, q_1, q_3\}$ signifie que sans avoir lu aucune lettre du mot *aabb*, A_N peut se déplacer de l'état q_0 à l'un des états q_0, q_1 ou q_3. Puis les deux transitions $\{q_0, q_1, q_3\} \xrightarrow{a} \{q_1, q_2, q_3\} \xrightarrow{\varepsilon} \{q_1, q_2, q_3\}$ indiquent que depuis un état parmi $\{q_0, q_1, q_3\}$, en lisant le symbole a, A_N atteint un état parmi $\{q_1, q_2, q_3\}$, puis A_N peut se déplacer suivant des transitions instantanées (si possible) pour se trouver dans un état parmi $\{q_1, q_2, q_3\}$. Les transitions de la séquence peuvent donc être regroupées de la façon suivante :

$$\{q_0\} \xrightarrow{\varepsilon} \{q_0, q_1, q_3\} \xrightarrow{a\cdot\varepsilon} \{q_1, q_2, q_3\} \xrightarrow{a\cdot\varepsilon} \{q_1, q_2, q_3\} \xrightarrow{b\cdot\varepsilon} \{q_1, q_3, q_4\} \xrightarrow{b\cdot\varepsilon} \{q_3, q_4\} \quad (2.2)$$

L'ensemble des exécutions d'un automate fini non-déterministe A sur un mot $w = w_1 \dots w_n$ peut donc être vue comme suit :

1. A débute dans l'ensemble $X_0 = \{q \in Q \mid \exists q_0 \in I,\ q_0 \xrightarrow{\varepsilon} q\}$ des états q pour lesquels il existe une séquence (éventuellement vide) de transitions ε d'un état initial à q ;
2. puis A lit le premier symbole w_1 et atteint l'ensemble $X_1 = \{q' \in Q \mid \exists q \in X_0,\ q \xrightarrow{w_1\cdot\varepsilon} q'\}$ des états q' pour lesquels il existe une séquence de transitions issue d'un état $q \in X_0$, débutant par w_1 et suivie d'une séquence (éventuellement vide) de transitions ε ;

3. puis A lit le second symbole w_2 et atteint l'ensemble d'états $X_2 = \{q' \in Q \mid \exists q \in X_1,\ q \xrightarrow{w_2 \cdot \varepsilon} q'\}$;
4. et ainsi de suite.

Nous pouvons *in fine* éliminer les ε pour obtenir la séquence de transitions qui suit :

$$\underbrace{\{q_0, q_1, q_3\}}_{X_0} \xrightarrow{a} \underbrace{\{q_1, q_2, q_3\}}_{X_1} \xrightarrow{a} \underbrace{\{q_1, q_2, q_3\}}_{X_2} \xrightarrow{b} \underbrace{\{q_1, q_3, q_4\}}_{X_3} \xrightarrow{b} \underbrace{\{q_3, q_4\}}_{X_4} \tag{2.3}$$

La figure 2.4 (page 22) présente un automate fini déterministe A_D tel que $\mathcal{L}(A_D) = \mathcal{L}(A_N)$. La séquence (2.3) correspond précisément à l'exécution de A_D sur le mot d'entrée *aabb*. Nous présentons maintenant, une construction de déterminisation des automates finis non-déterministes qui généralise le principe exposé ci-dessus. Celle-ci requiert donc deux phases :

1. l'élimination des transitions instantanées (ou clôture instantanée) ;
2. et l'élimination des choix non déterministes.

2.1.1 Clôture instantanée

Reprenons la séquence de transitions de l'équation (2.1). La première transition $\{q_0\} \xrightarrow{\varepsilon} \{q_0, q_1, q_3\}$ définit l'ensemble des états accessibles depuis $\{q_0\}$ en ne suivant que des transitions instantanées. Cet ensemble s'appelle la **clôture instantanée** de l'ensemble $\{q_0\}$.

Définition 2.1 (Clôture instantanée) Soit un automate fini $A = (Q, \Sigma, \delta, I, F)$, et q un état de A. La **clôture instantanée** de q, notée $\mathrm{cl}_\varepsilon(q)$ est l'ensemble des états de A accessibles depuis q par une séquence (éventuellement vide) de transitions instantanées :

$$\mathrm{cl}_\varepsilon(q) = \{q' \in Q \mid q \xrightarrow{\varepsilon} q'\} \tag{2.4}$$

♦

▷ Pour tout état $q \in Q$, $q \in \mathrm{cl}_\varepsilon(q)$.

La clôture instantanée se généralise naturellement à un ensemble d'états $X \subseteq Q$:

$$\mathrm{cl}_\varepsilon(X) = \bigcup_{q \in X} \mathrm{cl}_\varepsilon(q) \tag{2.5}$$

Exemple 2.2 Pour A_N en figure 2.2, la clôture instantanée de q_4 est $\{q_4, q_3\}$, la clôture instantanée de q_0 est $\{q_0, q_1, q_3\}$ et la clôture instantanée de q_1 est $\{q_1\}$. Sur l'automate fini de l'exemple 3.6 (page 37), $\mathrm{cl}_\varepsilon(q_0) = \{q_0, q_1, q_2, q_3, q_4, q_5, q_{10}, q_{11}\}$. ♦

Nous verrons en section 2.1.3 comment exploiter $\mathrm{cl}_\varepsilon(.)$ pour obtenir un AFD qui reconnaît le même langage qu'un AFN donné.

2.1.2 Successeurs

Dans l'équation (2.1), la seconde transition $\{q_0, q_1, q_3\} \xrightarrow{a} \{q_1, q_2, q_3\}$ définit $\{q_1, q_2, q_3\}$ comme l'ensemble des états qui sont accessibles par depuis un état parmi $\{q_0, q_1, q_3\}$ en lisant le symbole a. Nous appelons $\{q_1, q_2, q_3\}$ les **successeurs** de $\{q_0, q_1, q_3\}$ par a.

Définition 2.2 (Successeurs d'un état) Soit un automate fini $A = (Q, \Sigma, \delta, I, F)$, q un état de A et s un symbole de Σ. L'ensemble des **successeurs** de q par s, notés $\text{succ}(q, s)$, est l'ensemble des états de A accessibles depuis q par une transition d'étiquette s :

$$\text{succ}(q, s) = \{q' \in Q \mid (q, s, q') \in \delta\} \tag{2.6}$$

♦

Les successeurs d'un ensemble d'états $X \subseteq Q$ sont obtenus par généralisation de la définition 2.2 :

$$\text{succ}(X, s) = \bigcup_{q \in X} \text{succ}(q, s) \tag{2.7}$$

Exemple 2.3 Pour A_N en figure 2.2, $\text{succ}(q_1, a) = \{q_2, q3\}$, $\text{succ}(q_1, b) = \emptyset$ et $\text{succ}(q_4, b) = \{q_3\}$. Sur l'automate fini de l'exemple 3.6 (page 37), $\text{succ}(\{q_4, q_5, q_{11}\}, 0) = \{q_6, q_{12}\}$ ♦

2.1.3 Automate déterministe équivalent

Nous utilisons maintenant $\text{cl}_\varepsilon(.)$ et succ(.) pour définir, à partir d'un AFN A_N donné, un AFD A_D tel que $\mathcal{L}(A_N) = \mathcal{L}(A_D)$. Reprenons l'équation (2.3). Nous avons $X_0 = \text{cl}_\varepsilon(I)$, puis $X_1 = \text{cl}_\varepsilon(\text{succ}(X_0, a))$, puis $X_2 = \text{cl}_\varepsilon(\text{succ}(X_1, a))$, puis $X_3 = \text{cl}_\varepsilon(\text{succ}(X_2, b))$ et enfin $X_4 = \text{cl}_\varepsilon(\text{succ}(X_3, b))$. Les états de l'automate A_D sont donc associés aux ensembles d'états X_0, X_1, X_2, etc., et y a une transition de X_i à X_j de symbole s lorsque $X_j = \text{cl}_\varepsilon(\text{succ}(X_i, s))$. L'unique état initial correspond à X_0, et tous les X_i tels que $X_i \cap F \neq \emptyset$ sont accepteurs. En effet, si l'exécution de A_D sur w aboutit dans X_i, alors pour tout $q \in X_i$ il existe une exécution $q_0 \xrightarrow{w} q$ dans A_N. Notons que les ensembles X_0, X_1, X_2, X_3 et X_4 de l'équation (2.3) sont calculés pour un mot d'entrée particulier. Afin de calculer un automate déterministe équivalent à A_N indépendamment de tout mot d'entrée, nous considérons l'ensemble des X_i possibles.

Définition 2.3 (Automate déterministe équivalent) Soit $A = (Q, \Sigma, \delta, I, F)$ un automate fini (non-déterministe). L'**automate déterministe équivalent** à A est défini comme $\mathsf{eqdet}(A) = (Q', \Sigma, \delta', q'_0, F')$ où :

- $Q' = 2^Q$ les parties de Q,
- $q'_0 = \text{cl}_\varepsilon(I)$,
- pour tout $X \in Q'$ et $s \in \Sigma$, $(X, s, X') \in \delta'$ ssi $X' = \text{cl}_\varepsilon(\text{succ}(X, s))$
- $F' = \{X \in Q' \mid X \cap F \neq \emptyset\}$

♦

La figure 2.4 représente un automate fini déterministe A_D qui reconnaît le même langage que l'automate fini non-déterministe A_N de la figure 2.2. Il s'agit de $\mathsf{eqdet}(A_N)$ **restreint à ses états accessibles**. Les états de A_D sont étiquetés par des ensembles d'états de A_N afin de faire la correspondance entre les deux automates. Pour chaque ensemble pertinent d'états de A_N, nous créons donc un état dans A_D. Nous montrons maintenant que $\mathsf{eqdet}(A)$ possède trois propriétés remarquables. Tout d'abord il est déterministe et complet. Ensuite, il est équivalent à A : $\mathcal{L}(A) = \mathcal{L}(\mathsf{eqdet}(A))$.

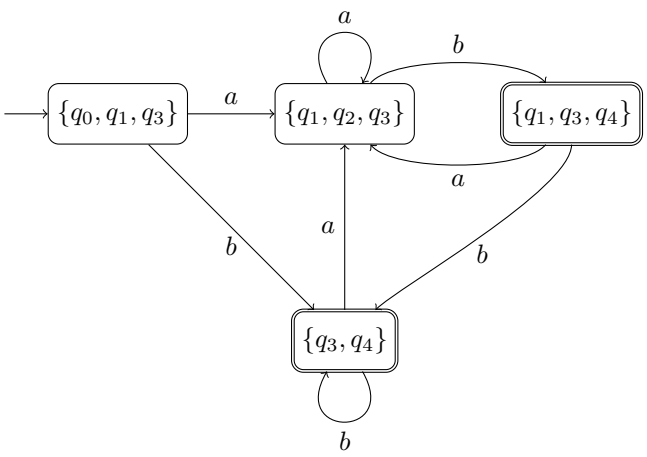


FIGURE 2.4 – Un automate fini déterministe A_D qui reconnaît le même langage que A_N (les états inaccessibles de A_D ne sont pas représentés).

Lemme 2.1 Quel que soit A automate fini, $\mathsf{eqdet}(A)$ est déterministe et complet. ♦

Preuve. D'après la définition 2.3, δ' est une fonction totale. En effet, pour chaque couple $(X, s) \in Q' \times \Sigma$, il existe un unique ensemble X' tel que $(X, s, X') \in \delta$. Il vient donc à la fois le déterminisme (au plus un triplet $(X, s, X') \in \delta'$ quels que soient X et s) et la complétude (au moins un triplet $(X, s, X') \in \delta'$ quels que soient X et s). ■

Lemme 2.2 Quel que soit A automate fini, $\mathcal{L}(\mathsf{eqdet}(A)) = \mathcal{L}(A)$. ♦

Preuve. Soit $w \in \mathcal{L}(\mathsf{eqdet}(A))$. Alors, il existe une exécution :

$$q'_0 \xrightarrow{a_0} q'_1 \xrightarrow{a_1} \cdots \xrightarrow{a_n} q'_{n+1}$$

telle que $w = a_0 a_1 \dots a_n$ et $q'_{n+1} \in F'$. Nous montrons alors par induction sur la longueur de cette exécution que q'_i est l'ensemble des états de A accessibles depuis un état initial de A après avoir lu i lettres de w.

Cas de base : $q'_0 = \mathrm{cl}_\varepsilon(I)$ est bien l'ensemble des états q_0^ε tels qu'il existe un état initial $q_0 \in I$ tel que $q_0 \xrightarrow{\varepsilon} q_0^\varepsilon$. Donc q'_0 est l'ensemble des états accessibles dans A après avoir lu 0 lettres de w

Supposons maintenant que q'_k est l'ensemble des états q_k pour lesquels il existe $q_0 \in I$ tel que $q_0 \xrightarrow{a_0 \dots a_{k-1}} q_k$. Puisque $q'_k \xrightarrow{a_k} q'_{k+1}$, par la définition 2.3 : $q'_{k+1} = \mathrm{cl}_\varepsilon\left(\mathrm{succ}(q'_k, a_k)\right)$.

Donc par l'équation (2.5), et par hypothèse d'induction, q'_k étant l'ensemble des états de A accessibles depuis q_0 après avoir lu $a_0 \dots a_{k-1}$:

$$\begin{aligned} q'_{k+1} &= \left\{ q_{k+1} \in Q \,|\, \exists q_k \in q'_k.\, \exists p \in Q.\; q_k \xrightarrow{a_k} p \xrightarrow{\varepsilon} q_{k+1} \right\} \\ &= \left\{ q_{k+1} \in Q \,|\, \exists q_0 \in I.\, \exists q_k \in q'_k.\, \exists p \in Q.\; q_0 \xrightarrow{a_0 \cdots a_{k-1}} q_k \xrightarrow{a_k} p \xrightarrow{\varepsilon} q_{k+1} \right\} \\ &= \left\{ q_{k+1} \in Q \,|\, \exists q_0 \in I.\, q_0 \xrightarrow{a_0 \cdots a_k} q_{k+1} \right\} \end{aligned}$$

Il vient que q'_{n+1} est l'ensemble des états accessibles depuis un état initial de A après avoir lu $a_0 \dots a_n = w$. Sachant que $q'_{n+1} \in F'$, nous avons donc qu'il existe $q_{n+1} \in F$ tel que $q_0 \xrightarrow{w} q_{n+1}$ dans A avec $q_0 \in I$. Par conséquent, $w \in \mathcal{L}(A)$.

La preuve de l'inclusion inverse, $\mathcal{L}(\mathsf{eqdet}(A)) \supseteq \mathcal{L}(A)$ est laissée au lecteur. ■

Nous déduisons des deux résultats précédents que le non-déterminisme ne permet pas de reconnaître plus de langages que le déterminisme ne le permet.

Théorème 2.1 (Equivalence AFD/AFN) Tout langage reconnu par un AFN est reconnu par un AFD. ♦

Nous avons donc prouvé que le non-déterminisme n'augmente pas **l'expressivité** des automates finis. Ainsi tout langage régulier admet une procédure effective, c'est à dire mécaniquement réalisable, qui le décide. Alors, à quoi sert le non déterminisme ? Il est parfois plus simple d'utiliser des automates non déterministes pour trouver la solution à un problème. C'est notamment le cas des langages L_n décrits dans l'exemple 2.4. Bien que cette solution ne soit pas une procédure effective telle quelle, nous pouvons la rendre déterministe ensuite. Notons cependant que l'opération de déterminisation a un coût non négligeable. Le non-déterministe est également utile pour certaines constructions (voir section 3.3.2 page 34).

Lemme 2.3 (Complexité de l'automate déterministe équivalent) Soit un automate fini $A = (Q, \Sigma, \delta, q_0, F)$ et $\mathsf{eqdet}(A) = (Q', \Sigma, \delta', q'_0, F')$. Nous avons $\mathsf{Card}(Q') = 2^{\mathsf{Card}(Q)}$. ♦

Preuve. Directement par la définition de Q' (voir définition 2.3). ■

Donc, par la définition 2.3 nous obtenons un AFD exponentiellement plus gros que l'AFN de départ. Assez souvent, de nombreux états de $\mathsf{eqdet}(A)$ ne sont pas accessibles depuis son état initial. C'est par exemple le cas pour l'automate A_N en figure 2.2 puisque $\mathsf{eqdet}(A_N)$ dont la partie accessible représentée en figure 2.4 ne comporte que 4 états au lieu de $2^5 = 32$ états. Ces états inaccessibles peuvent donc être supprimés sans que cela ne modifie le langage reconnu par $\mathsf{eqdet}(A)$ puisqu'aucune exécution de l'AFD ne passe par ces états. Nous voyons dans la section suivante un algorithme qui évite directement de construire ces états inutiles. Cependant il existe aussi des AFN pour lesquels le plus petit AFD équivalent compte exponentiellement plus d'états.

Exemple 2.4 Sur l'alphabet $\{a, b\}$, soit $L_n = \{a, b\}^* \cdot a \cdot \{a, b\}^{n-1}$ la famille des langages dont les mots ont un a en position n à partir de la fin. L'automate en figure 2.5 reconnaît le langage

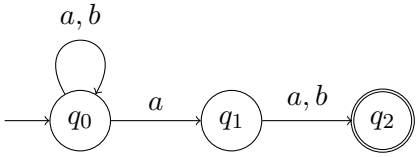


FIGURE 2.5 – Un AFN complexe à déterminiser.

L_2. Il s'avère que pour tout $n \geq 1$, le plus petit AFD qui reconnaît L_n est exponentiellement plus gros (en nombre d'états), que le plus petit AFN correspondant (comme celui en figure 2.5 par exemple). ♦

2.2 Algorithme de déterminisation

La définition 2.3 montre qu'il faut deux opérations pour le calcul de $\mathsf{eqdet}(A)$:
— le calcul de la clôture instantanée $\mathrm{cl}_\varepsilon(.)$,
— et le calcul des successeurs succ(.).
Le calcul de succ(.) est trivial depuis la relation de transition δ de A. Nous donnons donc un algorithme pour le calcul de $\mathrm{cl}_\varepsilon(.)$ en section 2.2.1, puis directement un algorithme pour le calcul de la partie accessible de $\mathsf{eqdet}(A)$ en section 2.2.2.

2.2.1 Calcul de la clôture instantanée

Le calcul de $\mathrm{cl}_\varepsilon(q)$ est réalisé de façon itérative : si $(q, \varepsilon, q') \in \delta$, alors $\mathrm{cl}_\varepsilon(q') \subseteq \mathrm{cl}_\varepsilon(q)$. On peut donc calculer les états accessibles depuis chaque état de A par une unique transition instantanée, puis propager cette information le long des transitions instantanées, et ce, jusqu'à obtention d'un point fixe. L'algorithme ClotureInstantanée de la figure 2.6 réalise ce calcul.

```
1  ENTREE: Un automate fini A = (Q, Σ, δ, I, F)
2  SORTIE: cl_ε, la clôture instantanée de Q
3
4  ClotureInstantanée(A) :
5  debut
6    pour tout q ∈ Q faire
7      cl_ε(q) ← {q}
8      cl'_ε(q) ← ∅
9    finpour
10
11   tantque cl_ε ≠ cl'_ε faire
12     cl'_ε ← cl_ε
13     pour tout q ∈ Q faire
14       cl_ε(q) ← cl'_ε(q) ∪ ( ⋃_{(q,ε,q')∈δ} cl'_ε(q') )
15     finpour
16   fintantque
17
18   retourner cl_ε
19 fin
```

FIGURE 2.6 – Algorithme de calcul de la clôture instantanée.

Exemple 2.5 Reprenons l'automate fini non déterministe A_N de la figure 2.2. Nous montrons dans le tableau suivant les étapes de calcul de la clôture instantanée pour A_N.

étape	q_0	q_1	q_2	q_3	q_4
0	$\{q_0\}$	$\{q_1\}$	$\{q_2\}$	$\{q_3\}$	$\{q_4\}$
1	$\{q_0, q_1, q_3\}$	$\{q_1\}$	$\{q_2\}$	$\{q_3\}$	$\{q_3, q_4\}$
2	$\{q_0, q_1, q_3\}$	$\{q_1\}$	$\{q_2\}$	$\{q_3\}$	$\{q_3, q_4\}$

L'étape 0 correspond aux lignes 5 à 8 de l'algorithme, alors que les étapes suivantes résultent de l'itération de la boucle «tantque» en lignes 10-15. L'algorithme converge après 2 étapes de propagation de la clôture instantanée. La fonction calculée renvoie bien les ensembles mentionnés à l'exemple 2.2. ♦

Théorème 2.2 Pour tout automate $A = (Q, \Sigma, \delta, I, F)$ l'algorithme 2.6 ClotureInstantanée calcule cl_ε en $\mathcal{O}(\mathsf{Card}(Q)^2)$. ♦

Preuve. L'algorithme 2.6 termine. En effet, la suite des valeurs de cl_ε est strictement croissante : pour tout $q \in Q$ la valeur de $\mathrm{cl}_\varepsilon(q)$ croît ou reste constante en ligne 14, et elle n'est modifiée nulle part ailleurs dans cette boucle. Par ailleurs si $\mathrm{cl}_\varepsilon \neq \mathrm{cl}'_\varepsilon$, alors il existe $q \in Q$ tel que $\mathrm{cl}'_\varepsilon(q) \subset \mathrm{cl}_\varepsilon(q)$. Nous avons alors une suite strictement croissante et bornée : pour tout $q \in Q$, $\mathrm{cl}_\varepsilon(q) \subseteq Q$.

Nous prouvons que l'algorithme est correct par l'invariance de «$\mathrm{cl}_\varepsilon(q)$ contient l'ensemble des états cibles d'une transition instantanée issue d'un état de $\mathrm{cl}'_\varepsilon(q) \cup \{q\}$». L'invariant est satisfait avant l'entrée dans la boucle, immédiatement après les initialisations en ligne 10. Supposons qu'il soit vrai en début de boucle. Après la ligne 12, cl'_ε contient la valeur de cl_ε à l'itération précédente qui satisfait l'invariant de boucle par hypothèse. Alors, en ligne 14, nous ajoutons à cl_ε tous les états cibles d'une transition instantanée issue de cl'_ε. Nous en déduisons donc cl_ε contient en plus de cl'_ε, tous les états cibles d'une transitions instantanée issue de cl'_ε. En sortie de boucle, $\mathrm{cl}_\varepsilon = \mathrm{cl}'_\varepsilon$. Par l'invariant ci-dessus, cl_ε contient tous les états cibles d'une transitions instantanée issue d'un état de cl_ε. Donc cl_ε est bien la clôture instantanée des états de A.

Enfin, si $q' \in \mathrm{cl}_\varepsilon(q)$ pour $q \in Q$, alors la plus longue séquence acyclique de transitions instantanées de q à q' est de longueur au plus $\mathsf{Card}(Q) - 1$. Nous en déduisons qu'il faut au plus $\mathsf{Card}(Q)-1$ itérations de la boucle **tantque** pour calculer cl_ε (*i.e.* pour que q' soit ajouté à $\mathrm{cl}_\varepsilon(q)$). Donc, il faut $\mathcal{O}(\mathsf{Card}(Q)^2)$ opérations pour calculer cl_ε pour chacun des $\mathsf{Card}(Q)$ états en supposant que les manipulations ensemblistes (union, différence) sont en temps constant. ■

2.2.2 Calcul de l'automate déterministe équivalent

Le principe de l'algorithme de déterminisation est de construire l'automate $\mathsf{eqdet}(A)$ conformément à la définition 2.3, en ne calculant que la **partie accessible** de celui-ci afin de limiter, lorsque c'est possible, l'explosion combinatoire présentée au lemme 2.3.

L'algorithme fonctionne par construction itérative de l'ensemble des états Q' de $\mathsf{eqdet}(A)$. À partir de l'état initial $q'_0 = \mathrm{cl}_\varepsilon(I)$ de $\mathsf{eqdet}(A)$, nous calculons les ensembles $S_1, \ldots, S_k$ des états accessibles depuis les états de q'_0 en considérant chaque symbole de $\Sigma = \{s_1, \ldots, s_k\}$. Ainsi, S_1 est l'ensemble des états de $\mathsf{eqdet}(A)$ accessibles depuis q'_0 par une transition de symbole s_1, suivie d'un nombre arbitraire de transitions instantanées, et de même pour $S_2, \ldots, S_k$. Les ensembles d'états parmi $S_1, \ldots, S_k$ différents de q'_0 donnent naissance à des états de $\mathsf{eqdet}(A)$. Puis le processus appliqué précédemment à q'_0 est appliqué à chacun de ces nouveaux états, et ainsi de suite jusqu'à convergence. La terminaison est garantie puisqu'il y a un nombre fini de parties de Q, et que chacune d'elle est traitée une seule fois. La figure 2.7 présente l'algorithme Déterminisation qui réalise les opérations mentionnées ci-dessus.

Exemple 2.6 Nous détaillons le fonctionnement de l'algorithme de déterminisation sur l'exemple de la figure 2.2. Le tableau suivant retrace les créations d'états de A_D durant le fonctionnement de l'algorithme.

```
1  ENTREE: A = (Q, Σ, δ, I, F) non–déterministe
2  SORTIE: A' = (Q', Σ, δ', q'_0, F') déterministe tel que L(A) = L(A')
3
4  Déterminisation(A) :
5  debut
6     Q' ← ∅;   F' ← ∅;   δ' ← ∅;
7     q'_0 ← cl_ε(I);
8     N ← {q'_0};
9     tantque (N ≠ ∅) faire
10       P ← elmt(N);   N ← N \ {P};
11       Q' ← Q' ∪ {P};
12       si (P ∩ F ≠ ∅) alors
13          F' ← F' ∪ {P}
14       finsi
15       pour tout s ∈ Σ faire
16          P' ← cl_ε (succ(P, s))
17          δ' ← δ' ∪ {(P, s, P')}
18          si (P' ∉ Q') alors
19             N ← N ∪ {P'}
20          finsi
21       finpour
22    fintantque
23
24    retourner A'
25 fin
```

FIGURE 2.7 – Algorithme de déterminisation.

Q'	a	b
$\{q_0, q_1, q_3\}$	$\{q_1, q_2, q_3\}$	$\{q_3, q_4\}$
$\{q_1, q_2, q_3\}$	$\{q_1, q_2, q_3\}$	$\{q_1, q_3, q_4\}$
$\{q_3, q_4\}$	$\{q_1, q_2, q_3\}$	$\{q_3, q_4\}$
$\{q_1, q_3, q_4\}$	$\{q_1, q_2, q_3\}$	$\{q_3, q_4\}$

Notons tout d'abord que l'état initial de A_D correspond à l'ensemble $\{q_0, q_1, q_3\}$, la clôture instantanée de q_0. Ensuite, depuis $\{q_0, q_1, q_3\}$, l'ensemble des états accessibles dans A_N en lisant un symbole a est $\{q_1, q_2, q_3\}$ qui est invariant par cl_ε. En lisant un symbole b, A_N atteint $\{q_3\}$ dont la clôture instantanée est $\{q_3, q_4\}$ (les clôtures instantanées pour A_N sont données dans l'exemple 2.5). Les ensembles $\{q_1, q_2, q_3\}$ et $\{q_3, q_4\}$, qui n'ont pas encore été traités ($Q' = \{\{q_0, q_1, q_3\}\}$), sont ajoutés à N et les états correspondant sont créés dans Q'.

Nous continuons alors le même processus à partir de $\{q_1, q_2, q_3\}$ qui est retiré de N. Par lecture de la lettre a, A_N atteint $\{q_1, q_2, q_3\}$. Celui-ci ayant déjà été traité (il appartient à Q'), il n'est pas ajouté à N. Sur lecture du symbole b, A_N atteint $\{q_1, q_4\}$ dont la clôture instantanée est $\{q_1, q_3, q_4\}$. Nous l'ajoutons alors à N et Q' puisqu'il n'a pas été traité.

L'algorithme se poursuit ainsi jusqu'à ce que plus aucun nouvel ensemble d'états ne soit créé. Alors, les colonnes a et b du tableau précédent donnent la relation de transition de A_D

qui est représenté en figure 2.4. ♦

Il reste à prouver la terminaison et la correction de l'algorithme de déterminisation.

Théorème 2.3 Pour tout automate (non-déterministe) $A = (Q, \Sigma, \delta, I, F)$ en entrée, l'algorithme 2.7 Déterminisation calcule un automate fini déterministe $A' = (Q', \Sigma, \delta', Q'_0, F')$ tel que $\mathcal{L}(A') = \mathcal{L}(A)$ en temps $\mathcal{O}(\mathsf{Card}(\Sigma).2^{\mathsf{Card}(Q)})$. ♦

Preuve. L'algorithme 2.7 termine. En effet, $Q' \subseteq 2^Q$ est invariant. Nous en déduisons d'une part que Q' est un ensemble fini puisque Q est fini. D'autre part, la suite des valeurs de Q' est strictement croissante puisqu'un élément est ajouté à N uniquement lorsqu'il n'appartient pas déjà à Q' et il est ajouté à Q' lorsqu'il est ensuite extrait de N (et c'est le seul endroit où Q' est modifié). Donc $2^Q \setminus Q'$ est une suite strictement décroissante dans un ensemble bien fondé.

La correction de l'algorithme 2.7 se montre en prouvant l'invariance de :

— «$(Q' \cup N, \Sigma, \delta', q'_0, F')$ est le préfixe de A' restreint à Q'». En d'autres termes : A' a été calculé jusqu'aux états de Q'. Pour P donné choisi en ligne 10, une transition est ajoutée en ligne 17 pour chaque symbole de Σ (boucle en ligne 15). De plus, chaque élément de Q' est traité une et une seule fois : il est ajouté à N en ligne 19 lorsqu'il n'est pas déjà dans Q' (test en ligne 18). Donc, en ligne 17, un triplet est ajouté à δ' exactement une fois pour P donné et s donné (du fait de la boucle en ligne 15 qui passe en revue chaque symbole de Σ).

Par ailleurs, si P est accepteur (conformément à la définition 2.3), il est ajouté à F' en ligne 13. Donc si nous avons un préfixe de A' en ligne 9, nous avons à nouveau un préfixe de A' en ligne 22.

— «$Q' \cup N$ est l'ensemble des états cibles d'une transition depuis un état de Q'». En ligne 16, P' est cible d'une transition depuis P. En ligne 18 soit $P' \in Q'$, soit il est ajouté à N en ligne 19. Sachant que P appartient à Q' (ligne 11), nous en déduisons que l'invariant est préservé.

Remarquons que ces deux invariants sont vrais à l'entrée de la boucle, après les initialisations jusqu'à la ligne 8. Puisque que $N = \emptyset$ en sortie de boucle, nous en déduisons que A' est la partie accessible de $\mathsf{eqdet}(A)$. En effet, par les deux invariants, A' est un préfixe de $\mathsf{eqdet}(A)$ qui contient tous les états accessibles. De plus, le premier invariant montre que q'_0, δ' et F' sont calculés conformément à la définition 2.3. Nous avons alors que $\mathcal{L}(A') = \mathcal{L}(\mathsf{eqdet}(A))$.

Enfin, nous avons montré au lemme 2.3 que $\mathsf{eqdet}(A)$ a au pire $2^{\mathsf{Card}(Q)}$ états. Nous avons vu précédemment que chaque état de A' apparaît exactement une fois dans N, nous en déduisons qu'il y a donc au pire $2^{\mathsf{Card}(Q)}$ itérations de la boucle **tantque**. En considérant que les opérations sur les ensembles (union, intersection) sont en temps constant, nous obtenons $\mathcal{O}(\mathsf{Card}(\Sigma).2^{\mathsf{Card}(Q)})$ opérations puisque pour chaque état de Q' nous considérons chacun des symboles de Σ. ■

Chapitre 3

Expressions régulières et théorème de Kleene

Nous avons vu au chapitre précédent que nous pouvons formaliser un problème de décision par la théorie des langages. C'est à dire qu'à toute question de décision $x \in E_{oui}$, nous pouvons associer une question $w \in L$ où w représente x et L représente E_{oui}. Nous avons également introduit les automates finis comme modèle d'algorithme pour le problème «$w \in L$?». Nous pouvons donc représenter x par un mot w, l'algorithme par un automate fini, mais comment représenter L ? S'il est fini, il suffit d'énumérer tous les mots qui le composent, mais que faire quand il est infini (cas de l'ensemble des entiers pairs de l'exemple 1.1 page 7) ? En section 3.1, nous identifions une classe de langages nommée **langages réguliers**, et nous montrons en section 3.3.1 que les langages réguliers admettent une représentation algébrique : les **expressions régulières** (définies en section 3.2). Puis en section 3.3.2 nous montrons que les langages décrits par les expressions régulières sont reconnus par les automates finis. Cette preuve montre notamment que les opérations ensemblistes sur les langages peuvent être transcrites directement sur les automates finis. Par exemple, il est possible de construire systématiquement l'automate fini reconnaissant le langage $L_1 \cup L_2$ à partir des automates finis reconnaissants L_1 et L_2 (L_1 et L_2 réguliers, bien entendu !). Enfin, nous montrons en section 3.3.3 que les langages reconnus par les automates finis sont représentés par des expressions régulières, et donc réguliers. Les résultats des sections 3.3.2 et 3.3.3 constituent le théorème de Kleene [1]. Ils sont **effectifs**, c'est à dire que pour une expression régulière donnée, il est possible de calculer un automate fini qui reconnaît le même langage et inversement.

3.1 Langages réguliers

Nous introduisons tout d'abord les opérateurs ensemblistes nécessaires à la définition des langages réguliers. L'union, l'intersection et la complémentation sont bien évidemment définies comme pour tout autre ensemble : pour tous langages L_1 et L_2

— $L_1 \cup L_2 = \{w \,|\, w \in L_1 \text{ ou } w \in L_2\}$ est **l'union** de L_1 et L_2.

— $L_1 \cap L_2 = \{w \,|\, w \in L_1 \text{ et } w \in L_2\}$ est **l'intersection** de L_1 et L_2.

1. Stephen Cole Kleene (1909-1994), notamment fondateur de la théorie de la récursion avec A. Church, K. Gödel et A. Turing.

— $\overline{L_1} = \{w \,|\, w \in \Sigma^* \text{ et } w \notin L_1\}$ est le **complément** de L_1.

▷ Nous avons vu au chapitre précédent (section 1.2.2 page 14) que pour un langage L reconnu par un automate fini A, il est possible de calculer un automate fini $\overline{A}$ qui reconnaît $\overline{L}$. Nous verrons en section 3.3.2 qu'à partir de deux automates A_1 et A_2 qui reconnaissent respectivement L_1 et L_2 il est possible de calculer un automate qui reconnaît $L_1 \cup L_2$. Il suffit alors d'utiliser successivement ces deux constructions pour calculer un automate qui reconnaît $L_1 \cap L_2$ puisque $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.

Les opérations ensemblistes sur les langages se traduisent intuitivement sur les problèmes que ces langages formalisent.

Exemple 3.1 Si L_1 encode l'ensemble des nombres pairs en base 2 (*i.e.* sur l'alphabet $\{0, 1\}$), et L_2 représente l'ensemble des multiples de 3 exprimés sur le même alphabet, alors $L_1 \cap L_2$ contient les encodages en base 2 de l'ensemble des nombres pairs et multiples de 3. ♦

Nous définissons ensuite deux autres opérations ensemblistes sur les langages qui leur sont spécifiques. Les langages sont des ensembles de mots qui sont eux-mêmes des séquences de lettres d'un alphabet donné. Nous avons vu en définition 1.5 (page 8) que les mots sont construits par concaténation de symboles pour former ainsi des séquences. Cette construction séquentielle est maintenant étendue aux langages. La **concaténation** de deux langages L_1 et L_2 produit l'ensemble des mots dont la première partie est un mot de L_1 et la seconde partie est un mot de L_2.

Définition 3.1 (Concaténation) La **concaténation** d'un langage L_2 à un langage L_1 est le langage :

$$L_1 \cdot L_2 = \{w_1 \cdot w_2 \,|\, w_1 \in L_1 \text{ et } w_2 \in L_2\}$$ ♦

La **fermeture de Kleene** de L définit le langage des mots qui sont des concaténations arbitrairement longues (*i.e.* non bornées) de mots de L.

Définition 3.2 (Fermeture de Kleene) La **fermeture de Kleene** d'un langage L est le langage :

$$L_1{}^* = \bigcup_{i \geq 0} L_1^i$$

où $\begin{cases} \text{— } L_1^0 = \{\varepsilon\} \text{ par convention ;} \\ \text{— et } L_1^n = L_1 \cdot L_1^{n-1}, \text{ lorsque } n \geq 1. \end{cases}$ ♦

Exemple 3.2 Soit $L_1 = \{\varepsilon, a, ab\}$ et $L_2 = \{b, ba\}$ deux langages.
— $L_1 \cdot L_2 = \{b, ba, ab, aba, abb, abba\}$
— $L_1{}^* = \{\varepsilon, a, ab, aa, aab, abab, aaa, aaab, aaba, aabab, abaa, abaab, ababa, ababab, \dots\}$
— $L_2{}^* = \{\varepsilon, b, ba, bb, bba, bab, baba, bbb, bbba, bbab, bbaba, babb, babba, babab, bababa, \dots\}$
— $L_1 \cdot L_2{}^* = L_2{}^* \cup \{a\} \cdot L_2{}^* \cup \{ab\} \cdot L_2{}^*$
— $L_1{}^* \cdot L_2 = L_1{}^* \cdot \{b\} \cup L_1{}^* \cdot \{ba\}$
— $(L_1 \cdot L_2)^* = \{\varepsilon\} \cup (L_1 \cdot L_2) \cup (L_1 \cdot L_2 \cdot L_1 \cdot L_2) \cup \dots$ ♦

▷ Nous voyons maintenant que la notation Σ^* pour représenter l'ensemble des mots sur Σ correspond en fait à la fermeture de Kleene. Cette notation est un peu abusive puisqu'on applique la fermeture à un alphabet et non pas à un langage. Il suffit de voir cet alphabet comme un ensemble de mots à une lettre pour que tout rentre dans l'ordre.

Nous définissons maintenant la classe des langages réguliers constructivement à partir des opérations ensemblistes usuelles et de la fermeture de Kleene.

Définition 3.3 (Langage régulier) L'ensemble des langages réguliers sur un alphabet Σ est défini par :

1. $\emptyset$ et $\{\varepsilon\}$ sont des langages réguliers.
2. $\{s\}$ est un langage régulier, pour tout $s \in \Sigma$.
3. si L_1 et L_2 sont des langages réguliers, alors $L_1 \cup L_2$, $L_1 \cdot L_2$ et ${L_1}^*$ sont des langages réguliers. ⧫

Exemple 3.3 Prenons par exemple $\Sigma = \{a, b\}$:
— $\{a, b\}^* = (\{a\} \cup \{b\})^*$ est le langage régulier constitué de l'ensemble des mots sur Σ
— $\{a, b\} \cdot \{a, b\}^* = (\{a\} \cup \{b\}) \cdot (\{a\} \cup \{b\})^*$ est le langage des mots non vides sur Σ
— $\{a, b\}^* \cdot \{a\} \cdot \{a, b\}^*$ est le langage des mots sur Σ qui contiennent au moins un a.
Maintenant, sur l'alphabet $\{0, 1\}$, le langage $\{0, 1\}^* \cdot \{0\}$ est l'ensemble des encodages binaires des entiers naturels pairs. ⧫

Le dernier langage dans l'exemple ci-dessus montre que certains langages réguliers sont reconnus par des automates finis puisque les automates de la figure 1.1 (page 10) les reconnaissent. Nous montrons en section 3.3 que de fait les langages réguliers sont précisément ceux qui sont reconnus par les automates finis. Une conséquence de ce résultat est que l'ensemble des langages réguliers est clos par les opérations booléennes classiques que sont l'intersection et la complémentation (en plus de l'union, par définition).

Lemme 3.1 Si L_1 et L_2 sont deux langages réguliers, alors :
— $\overline{L_1} = \Sigma^* \setminus L_1$, le complémentaire de L_1 est un langage régulier
— et $L_1 \cap L_2$ est un langage régulier. ⧫

Preuve. La preuve de ce résultat est laissée au lecteur qui a lu l'intégralité de ce chapitre. ■

3.2 Expressions régulières

Nous introduisons maintenant une notation algébrique des langages. Nous prouvons ensuite en section 3.3 qu'elle permet de définir précisément les langages réguliers.

Définition 3.4 (Expression régulière) L'ensemble des expressions régulières sur un alphabet Σ est défini par :

1. $\varnothing$, ϵ et s, quel que soit $s \in \Sigma$ sont des expressions régulières.

2. Si α et β sont deux expressions régulières, alors $(\alpha+\beta)$, $\alpha\beta$ et $(\alpha)^\bigstar$ sont des expressions régulières. ⧫

Notons que $\varnothing$ diffère de $\emptyset$, que ϵ diffère de ε et que $(.)^\bigstar$ diffère de $(.)^*$. Nous verrons par la suite que ces notations décrivent (construisent) les mêmes objets, mais dans deux algèbres différentes : respectivement les expressions régulières et les langages réguliers.

Exemple 3.4 Soit $\Sigma = \{a, b\}$ un alphabet :

1. $(a + b)^\bigstar$ est une expression régulière.
2. $(a + b)(a + b)^\bigstar$ est une expression régulière.
3. $(a + b)^\bigstar a(a + b)^\bigstar$ est une expression régulière.

Sur l'alphabet $\{0, 1\}$, $(0 + 1)^\bigstar 0$ est une expression régulière. ⧫

Les expressions régulières ont pour but de permettre la manipulation de langages. Ceux-ci étant en général infinis, il n'est en effet pas possible de les décrire, ni de les manipuler (notamment algorithmiquement), directement. Nous établissons maintenant le lien entre expression régulière et langage.

Définition 3.5 (Langage d'une expression régulière) Le langage représenté par une expression régulière est défini par :

1. $\mathcal{L}(\varnothing) = \emptyset$ et $\mathcal{L}(\epsilon) = \{\varepsilon\}$.
2. $\mathcal{L}(s) = \{s\}$ quel que soit $s \in \Sigma$.
3. $\mathcal{L}((\alpha + \beta)) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$.
4. $\mathcal{L}(\alpha\beta) = \mathcal{L}(\alpha) \cdot \mathcal{L}(\beta)$.
5. $\mathcal{L}(\alpha^\bigstar) = (\mathcal{L}(\alpha))^*$. ⧫

Nous pouvons maintenant établir des correspondances visibles entres les exemples 3.3 et 3.4.

Exemple 3.5 Reprenons les expressions régulières sur $\Sigma = \{a, b\}$ de l'exemple 3.4 :

1. $\mathcal{L}((a + b)^\bigstar) = (\mathcal{L}(a + b))^* = (\mathcal{L}(a) \cup \mathcal{L}(b))^* = (\{a\} \cup \{b\})^* = \{a, b\}^*$ est l'ensemble des mots sur Σ.
2. $\mathcal{L}((a + b)(a + b)^\bigstar) = \mathcal{L}(a + b) \cdot \mathcal{L}((a + b)^\bigstar) = \{a, b\} \cdot \{a, b\}^*$ est l'ensemble des mots non vides (*i.e.* différents de ε) sur Σ.
3. $\mathcal{L}((a + b)^\bigstar a(a + b)^\bigstar) = \mathcal{L}((a + b)^\bigstar) \cdot \mathcal{L}(a(a + b)^\bigstar) = \{a, b\}^* \cdot \mathcal{L}(a) \cdot \mathcal{L}((a + b)^\bigstar) = \{a, b\}^* \cdot a \cdot \{a, b\}^*$ est l'ensemble des mots sur Σ qui contiennent au moins un a.

Sur l'alphabet $\{0, 1\}$, $\mathcal{L}((0 + 1)^\bigstar 0) = \mathcal{L}((0 + 1)^\bigstar) \cdot \mathcal{L}(0) = \mathcal{L}((0 + 1))^* \cdot \{0\} = \{0, 1\}^* \cdot \{0\}$ est le langage des encodages binaires des entiers naturels pairs. ⧫

Quelques propriétés des expressions régulières :

— La somme est **commutative** : $\alpha+\beta = \beta+\alpha$ et **associative** : $(\alpha+\beta)+\gamma = \alpha+(\beta+\gamma)$.
— La concaténation est **associative** : $(\alpha\beta)\gamma = \alpha(\beta\gamma)$ mais elle n'est **pas commutative** : en général, $\alpha\beta \neq \beta\alpha$.
— La concaténation est **distributive** à gauche : $\alpha(\beta+\gamma) = \alpha\beta+\alpha\gamma$ et à droite : $(\beta+\gamma)\alpha = \beta\alpha + \gamma\alpha$ sur la somme.

— L'expression $\varnothing$ est l'**élément neutre** pour la somme : $\varnothing + \alpha = \alpha = \alpha + \varnothing$ et l'**élément absorbant** pour la concaténation : $\varnothing\alpha = \varnothing = \alpha\varnothing$.
— L'expression ϵ est l'**élément neutre** pour la concaténation : $\epsilon\alpha = \alpha = \alpha\epsilon$.
— Enfin, l'étoile de kleene est **prioritaire** sur la concaténation qui est elle-même est prioritaire sur la somme.

3.3 Théorème de Kleene

Les exemples 3.3 et 3.5 montrent que certains langages réguliers sont représentés par une expression régulière et inversement. Par ailleurs, l'exemple des encodages binaires des nombres naturels pairs montre que certains langages réguliers sont à la fois représentés par des expressions régulières, et reconnus par des automates finis (voir exemple 1.11 page 14). Nous montrons maintenant un lien fondamental entre les automates finis, les langages réguliers, et les expressions régulières.

Théorème 3.1 Les trois propositions suivantes sont équivalentes pour tout langage L :

1. L est reconnu par un automate fini
2. L est un langage régulier
3. L est représentable par une expression régulière ⧫

Ce théorème découle du résultat de la section 3.3.1 et du théorème de Kleene ci-dessous. Le théorème de Kleene est fondamental car il fait le lien entre l'algèbre (*i.e.* les expressions régulières) et le calcul (les automates finis) unifiant ainsi ces deux approches de la théorie des langages.

Théorème 3.2 (Théorème de Kleene) Un langage est régulier si et seulement s'il est reconnu par un automate fini. ⧫

La preuve du théorème de Kleene résulte des lemmes 3.4 et 3.5.

3.3.1 Des langages réguliers aux expressions régulières

Nous montrons tout d'abord que les langages réguliers peuvent être représentés par des expressions régulières.

Lemme 3.2 Tout langage régulier est représenté par une expression régulière ⧫

Preuve. Nous utilisons la technique d'induction pour prouver ce théorème. Pour cette preuve, nous notons (LRER) la propriété énoncée dans le lemme 3.2.

La preuve par induction s'appuie sur les définitions 3.3, 3.4 et 3.5. Soient les langages réguliers suivants :

1. «$\emptyset$». Nous avons $\emptyset = \mathcal{L}(\varnothing)$ par la définition 3.5, donc le langage régulier $\emptyset$ est représentable par l'expression régulière $\varnothing$.
2. «$\{\varepsilon\}$». D'après la définition 3.5, $\{\varepsilon\} = \mathcal{L}(\epsilon)$ donc le langage régulier $\{\varepsilon\}$ est représentable par l'expression régulière ϵ.
3. «$\{s\}$, pour $s \in \Sigma$». En utilisant la définition 3.5, $\{s\} = \mathcal{L}(s)$, donc le langage régulier $\{s\}$ est représentable par l'expression régulière s.

Nous avons montré que ces langages réguliers «de base» ont tous la propriété (LRER). Il nous reste à le montrer pour les constructions de la définition 3.3 : c'est l'étape d'induction. Soient L_1 et L_2 deux langages réguliers représentés respectivement par les expressions régulières α_1 et α_2 respectivement, c'est à dire $L_1 = \mathcal{L}(\alpha_1)$ et $L_2 = \mathcal{L}(\alpha_2)$.

1. «$L_1 \cup L_2$». Par hypothèse d'induction, $L_1 \cup L_2 = \mathcal{L}(\alpha_1) \cup \mathcal{L}(\alpha_2)$. Puis par la définition 3.5, $\mathcal{L}(\alpha_1) \cup \mathcal{L}(\alpha_2) = \mathcal{L}(\alpha_1 + \alpha_2)$. Donc le langage régulier $L_1 \cup L_2$ est représenté par l'expression régulière $\alpha_1 + \alpha_2$.
2. «$L_1 \cdot L_2$». Par hypothèse d'induction $L_1 \cdot L_2 = \mathcal{L}(\alpha_1) \cdot \mathcal{L}(\alpha_2)$. Par la définition 3.5, $\mathcal{L}(\alpha_1) \cdot \mathcal{L}(\alpha_2) = \mathcal{L}(\alpha_1 \alpha_2)$. Nous en déduisons que le langage régulier $L_1 \cdot L_2$ est représenté par l'expression régulière $\alpha_1 \alpha_2$.
3. «${L_1}^*$». Par hypothèse d'induction ${L_1}^* = \mathcal{L}(\alpha_1)^*$. Par la définition 3.5, $\mathcal{L}(\alpha_1)^* = \mathcal{L}({\alpha_1}^\bigstar)$. Il vient que le langage régulier L_1 est représenté par l'expression régulière ${\alpha_1}^\bigstar$.

Nous venons de montrer que ces trois constructions préservent la propriété (LRER). Par conséquent, la propriété (LRER) est démontrée. ■

Nous énonçons maintenant la contraposée : tout langage représenté par une expression régulière est régulier.

Lemme 3.3 Tout langage représenté par une expression régulière, est un langage régulier ⧫

Preuve. La preuve de ce lemme est laissée au lecteur. Elle est aisément réalisée par induction en se basant sur les définitions 3.5 et 3.3, et selon une structure similaire à la preuve du lemme 3.2. ■

3.3.2 Des expressions régulières aux automates finis

Nous présentons ci-dessous la construction de Thompson qui permet de calculer un automate fini qui accepte le langage décrit par une expression régulière donnée.

Lemme 3.4 Pour toute expression régulière α nous pouvons calculer un automate fini A tel que $\mathcal{L}(\alpha) = \mathcal{L}(A)$ ⧫

Preuve. Nous allons montrer ce lemme par induction : nous allons construire pour chaque structure d'expression régulière présentée dans la définition 3.4, un automate fini qui reconnaît le langage régulier correspondant.

Tous les automates construits dans cette preuve ont un seul état initial, noté i, et un unique état accepteur noté f. Pour les constructions inductives, nous ferons l'hypothèse que cette propriété est vraies, et notre construction fera en sorte de la préserver.

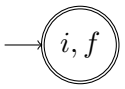
Nous commençons par présenter les cas de base, soient les expressions régulières suivantes :

1. «$\varnothing$». L'automate fini reconnaissant le langage $\mathcal{L}(\varnothing) = \emptyset$ ne doit accepter aucun mot. Il y a beaucoup d'automates qui remplissent cette tâche : tous ceux qui n'ont pas d'état accepteur accessible. Cependant, le plus petit automate qui ne reconnaît aucun mot est celui qui est réduit à un état initial non accepteur et un état accepteur non accessible [2] :

2. Formellement, l'état accepteur n'est pas nécessaire mais il permet de simplifier la construction puisque chaque règle que nous donnons dans cette preuve produit ainsi un automate fini avec exactement un état initial et un état accepteur.

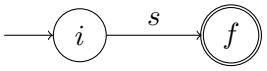
Soit $A_\varnothing$ l'automate ci-dessus, nous avons $\mathcal{L}(A_\varnothing) = \emptyset$ car l'état accepteur de $A_\varnothing$ est inaccessible, et donc $\mathcal{L}(A_\varnothing) = \mathcal{L}(\varnothing)$.

2. «ϵ». L'automate fini qui reconnaît le langage $\mathcal{L}(\epsilon) = \{\varepsilon\}$ doit donc accepter le mot d'entrée sans jamais lire un seul caractère. C'est donc un automate dont l'état initial est accepteur. Nous prenons le plus petit d'entre eux : celui qui n'a qu'un état et aucune transition :



Soit A_ϵ l'automate ci-dessus. Nous avons $\mathcal{L}(A_\epsilon) = \{\varepsilon\}$ puisque d'une part l'état initial est accepteur et d'autre part l'automate ne possède aucune transition. Nous en déduisons $\mathcal{L}(A_\epsilon) = \mathcal{L}(\epsilon)$.

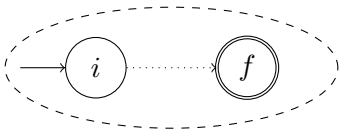
3. «$s,\ s \in \Sigma$». L'automate fini qui reconnaît le langage $\mathcal{L}(s) = \{s\}$ doit, à partir de l'état initial, lire s et atteindre ainsi un état accepteur :



Soit A_s l'automate ci-dessus. La seule exécution acceptante de A_s est $i \xrightarrow{s} f$, donc $\mathcal{L}(A_s) = \{s\}$. Il vient $\mathcal{L}(A_s) = \mathcal{L}(s)$.

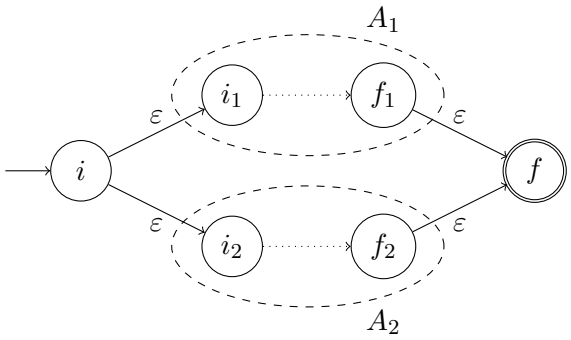
> ▷ Notons que ces trois langages sont reconnus par une infinité d'automates finis qui reconnaissent uniquement ces langages (toutes les autres transitions de l'automate ne conduisent jamais dans un état accepteur).

Il reste maintenant à démontrer que la somme (ou union), la concaténation et la fermeture de Kleene d'expressions régulières sont reconnues par des automates finis. Il s'agit de l'étape d'induction : nous considérons deux expressions régulières α_1 et α_2 qui sont reconnues par les automates finis A_1 et A_2 (hypothèse d'induction). Nous représentons A_1 et A_2 par leur état initial et leur état final (uniques par construction) de la façon suivante :



Nous considérons les expressions régulières suivantes :

1. «$\alpha_1 + \alpha_2$». L'automate fini reconnaissant ce langage doit choisir suivant le symbole lu si le mot appartient à $\mathcal{L}(\alpha_1)$ ou à $\mathcal{L}(\alpha_2)$, puis exécuter l'automate correspondant (A_1 ou A_2 respectivement) sur le mot d'entrée. Cependant, le premier symbole du mot ne permet pas nécessairement de choisir entre $\mathcal{L}(\alpha_1)$ et $\mathcal{L}(\alpha_2)$, et il peut être nécessaire de considérer un nombre arbitrairement grand de symboles du mot d'entrée pour faire ce choix. De plus, une fois ce choix fait, il faudrait revenir au début du mot d'entrée pour appliquer celui de A_1 et de A_2 qui aura été retenu, ce qu'un automate fini ne peut pas faire. L'idée consiste à utiliser les transitions ε afin de rester au début du mot d'entrée, et à utiliser le non-déterminisme pour faire le choix à notre place :

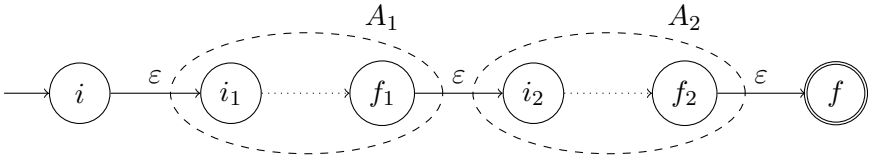


Ainsi, un mot d'entrée w est soumis à A_1 **et** à A_2. Soit w est accepté par A_1 et A_2, soit w est accepté par A_1 mais pas par A_2, et seule une (ou plusieurs) exécution(s) de A_1 accepte(nt), soit c'est la situation symétrique, soit finalement, w n'est accepté ni par A_1, ni par A_2, et ce mot est aussi rejeté par le nouvel automate.

Soit A_+ l'automate ci-dessus. Formellement, si $w \in \mathcal{L}(A_+)$ alors il existe une exécution $i \xrightarrow{w} f$ dans A_+. Soit cette exécution se décompose en $i \xrightarrow{\varepsilon} i_1 \xrightarrow{w} f_1 \xrightarrow{\varepsilon} f$ alors $w \in \mathcal{L}(A_1)$ par construction, soit inversement, elle se décompose en $i \xrightarrow{\varepsilon} i_2 \xrightarrow{w} f_2 \xrightarrow{\varepsilon} f$ alors $w \in \mathcal{L}(A_2)$. Nous en déduisons donc que $w \in \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$. Par hypothèse d'induction, $\mathcal{L}(A_1) = \mathcal{L}(\alpha_1)$ et $\mathcal{L}(A_2) = \mathcal{L}(\alpha_2)$, donc $w \in \mathcal{L}(\alpha_1 + \alpha_2)$.

Inversement, si $w \in \mathcal{L}(\alpha_1 + \alpha_2)$, alors $w \in \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$ par hypothèse d'induction. Si $w \in \mathcal{L}(A_1)$, alors il existe une exécution $i_1 \xrightarrow{w} f_1$ de A_1 qui accepte w, et donc $i \xrightarrow{\varepsilon} i_1 \xrightarrow{w} f_1 \xrightarrow{\varepsilon} f$ est une exécution acceptante de A_+. Nous montrons symétriquement que si $w \in \mathcal{L}(A_2)$, alors $i \xrightarrow{\varepsilon} i_2 \xrightarrow{w} f_2 \xrightarrow{\varepsilon} f$ est une exécution acceptante de A_+, il vient alors $w \in \mathcal{L}(A_+)$.

2. «$\alpha_1\alpha_2$». Les mots de ce langage sont ceux qui ont en première partie un mot de $\mathcal{L}(\alpha_1)$, et en seconde partie un mot de $\mathcal{L}(\alpha_2)$. L'automate reconnaissant $\mathcal{L}(\alpha_1) \cdot \mathcal{L}(\alpha_2)$ doit donc, pour tout mot reconnu par A_1, reconnaître ensuite un mot de L_2. Ceci revient donc à relier l'état final de A_1 à l'état initial de A_2 :

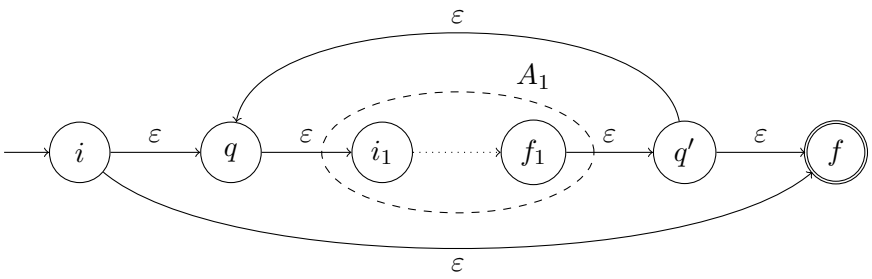


Soit $A.$ l'automate ci-dessus. Si $w \in \mathcal{L}(A.)$ alors $i \xrightarrow{w} f$ est une exécution acceptante de $A.$. Il existe donc deux mots w_1 et w_2 tels que cette exécution se décompose en $i \xrightarrow{\varepsilon} i_1 \xrightarrow{w_1} f_1 \xrightarrow{\varepsilon} i_1 \xrightarrow{w_2} f_2 \xrightarrow{\varepsilon} f$ et $w = w_1 \cdot w_2$. Alors, A_1 accepte w_1 et A_2 accepte w_2, donc par hypothèse d'induction $w_1 \in \mathcal{L}(\alpha_1)$ et $w_2 \in \mathcal{L}(\alpha_2)$ et $w \in \mathcal{L}(\alpha_1) \cdot \mathcal{L}(\alpha_2)$.

Considérons maintenant un mot $w \in \mathcal{L}(\alpha_1) \cdot \mathcal{L}(\alpha_2)$. Nous pouvons donc le décomposer en $w = w_1 \cdot w_2$ avec $w_1 \in \mathcal{L}(\alpha_1)$ et $w_2 \in \mathcal{L}(\alpha_2)$. Par hypothèse d'induction, w_1 est accepté par A_1, donc $i_1 \xrightarrow{w_1} f_1$ est une exécution de A_1, et symétriquement, w_2 est accepté par A_2, donc $i_2 \xrightarrow{w_2} f_2$ est une exécution de A_2. Il vient alors que $i \xrightarrow{\varepsilon} i_1 \xrightarrow{w_1} f_1 \xrightarrow{\varepsilon} i_1 \xrightarrow{w_2} f_2 \xrightarrow{\varepsilon} f$ est une exécution de $A.$, donc $w \in \mathcal{L}(A.)$.

3. «$\alpha_1{}^\bigstar$». Les mots de ce langages sont : le mot vide ε, les mots de $\mathcal{L}(\alpha_1)$, les mots de $\mathcal{L}(\alpha_1) \cdot \mathcal{L}(\alpha_1)$, les mots de $\mathcal{L}(\alpha_1) \cdot \mathcal{L}(\alpha_1) \cdot \mathcal{L}(\alpha_1)$, etc. L'automate reconnaissant la fermeture de Kleene de $\mathcal{L}(\alpha_1)$ doit donc permettre de soustraire le mot d'entrée à

A_1 (pour reconnaître le mot vide), ou au contraire d'appliquer successivement A_1 un nombre arbitraire de fois pour reconnaître l'entièreté du mot d'entrée :

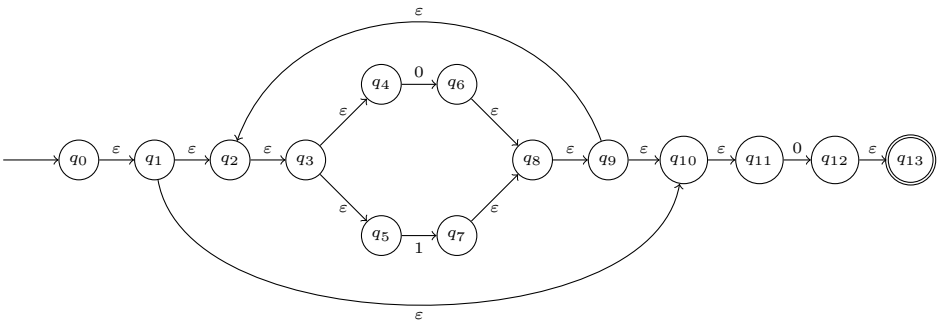


Soit $A_\star$ l'automate ci-dessus. Soit $w \in \mathcal{L}(A_\star)$. Si w est accepté sur l'exécution $i \xrightarrow{\varepsilon} f$ de $A_\star$, alors par définition, $w = \varepsilon \in \mathcal{L}({\alpha_1}^\star)$. Nous prouvons maintenant par induction que si $w \neq \varepsilon$ est accepté par $A_\star$ alors il appartient à $\mathcal{L}({\alpha_1}^\star)$. Le cas de base est constitué par l'exécution $i \xrightarrow{\varepsilon} q \xrightarrow{\varepsilon} i_1 \xrightarrow{w} f_1 \xrightarrow{\varepsilon} q' \xrightarrow{\varepsilon} f$. Donc, $w \in \mathcal{L}(A_1)$ et par hypothèse d'induction sur A_1, $w \in \mathcal{L}(\alpha_1)$ et finalement, $w \in \mathcal{L}({\alpha_1}^\star)$. Nous considérons maintenant une exécution de la forme : $i \xrightarrow{\varepsilon} q \xrightarrow{\varepsilon} i_1 \xrightarrow{w_1} f_1 \xrightarrow{\varepsilon} q \xrightarrow{w'} q' \xrightarrow{\varepsilon} f$ avec $w = w_1 \cdot w'$ et $w' \in \mathcal{L}(A_\star)$ puisque $i \xrightarrow{\varepsilon} q \xrightarrow{w'} q' \xrightarrow{\varepsilon} f$ est une exécution acceptante de $A_\star$. Nous avons $w_1 \in \mathcal{L}(A_1)$, donc $w_1 \in \mathcal{L}(\alpha_1)$ par hypothèse d'induction sur A_1. D'autre part, par hypothèse d'induction sur l'exécution de $A_\star$, il vient $w' \in \mathcal{L}({\alpha_1}^\star)$. Alors $w_1 \cdot w' \in \mathcal{L}(\alpha_1) \cdot \mathcal{L}({\alpha_1}^\star)$, et donc $w \in \mathcal{L}({\alpha_1}^\star)$.

Inversement, considérons un mot $w \in \mathcal{L}({\alpha_1}^\star)$. Si $w = \varepsilon$, alors $i \xrightarrow{\varepsilon} f$ est une exécution acceptante de $A_\star$, donc $w \in \mathcal{L}(A_\star)$. Nous prouvons maintenant par induction sur $w \neq \varepsilon$ son appartenance à $\mathcal{L}(A_\star)$. Puisque $w \in \mathcal{L}({\alpha_1}^\star) = \mathcal{L}(\alpha_1)^* = \bigcup_{i \geq 0} \mathcal{L}(\alpha_1)^i$, le cas de base est $w \in \mathcal{L}(\alpha_1)$. Par hypothèse d'induction sur A_1, $w \in \mathcal{L}(A_1)$, donc $i \xrightarrow{\varepsilon} q \xrightarrow{\varepsilon} i_1 \xrightarrow{w} f_1 \xrightarrow{\varepsilon} q' \xrightarrow{\varepsilon} f$ est une exécution de $A_\star$ qui accepte w. Supposons maintenant que $w = w_1 \cdot w'$ avec $w_1 \in \mathcal{L}(\alpha_1)$ et $w' \in \mathcal{L}({\alpha_1}^\star)$. Par hypothèse d'induction sur w, $i \xrightarrow{w'} f$ est une exécution de $A_\star$. D'autre part, par hypothèse d'induction sur A_1, puisque $w \in \mathcal{L}(\alpha_1)$, il vient $w \in \mathcal{L}(A_1)$, donc $i_1 \xrightarrow{w_1} f_1$ est une exécution de A_1. Alors, $i \xrightarrow{\varepsilon} q \xrightarrow{\varepsilon} i_1 \xrightarrow{w_1} f_1 \xrightarrow{\varepsilon} q \xrightarrow{w'} q' \xrightarrow{\varepsilon} f$ est une exécution de $A_\star$, donc $w \in \mathcal{L}(A_\star)$. ■

Notons qu'en corollaire de cette preuve nous avons une procédure effective pour calculer les opérations d'union, de concaténation et de fermeture de Kleene sur les langages réguliers, directement sur les automates qui les reconnaissent. Nous remarquons en outre que les automates construits par cette procédure sont non-déterministes puisqu'ils contiennent des transitions étiquetées ε.

Exemple 3.6 En appliquant la construction de Kleene donnée ci-dessus, nous calculons un automate fini non-déterministe pour l'expression régulière $(0 + 1)^\star 0$ représenté ci-dessous.



Nous laissons le soin au lecteur de comparer cet automate à ceux de la figure 1.1 page 10 qui reconnaissent le même langage. Aux chapitres 2 et 6 nous étudierons des techniques permettant de comparer ces automates. ⧫

3.3.3 Des automates finis aux expressions régulières

Le lemme suivant montre que pour tout automate fini, il est possible de construire une expression régulière qui représente le langage reconnu par cet automate. Cette construction est due à Robert McNaughton et Hisao Yamada.

Lemme 3.5 Pour tout langage reconnu par un automate fini A nous pouvons calculer une expression régulière α telle que $\mathcal{L}(A) = \mathcal{L}(\alpha)$. ⧫

Preuve. Soit $A = (Q, \Sigma, \delta, I, F)$ un automate fini, et soit $Q = \{q_1, \ldots, q_n\}$ l'ensemble de ses états. La preuve construit une expression régulière pour $\mathcal{L}(A)$ de la façon suivante :

- on associe une expression régulière élémentaire à chaque transition de l'automate,
- puis, en suivant la structure de l'automate on construit une expression régulière pour $\mathcal{L}(A)$ par application des opérateurs d'union, de concaténation, et de fermeture de Kleene à ces expressions élémentaires.

Nous notons $R(i, j, k)$ une expression régulière sur Σ qui représente les mots permettant de passer de l'état q_i à l'état q_j en ne passant que par des états parmi $\{q_1, \ldots, q_{k-1}\}$. C'est à dire que chaque symbole d'un mot de $\mathcal{L}(R(i, j, k))$ correspond à l'étiquette d'une transition menant de q_i à q_j, et ces transitions ne concernent que des états parmi $\{q_1, \ldots, q_{k-1}\}$.

Nous montrons maintenant comment construire $R(i, j, k)$ de façon inductive. L'idée est de construire successivement les expressions $R(i, j, k)$ en n'utilisant que des états intermédiaires dans l'ensemble $\emptyset$, puis dans l'ensemble $\{q_1\}$, puis dans l'ensemble $\{q_1, q_2\}$, et ainsi de suite jusqu'à l'ensemble $\{q_1, q_2, \ldots, q_n\}$.

Considérons en premier lieu le cas $k = 1$, c'est à dire les transitions dans l'automate entre q_i et q_j, sans passer par aucun état intermédiaire. Nous avons [3] :

$$R(i, j, 1) = \begin{cases} +_{(q_i, s, q_j) \in \delta}\ s & \text{si } i \neq j \\ \epsilon + (+_{(q_i, s, q_j) \in \delta}\ s) & \text{si } i = j \end{cases} \tag{3.1}$$

$R(i, j, 1)$ peut donc être vu comme les expressions régulières basiques : $\varnothing$, ϵ et s pour $s \in \Sigma$ qui peuvent être formées uniquement à partir des transitions de A.

3. Puisque $\varnothing$ est l'élément neutre pour la somme d'expressions régulières, $(+_{(q_i, s, q_j) \in \delta}\ s) = \varnothing$ s'il n'existe pas de telle transition dans δ.

Nous passons maintenant à l'étape d'induction, en considérant $k > 1$. C'est à dire que nous supposons que nous savons construire $R(i, j, k)$, et nous montrons comment construire $R(i, j, k + 1)$. Intuitivement, cela revient à produire des expressions régulières pour des séquences de transitions dans A. Les mots qui permettent de passer de q_i à q_j en ne passant que par des états parmi $\{q_1, \ldots, q_k\}$ sont décrits par $R(i, j, k + 1)$ et correspondent à :

— ceux qui permettent de passer de q_i à q_j en n'utilisant que $\{q_1, \ldots, q_{k-1}\}$. Par induction, ces mots sont décrits par $R(i, j, k)$.

— ceux qui permettent de passer de q_i à q_k, puis de q_k à q_k un certain nombre de fois, et enfin de q_k à q_j, le tout en n'utilisant que des états parmi $\{q_1, \ldots, q_{k-1}\}$. Ces trois étapes correspondent aux mots décrits par $R(i, k, k)$, $R(k, k, k)$ et $R(k, j, k)$ respectivement.

Ces deux situations sont représentées en figure 3.1.

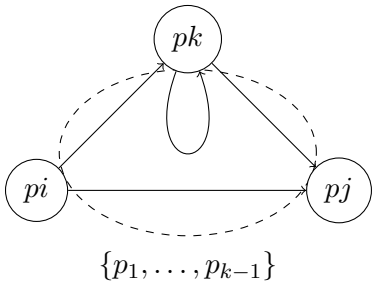


FIGURE 3.1 – Construction de $R(i, j, k)$.

Nous avons donc :

$$R(i, j, k + 1) = R(i, j, k) + R(i, k, k)\big(R(k, k, k)\big)^{\bigstar} R(k, j, k) \tag{3.2}$$

$R(i, j, k + 1)$ est bien une expression régulière puisqu'elle est formée, à partir d'expressions régulières (par induction, $R(\cdot, \cdot, k)$ est une expression régulière), en respectant la définition 3.4.

Notons que nous construisons bien $R(i, j, k + 1)$ inductivement, c'est à dire uniquement à partir de $R(., ., k)$ que nous avons définie précédemment. Plus précisément, nous avons défini $R(i, j, 1)$ et nous donnons ici un algorithme récursif pour construire $R(i, j, 2)$, $R(i, j, 3)$, etc.

Finalement, le langage reconnu par A est représenté par l'ensemble des mots permettant d'aller d'un état initial à l'un des états accepteurs, en utilisant (si nécessaire) tous les états de l'automate :

$$R(A) = +_{q_i \in I,\ q_j \in F}\ R(i, j, n + 1) \tag{3.3}$$

où n est le nombre d'états de l'automate. Nous voyons donc que $\mathcal{L}(A)$ est bien représentable par une expression régulière. ■

La définition inductive de $R(i, j, k)$ ci-dessus nous donne un algorithme pour calculer une expression régulière qui décrit le langage reconnu par un automate fini. Il suffit pour cela de partir de la définition de $R(A)$ (équation 3.3) et de développer récursivement l'expression selon l'équation 3.2 jusqu'à obtenir des expressions élémentaires de la forme $R(i, j, 1)$ auxquelles nous pouvons substituer des expressions régulières selon l'équation 3.1.

Exemple 3.7 Considérons l'automate A_2 de la figure 1.1(b) page 10 dont nous numérotons les états dans l'ordre $q_0 \mapsto 1$ et $q_1 \mapsto 2$. La figure 3.2 représente l'automate A obtenu après renumérotation des états.

$$\begin{aligned}
R(A) &= R(1,2,3) \\
&= R(1,2,2) + R(1,2,2)R(2,2,2)^\star R(2,2,2) \\
&= (R(1,2,1) + R(1,1,1)R(1,1,1)^\star R(1,2,1)) \\
&+ (R(1,2,1) + R(1,1,1)R(1,1,1)^\star R(1,2,1)) \\
&(R(2,2,1) + R(2,1,1)R(1,1,1)^\star R(1,2,1))^\star \\
&(R(2,2,1) + R(2,1,1)R(1,1,1)^\star R(1,2,1)) \\
&= (0 + (0+1)(0+1)^\star 0) \\
&+ (0 + (0+1)(0+1)^\star 0) \\
&(\epsilon + \varnothing(0+1)^\star 0)^\star \\
&(\epsilon + \varnothing(0+1)^\star 0)
\end{aligned}$$

En remarquant d'une part que $\varnothing$ est l'élément absorbant pour la concaténation et l'élément neutre pour la somme, et d'autre part que ϵ est l'élément neutre pour la concaténation (et la fermeture de Kleene), nous pouvons simplifier :

$$R(A) = 0 + (0+1)(0+1)^\star 0$$

Pour rappel, l'automate A_2 duquel nous sommes partis reconnaît les encodages binaires des entiers naturels pairs qui sont décrits par l'expression régulière $(0+1)^\star 0$. Nous pouvons prouver que l'expression régulière $R(A)$ décrit bien ce langage :

$$\begin{aligned}
\mathcal{L}(R(A)) &= \mathcal{L}(0 + (0+1)(0+1)^\star 0) \\
&= \mathcal{L}(0) \cup \mathcal{L}((0+1)(0+1)^\star 0) \\
&= \{0\} \cup \mathcal{L}((0+1)) \cdot \mathcal{L}((0+1)^\star) \cdot \mathcal{L}(0) \\
&= \{0\} \cup \{0,1\} \cdot \mathcal{L}((0+1))^* \cdot \{0\} \\
&= (\{\varepsilon\} \cup \{0,1\} \cdot \{0,1\}^*) \cdot \{0\} && \text{(factorisation)} \\
&= \left(\{0,1\}^0 \cup \{0,1\}^1 \cdot \bigcup_{i \geq 0} 0,1^i \right) \cdot \{0\} \\
&= \{0,1\}^* \cdot \{0\} \\
&= \mathcal{L}((0+1)^\star)\mathcal{L}(0) \\
&= \mathcal{L}((0+1)^\star 0)
\end{aligned}$$

♦

Cette construction permet donc d'obtenir une expression régulière qui décrit le langage reconnu par un automate fini. Elle est difficilement utilisable en pratique d'une part parce

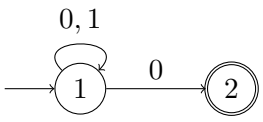


FIGURE 3.2 – L'automate A_2 de la figure 1.1(b) avec états renumérotés.

qu'elle engendre un grand nombre de calculs, et d'autre part parce que l'expression obtenue est tellement peu concise qu'elle en est inexploitable. Nous laissons le soin au lecteur de le vérifier sur l'automate A_1 de la figure 1.1(a) page 10. D'autres constructions existent pour construire une expression régulière qui décrit le langage reconnu par un automate fini, nous laissons le soin au lecteur de se référer à la bibliographie fournie au début du polycopié pour plus de détails.

Chapitre 4

Langages non réguliers

Au chapitre 1, nous avons formalisé les problèmes de décision par des langages. L'ensemble des instances positives est représenté par un langage L, et le problème de décision devient «est-ce que $w \in L$?» pour toute instance représentée par le mot w. Dans le même chapitre, nous avons introduit les automates finis comme modèle d'algorithme pour résoudre les problèmes de décision. C'est à dire que pour un langage L donné, si nous connaissons un automate fini qui reconnaît L, alors nous avons un algorithme qui décide $w \in L$.

Au chapitre précédent, nous avons étudié les langages réguliers. Nous avons particulièrement montré que les langages reconnus par les automates finis sont les langages réguliers. Ainsi donc, le problème de décision «$w \in L$? » peut être résolu par un automate fini uniquement si L est un langage régulier. Il se pose donc la question de l'existence de langages non réguliers. Nous montrons en section 4.1 qu'il existe de tels langages, puis nous donnons une condition nécessaire pour qu'un langage soit régulier en section 4.2. La négation de cette condition nécessaire donne une condition suffisante pour qu'un langage ne soit pas régulier, et donc pour prouver l'irrégularité.

4.1 Existence de langages non réguliers

Nous montrons maintenant qu'il existe strictement plus de langages qu'il n'existe de langages réguliers. L'ensemble des langages et l'ensemble des langages réguliers sont tous deux infinis. Nous introduisons donc dans un premier temps les outils permettant de comparer la taille de deux ensembles infinis.

Définition 4.1 (Bijection) Soient deux ensembles A et B et $f : A \to B$ une fonction de A dans B. La fonction f est une **bijection** si et seulement si :

— quels que soient $a, a' \in A$ avec $a \neq a'$, $f(a) \neq f(a')$ (injection) ;

— et, quel que soit $b \in B$, il existe $a \in A$ tel que $b = f(a)$ (surjection). ⧫

▷ Une bijection définit une correspondance (association) entre les éléments de A et de B, de telle façon que chaque élément de A identifie un unique élément de B, et vice-versa.

Exemple 4.1 — L'identité est une bijection triviale entre $\mathbb{N}$ et lui-même
— La fonction $f_2 : 2\mathbb{N} \to \mathbb{N}$ définie par $f_2(n) = n/2$ est une bijection de l'ensemble des nombres naturels pairs $2\mathbb{N}$ dans l'ensemble des nombres naturels $\mathbb{N}$
— La fonction $f_3 : \mathbb{Z} \to \mathbb{N}$ définie par $f_3(z) = 2z$ si $z \geq 0$ et $f_3(z) = -2z - 1$ si $z < 0$ est une bijection des entiers relatifs dans les entiers naturels ⧫

Définition 4.2 (Ensemble dénombrable) Un ensemble A est **dénombrable** si et seulement s'il existe une bijection $f : A \to \mathbb{N}$ entre A et l'ensemble des entiers naturels $\mathbb{N}$. ⧫

Un ensemble est donc dénombrable s'il est possible d'attribuer un numéro (entier naturel) à chacun de ses éléments. Intuitivement, on peut énumérer les éléments d'un ensemble dénombrable, bien qu'il soit infini. Il y a donc un premier élément, et tout élément admet un élément suivant. Ainsi, ces ensembles peuvent être définis par des fonctions récursives (ne terminant éventuellement pas).

Exemple 4.2 L'exemple 4.1 montre que l'ensemble des entiers naturels pairs $2\mathbb{N}$ d'une part et l'ensemble des entiers relatifs $\mathbb{Z}$ d'autre part sont dénombrables. Il montre par ailleurs qu'il y a "autant" d'entiers naturels pairs et d'entiers relatifs qu'il y a d'entiers naturels : ils sont tous les trois de taille $\aleph_0$.

A contrario, l'ensemble des nombres réels $\mathbb{R}$ n'est pas dénombrable. Nous le montrons maintenant afin d'introduire la technique de preuve par **diagonalisation** de Cantor. Supposons que $\mathbb{R}$ est dénombrable, alors il existe une bijection qui associe l'entier naturel 0 à un nombre réel, puis l'entier naturel 1 à un autre nombre réel, et ainsi de suite. Considérons uniquement la partie $]0;1[$ de $\mathbb{R}$, nous pouvons représenter la bijection de la façon suivante :

$\mathbb{N}$	$\mathbb{R}$
0	$0.\mathbf{1}3320900654\ldots$
1	$0.2\mathbf{7}124232478\ldots$
2	$0.73\mathbf{6}75190128\ldots$
3	$0.491\mathbf{2}4219075\ldots$
$\vdots$	$\vdots$

Nous construisons alors un nombre réel dont la première décimale diffère de celle en gras dans le nombre réel d'index 0, dont la seconde décimale diffère de celle en gras dans le nombre réel d'index 1, dont la troisième décimale diffère de celle en gras dans le nombre réel d'index 2, et ainsi de suite. Le nombre réel $r = 0.3458\ldots$ dont la suite des décimales est obtenue par le même procédé est un exemple d'un tel nombre réel. Cependant, le nombre r a beau appartenir à $\mathbb{R} \cap]0;1[$, il ne peut figurer dans aucune ligne du tableau précédent (*i.e.* de la bijection) puisqu'il diffère de chacun des nombres réels par au moins toutes les décimales en gras (la fameuse diagonale). Nous en déduisons qu'il n'existe pas de bijection entre $\mathbb{N}$ et $\mathbb{R}$, et que $\mathbb{R}$ n'est pas dénombrable. ⧫

Nous montrons maintenant que l'ensemble des mots sur un alphabet donné est dénombrable.

Lemme 4.1 L'ensemble Σ^* des mots sur un alphabet Σ est dénombrable. ⧫

Preuve. Pour prouver ce résultat, nous devons exhiber une bijection f entre l'ensemble des mots finis de Σ^* et l'ensemble des naturels $\mathbb{N}$.

Remarquons qu'il y a un nombre fini de mots de longueur n fixée, pour $n \geq 0$, dans Σ^*. La solution consiste donc à énumérer les mots de Σ^* par longueur croissante, et pour les mots de même longueur, à les classer par ordre lexicographique. Ceci construit effectivement une bijection entre Σ^* et $\mathbb{N}$. ■

Puisque l'ensemble des mots sur un alphabet fixé Σ est dénombrable, l'ensemble des expressions régulières sur Σ est lui aussi dénombrable. En effet, une expression régulière sur Σ est un mot sur $\Sigma \cup \{\varnothing, \epsilon, +, ^\star, (,)\}$.

Nous montrons maintenant que l'ensemble des langages est indénombrables.

Lemme 4.2 L'ensemble des parties d'un ensemble dénombrable n'est pas dénombrable. ♦

Preuve. La preuve de ce lemme se fait par contradiction en utilisant la méthode de **diagonalisation**. Soit donc un ensemble dénombrable $A = \{a_1, a_2, \ldots\}$, et supposons que l'ensemble des ses sous-ensembles $S = \{s_1, s_2, \ldots\}$ est dénombrable. Nous pouvons alors construire le tableau suivant :

	a_1	a_2	a_3	a_4	$\cdots$
s_1	×		×	×	$\cdots$
s_2		×	×		$\cdots$
s_3	×				$\cdots$
s_4	×	×			$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

où × dans la colonne (s_i, a_j) indique que $a_j \in s_i$, et à l'inverse, l'absence de × signifie que $a_j \notin s_i$.

Le principe de la preuve est de montrer qu'il existe un élément de S qui n'est pas énuméré dans ce tableau, ce qui permet de conclure que S n'est pas dénombrable. Soit :

$$D = \{a_i \mid a_i \notin s_i\} \tag{4.1}$$

L'ensemble D est bien un sous-ensemble de A, cependant, il ne peut être l'un des s_i du tableau. En effet, supposons que $D = s_k$, alors :

— si $a_k \in s_k$ alors $a_k \in D$ par l'hypothèse $D = s_k$, ce qui contredit l'équation (4.1).

— inversement, si $a_k \notin s_k$, alors $a_k \in D$ par l'équation (4.1), ce qui contredit l'hypothèse $D = s_k$.

Par conséquent D n'existe pas, et S n'est donc pas dénombrable. ■

Nous déduisons alors des deux lemmes précédents qu'il existe des langages non réguliers. Ils ne sont donc pas reconnus par des automates finis, mais par des machines plus complexes [1]. Ils ne sont pas non plus représentables par des expressions régulières.

Théorème 4.1 Il existe des langages qui ne sont pas réguliers. ♦

1. Les automates à pile seront vus dans le cours de «Compilation», et les machines de Turing sont étudiées dans le cours de «Calculabilité et complexité».

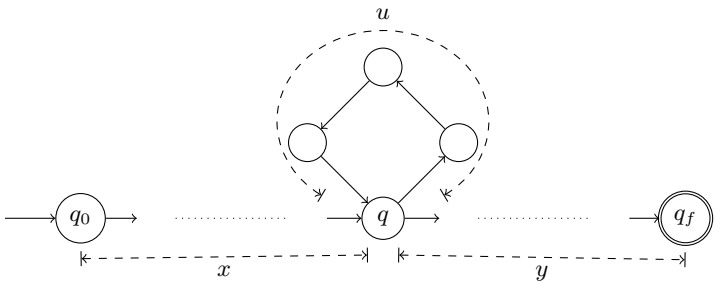
Preuve. Par le lemme 4.1, nous savons qu'il existe un nombre dénombrable d'expressions régulières puisqu'une expression régulière peut être vue comme un mot sur l'alphabet $\Sigma \cup \{\varnothing, \epsilon, +, \star\}$ où Σ est un alphabet de symboles. Par le théorème 3.1, nous savons qu'il y a autant de langages réguliers que d'expressions régulières, et donc que l'ensemble des langages réguliers est dénombrable. Par le lemme 4.2, l'ensemble des langages étant l'ensemble des (parties des) ensembles de mots, nous savons que l'ensemble des langages est non dénombrable.

Par conséquent, certains langages (en nombre infini) ne sont pas réguliers. ■

4.2 Lemme de l'étoile

Il existe donc des langages qui ne sont pas réguliers. Il n'est pas possible de déterminer si un langage est régulier. Cependant, nous exhibons maintenant une condition suffisante pour qu'un langage ne soit pas régulier.

1. Remarquons tout d'abord qu'un langage non régulier est nécessairement infini.
2. Puisqu'un alphabet est fini, un langage infini contient des mots de longueur non bornée.
3. Tout langage régulier est reconus par un automate fini A. Soit N le nombre d'états de A.
4. Tout mot w appartenant à $\mathcal{L}(A)$, et tel que $|w| > N$, est reconnu par une exécution de A qui passe au moins deux fois par le même état :



5. Par conséquent, le langage $x \cdot u^* \cdot y$ est inclus dans $\mathcal{L}(A)$: il est possible de «boucler» autant de fois que souhaité sur l'état q avant de se diriger vers q_F.

Ceci nous donne donc une condition nécessaire pour qu'un langage soit régulier : tout mot suffisamment long doit contenir un sous mot qui peut-être répété un nombre arbitrairement grand de fois. Ceci n'est pas un moyen de prouver qu'un langage est régulier : il faudrait considérer un nombre infini de mots (cette condition est nécessaire mais non suffisante). Par contre, cette méthode permet de prouver qu'un langage n'est pas régulier en montrant qu'il comporte (au moins) un mot qui ne respecte pas ce principe.

Théorème 4.2 (Lemme de l'étoile/Pumping lemma) Soient L un langage régulier infini, et $A = (Q, \Sigma, \delta, q_0, F)$ un automate fini déterministe tel que $L = \mathcal{L}(A)$. Pour tout mot $w \in L$ tel que $|w| \geq \mathsf{Card}(Q)$, il existe $x, u, y \in \Sigma^*$, tels que $u \neq \varepsilon$ et $|x \cdot u| \leq \mathsf{Card}(Q)$, vérifiant : $x \cdot u \cdot y = w$ et $\forall k \geq 0,\ x \cdot u^k \cdot y \in L$. ♦

Preuve. Puisque $w \in L$, il existe une exécution $q_0 \xrightarrow{w} q$ de A telle que $q \in F$. Sachant que $|w| \geq \mathsf{Card}(Q)$, il existe au moins un état q' qui apparaît au moins deux fois sur cette exécution. Nous pouvons par ailleurs choisir parmi tous les états candidats l'état q' qui est le plus proche

de q_0 sur l'exécution $q_0 \xrightarrow{w} q$. Nous pouvons alors la découper en : $q_0 \xrightarrow{x} q' \xrightarrow{u} q' \xrightarrow{y} q$. Puisque q' est choisi le plus proche possible de q_0, nous avons $|x \cdot u| \leq \mathsf{Card}(Q)$. Par ailleurs $u \neq \varepsilon$ puisque q' apparaît au moins deux fois sur l'exécution $q_0 \xrightarrow{w} q$ et A est déterministe. Finalement, pour tout $k \in \mathbb{N}$, $q_0 \xrightarrow{x} q' \xrightarrow{u^k} q' \xrightarrow{y} q$ est une exécution de A, et par $q \in F$ il vient, $x \cdot u^k \cdot y \in L$. ■

Afin de prouver qu'un langage n'est pas régulier nous falsifions le lemme de l'étoile. Intuitivement, il s'agit de prouver que quel que soit l'automate fini A considéré, celui-ci ne peut pas reconnaître L. Il n'est bien entendu pas possible d'examiner tous les automates finis, mais nous quantifions de la façon suivante :

— La taille de l'automate A devient un paramètre de la preuve, et nous choisissons w de tel sorte à ce qu'il soit plus grand que ce nombre d'états afin de faire apparaître un cycle sur tout chemin acceptant w dans A.

— D'autre part, la décomposition $w = x \cdot u \cdot y$ devient le second paramètre de la preuve. Nous considérons donc toutes les décompositions satisfaisant les contraintes sur x, u et y, et nous fixons une k telle que $x \cdot u^k \cdot y \notin L$.

> ▷ Le lemme de l'étoile permet de montrer qu'il n'existe pas d'automate fini **déterministe** qui reconnaît un langage L donné. Cela suffit à montrer qu'il n'existe pas non plus d'automate **non-déterministe** qui reconnaît L puisque nous avons vu au chapitre 2 qu'à tout automate non-déterministe correspond un automate déterministe qui reconnaît le même langage.

Exemple 4.3 Nous allons montrer que le langage :

$$L = \{a^n \cdot b^n \,|\, n \geq 0\}$$

sur l'alphabet $\Sigma = \{a, b\}$ n'est pas régulier.

1. Supposons que L est régulier, alors il est reconnu par (au moins) un automate fini déterministe A. Soit N le nombre d'états de A, prenons alors $w = a^N \cdot b^N$.
2. Toute décomposition $w = x \cdot u \cdot y$ qui respecte à la fois $|x \cdot u| \leq N$ et $u \neq \varepsilon$ est telle que $x = a^i$, $u = a^j$ et $y = a^l \cdot b^N$ avec $i + j + l = N$ et $j \neq 0$.
3. Alors, le mot $x \cdot u^2 \cdot y = a^i \cdot a^{2j} \cdot a^l \cdot b^N = a^{N+j} \cdot b^N$ n'est pas dans L puisque $j > 0$.
4. Ceci entre donc en contradiction avec le lemme de l'étoile (ou satisfait sa négation), par conséquence l'hypothèse «L est régulier» est fausse.

Notons que dans cet exemple N, x, u et y sont des paramètres et que nous quantifions bien ainsi sur tous les automates possibles A conformément au lemme de l'étoile. En effet, N est une quantification sur le nombre d'états de A, et x, u, y quantifient sur tous les cycles (donc les transitions) de A. Notons enfin les étapes de la preuve :

1. le choix de w qui doit être fait pour favoriser un découpage $x \cdot u \cdot y$ intéressant ;
2. le découpage de w selon les contraintes $|x \cdot u| \leq N$ et $u \neq \varepsilon$ qui considère **toutes les décompositions x, u, y possibles** ;

3. la mise en évidence d'une puissance particulière de u qui vient briser la symétrie de w. Notons l'importance de $u \neq \varepsilon$ pour contredire le lemme de l'étoile
4. la contradiction qui conclut la preuve. ♦

Le théorème 4.2 et l'exemple 4.3 montrent les limites des automates finis. Ils ne peuvent pas mémoriser une information de taille non bornée. Ainsi, pour le langage $L = \{a^n b^n \mid n \in \mathbb{N}\}$ de l'exemple 4.3, le nombre n de symboles a reconnus avant de lire les symboles b ne peut pas être stocké pour un nombre de a arbitrairement grand. En effet, la seule mémoire dont dispose un automate fini est constituée par ses états, en nombre fini.

Chapitre 5

Grammaires

Un automate fini est une description **analytique** d'un langage régulier : c'est un algorithme pour reconnaître les mots du langage. Les **grammaires** donnent une description **générative** d'un langage : elles explicitent des règles de construction des mots du langage.

Nous connaissons déjà la notion de grammaires en langage naturel. Par exemple, en français, une phrase déclarative a la forme :

"sujet" "verbe" "complément"

La grammaire a pour rôle de fixer la structure des phrases.

En informatique, les grammaires sont couramment utilisées pour définir la syntaxe des langages de programmation et pour construire les compilateurs. Par exemple, la grammaire suivante : http://www.cs.man.ac.uk/~pjj/bnf/c_syntax.bnf génère l'ensemble des programmes C.

5.1 Grammaire, dérivation, langage

Définition 5.1 (Grammaire) Une **grammaire** est un quadruplet $G = (V, \Sigma, R, S)$ où :
- V et Σ sont deux alphabets disjoints. Les symboles de V sont dits **non-terminaux**, ceux de Σ sont dits **terminaux**.
- $R \subseteq (V \cup \Sigma)^* \times (V \cup \Sigma)^*$ est un ensemble fini de **règles** (α, β) telles que α contient au moins un symbole non-terminal.
- $S \in V$ est le symbole non-terminal **initial**. ⧫

Les symboles non-terminaux (i.e. symboles de V) sont usuellement notés par des lettres majuscules $A, B, \ldots$, alors que les symboles terminaux (i.e. symboles de Σ) sont représentés par des lettres minuscules $a, b, \ldots$. L'ensemble R définit les règles de **substitution**. Une règle (α, β) sera souvent représentée sous la forme $\alpha \to \beta$. Elle signifie que β peut être substitué à α. Le symbole non-terminal initial est généralement noté S. Lorsqu'une grammaire comporte plusieurs règles avec le même membre gauche, par exemple : $A \to aA$ et $A \to Bb$, on écrit souvent : $A \to aA \,|\, Bb$ pour simplifier (le symbole | signifie «ou »).

Exemple 5.1 La grammaire G_1 ci-dessous définit cinq règles. Les deux premières signifient que S peut être remplacé par aS ou par bT. Les trois dernières indiquent que T peut être remplacé par bS, aT ou ε.

$$\begin{aligned} S &\longrightarrow aS \\ S &\longrightarrow bT \end{aligned} \qquad \begin{aligned} T &\longrightarrow bS \\ T &\longrightarrow aT \\ T &\longrightarrow \varepsilon \end{aligned}$$

On aurait également pu l'écrire sous la forme :

$$\begin{aligned} S &\to aS \,|\, bT \\ T &\to bS \,|\, aT \,|\, \varepsilon \end{aligned}$$ ♦

Une grammaire est un système de génération de mots. À partir du symbole initial S, les règles de substitution permettent de dériver des mots sur l'alphabet $(V \cup \Sigma)$.

Définition 5.2 Une grammaire $G = (V, \Sigma, R, S)$ **dérive** $v \in (V \cup \Sigma)^*$ à partir de $u \in (V \cup \Sigma)^*$, noté $u \Rightarrow v$, s'il est possible de décomposer $u = xu'y$, $v = xv'y$ et $u' \to v'$ est une règle de R.♦

Exemple 5.2 Considérons la grammaire G_1 introduite à l'exemple 5.1.
- $S \Rightarrow aS$ est une dérivation qui applique la règle $S \to aS$
- $abS \Rightarrow abbT$ est une dérivation qui applique la règle $S \to bT$
- $abTb \Rightarrow abb$ est une dérivation qui applique la règle $T \to \varepsilon$
- $S \Rightarrow \varepsilon$ ne correspond pas à l'application d'une règle de G_1
- $aSTb \Rightarrow abTb$ ne correspond pas à l'application d'une règle de G_1 ♦

Une dérivation consiste donc à remplacer un facteur u' de u par un mot v' selon une des règles de la grammaire G. Ceci se généralise à des applications successives de règles : on note $u \Rightarrow^* v$ s'il existe une séquence finie de dérivations :

$$u \Rightarrow w_0 \Rightarrow w_1 \Rightarrow \cdots \Rightarrow w_n \Rightarrow v$$

Parmi toutes les dérivations, celles qui produisent des mots sur l'alphabet des symboles terminaux Σ nous intéressent plus particulièrement.

Définition 5.3 Le **langage généré** par une grammaire $G = (V, \Sigma, R, S)$ est l'ensemble des mots dérivés depuis le symbole initial S qui ne contiennent que des symboles terminaux :

$$\mathcal{L}(G) = \{w \in \Sigma^* \,|\, S \Rightarrow^* w\}$$ ♦

Exemple 5.3 Considérons à nouveau la grammaire G_1 introduite à l'exemple 5.1.
- $S \Rightarrow bT \Rightarrow baT \Rightarrow ba$ est une séquence de dérivation de G_1
- $aSTb \Rightarrow abTTb \Rightarrow abTb$ est une séquence de dérivations de G_1

Nous avons $ba \in \mathcal{L}(G_1)$ mais $abTb \notin \mathcal{L}(G_1)$ car T est un symbole non-terminal.

Le langage de G_1 est constitué de tous les mots sur $\{a, b\}$ qui contiennent un nombre impair de b. ♦

5.2 Grammaires régulières et langage réguliers

Dans cette section, on étudie les grammaires qui génèrent les langages réguliers.

Définition 5.4 Une grammaire $G = (V, \Sigma, R, S)$ est **linéaire droite** si toutes les règles de R ont la forme $A \to wB$ ou $A \to w$ pour $w \in \Sigma^*$ et $B \in V$. ⧫

▷ On peut également définir la notion de grammaire **linéaire gauche** : ses règles ont la forme $A \to Bw$ ou $A \to w$ pour $w \in \Sigma^*$ et $B \in V$. Nous verrons en TD que les grammaires linéaires gauches et droites sont équivalentes : toute grammaire linéaire gauche peut être transformée en une grammaire linéaire droite qui génère le même langage, et inversement.

Définition 5.5 On appelle **grammaire régulière** une grammaire qui est soit linéaire gauche, soit linéaire droite. ⧫

▷ Il existe dans la littérature une notion plus générale de grammaire **linéaire** : les grammaires dont les membres droits contiennent au plus un symbole non terminal. Formellement, les règles d'une grammaire linéaire ont la forme : $\alpha \to \beta$ avec $\beta \in \Sigma^* V \Sigma^*$.
Les grammaires linéaires droites et gauches forment deux sous-classes des grammaires linéaires. Comme on le montre ci-dessous, les grammaires linéaires droites et linéaires gauches génèrent les langages réguliers.
Certaines grammaires linéaires (ni droites, ni gauches) génèrent des langages non-réguliers. Par exemple la grammaire suivante génère le langage $\{a^n b^n \mid n \geq 0\}$.

$$S \to aA \mid Bb \mid \varepsilon$$
$$A \to Sb$$
$$B \to aS$$

Théorème 5.1 Un langage L est régulier si et seulement si il est généré par une grammaire régulière. ⧫

Nous avons vu au chapitre 3 que les langages réguliers sont ceux qui sont reconnus par un automate fini. La preuve du théorème 5.1 repose sur la construction d'un automate fini qui reconnaît le langage généré par une grammaire régulière donnée, et inversement. Nous montrons tout d'abord comment construire un automate fini qui reconnaît le langage généré par une grammaire linéaire droite. On rappelle que toute grammaire régulière est équivalente à une grammaire linéaire droite.

Lemme 5.1 Pour toute grammaire linéaire droite $G = (V, \Sigma, R, S)$, il existe un automate fini A_G qui reconnaît le langage $\mathcal{L}(G)$. ⧫

Preuve. On construit l'automate non-déterministe $A_G = (Q, \Sigma, \delta, I, F)$ où :

— $Q = V \cup \{f\}$ avec $f \notin V$. Les états de A_G correspondent aux symboles non-terminaux de G, plus un état supplémentaire noté f.
— $I = \{S\}$,
— $F = \{f\}$,
— enfin, pour toute règle $A \to wB$, il existe une transition $A \xrightarrow{w} B$ dans δ, et pour toute règle $A \to w$ de R, il existe une transition $A \xrightarrow{w} f$ dans δ.

On montre que $S \Rightarrow w_1B_1 \Rightarrow w_1w_2B_2 \Rightarrow \cdots \Rightarrow w_1w_2 \dots w_nB_n$ est une dérivation de G si et seulement si $S \xrightarrow{w_1} B_1 \xrightarrow{w_2} B_2 \cdots \xrightarrow{w_n} B_n$ est une exécution de A_G par récurrence sur n.

Enfin, on montre que les langages coïncident en remarquant que $S \Rightarrow \cdots \Rightarrow w_1w_2 \dots w_nB_n \Rightarrow w_1w_2 \dots w_nw_{n+1}$ avec $w_{n+1} \in \Sigma^*$ si et seulement si $S \xrightarrow{w_1} \cdots \xrightarrow{w_n} B_n \xrightarrow{w_{n+1}} f$. ■

Il reste maintenant à démontrer comment construire une grammaire régulière (linéaire droite) qui génère le langage reconnu par un automate fini donné. Sans perte de généralité, on suppose que l'automate fini est déterministe (voir chapitre 2).

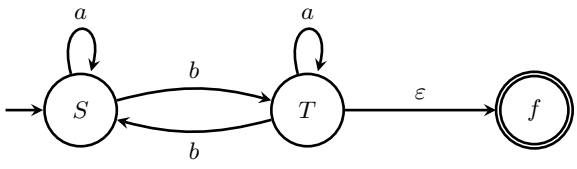
Lemme 5.2 Pour tout automate fini déterministe $A = (Q, \Sigma, \delta, q_0, F)$, il existe une grammaire régulière qui génère le langage $\mathcal{L}(A)$. ♦

Preuve. On construit la grammaire $G_A = (V, \Sigma, R, S)$ où :
— $V = Q$, il y a un symbole non-terminal par état de A.
— $S = q_0$, l'état initial de A.
— R est l'ensemble consistué des règles : $q \to wq'$ pour toute transition $q \xrightarrow{w} q'$ de δ, et $q \to \varepsilon$ pour tout état acceptant $q \in F$.

Comme précédemment, on montre que $S \Rightarrow w_1B_1 \Rightarrow w_1w_2B_2 \Rightarrow \cdots \Rightarrow w_1w_2 \dots w_nB_n$ est une dérivation de G_A si et seulement si $S \xrightarrow{w_1} B_1 \xrightarrow{w_2} B_2 \cdots \xrightarrow{w_n} B_n$ est une exécution de A par récurrence sur n.

La définition des règles $q \to \varepsilon$ pour $q \in F$ permet de conclure à l'égalité de $\mathcal{L}(A)$ et $\mathcal{L}(G_A)$. ■

Exemple 5.4 D'après la preuve du lemme 5.1, on construit l'automate A_1 qui reconnaît $\mathcal{L}(G_1)$:



♦

5.3 Au-delà des grammaires régulières

Toutes les grammaires ne sont pas régulières. Comme nous l'avons vu précédemment, il existe des grammaires (possiblement linéaires) qui génèrent des langages non-réguliers.

Exemple 5.5 La grammaire G_2 définie par $S \to aSb \,|\, \varepsilon$ génère le langage $\{a^nb^n \,|\, n \in \mathbb{N}\}$. ♦

Le linguiste et informaticien Noam Chomsky a défini la classification suivante des grammaires :

Type 3 Les grammaires régulières dont les règles sont toutes linéaires droites :

$$A \to wB$$
$$A \to w$$

ou alors toutes linéaires gauches :

$$A \to Bw$$
$$A \to w$$

où $A, B \in V$ et $w \in \Sigma^*$. Le membre gauche d'une règle est seulement constitué d'un symbole non-terminal. Le membre droit contient au plus un symbole non-terminal, à droite (resp. à gauche) des symboles terminaux.

Le langage généré par une grammaire régulière est régulier, et donc reconnu par un automate fini.

Type 2 Les grammaires hors-contextes dont les productions sont de la forme :

$$A \to \beta$$

où $A \in V$ et $\beta \in (V \cup \Sigma)^*$. Seul le membre gauche est contraint : il ne peut s'agir que d'un seul symbole non-terminal.

Le langage d'une grammaire hors-contexte est reconnu par un automate à pile.

Type 1 Les grammaires contextuelles ont des productions de la forme :

$$\alpha \to \beta$$

où $\alpha, \beta \in (V \cup \Sigma)^*$, α contient au moins un symbole non-terminal, et β contient au moins autant de symboles que α : $|\alpha| \leq |\beta|$. La règle $S \to \varepsilon$ est autorisée si S n'apparaît en membre droit d'aucune règle.

Les langages contextuels sont reconnus par des automates linéairement bornés.

Type 0 Aucune restriction. Les langages définis par de telles grammaires sont reconnus par des machines de Turing, sans garantie de terminaison (récursivement énumérable).

On peut montrer que les familles de langages associées à ces types sont incluses les unes dans les autres, mais non égales. Ainsi, tous les langages générés avec une grammaire de type i ($1 \leq i \leq 3$) peuvent être générés par une grammaire de type $i - 1$. De plus, pour chaque $0 \leq i \leq 2$, il existe des langages générés par une grammaire de type i mais par aucune grammaire de type $i+1$. Il existe également des langages qui ne sont générés par aucune grammaire.

La grammaire G_1 de l'exemple 5.1 fait partie du type 3 et la G_2 de l'exemple 5.5 du type 2 ; aucune grammaire de type 3 ne génère le langage généré par G_2, puisque ce dernier est non-régulier. L'exemple suivant donne une grammaire G_3 de type 1 qu'il n'est pas possible de transformer en grammaire de type 2.

Exemple 5.6 Soit la grammaire G_3 suivante :

$$S \longrightarrow abc \qquad cB \longrightarrow Bc$$
$$S \longrightarrow aSBc \qquad bB \longrightarrow bb$$

Le langage généré est $\mathcal{L}(G_3) = \{a^n b^n c^n | n \geq 1\}$. En effet, on montre par récurrence sur le nombre de substitutions que dans tout mot dérivé, le nombre de lettres a est égal au nombre de lettres c ainsi qu'au nombre de lettres b et B additionnées ; de plus, les mots dérivés sont d'une des deux formes $a^*S(B+c)^*$ ou $a^*b^*(B+c)^*$. Dans les deux cas, les terminaux a sont toujours placés avant les terminaux b, eux-mêmes placés avant les terminaux c. Lorsque le dernier symbole non-terminal est substitué, le mot généré est donc bien de la forme $a^n b^n c^n$. Inversement, tout mot $a^n b^n c^n$ $(n > 0)$ peut être généré en appliquant $n-1$ fois la règle $S \longrightarrow aSBc$, une fois la règle $S \longrightarrow abc$, $\frac{n(n-1)}{2}$ fois la règle $cB \longrightarrow Bc$ puis $n-1$ fois la règle $bB \longrightarrow bb$. ⧫

L'exemple suivant fournit une grammaire de type 0 qui n'est pas de type 1.

Exemple 5.7 La grammaire G_4 ci-dessous n'est pas de type 1 à cause des règles $E \longrightarrow \varepsilon$ et $F \longrightarrow \varepsilon$. Déterminer le langage généré est laissé au soin du lecteur ...

$$\begin{aligned} S &\longrightarrow EF \\ E &\longrightarrow aEA \mid bEB \mid \varepsilon \\ AF &\longrightarrow aF \\ BF &\longrightarrow bF \\ F &\longrightarrow \varepsilon \end{aligned} \qquad \begin{aligned} Aa &\longrightarrow aA \\ Ab &\longrightarrow bA \\ Ba &\longrightarrow aB \\ Bb &\longrightarrow bB \end{aligned}$$

Notez toutefois que $\mathcal{L}(G_4)$ peut être généré avec une grammaire de type 1 ; pour cela, il suffit de dupliquer toutes les règles mettant en jeu E et F en une règle avec E_1 et F_1 et une règle avec E_2 et F_2, de remplacer les règles $E \longrightarrow \varepsilon$ et $F \longrightarrow \varepsilon$ par $E_1 \longrightarrow a$, $F_1 \longrightarrow a$, $E_2 \longrightarrow b$, $F_2 \longrightarrow b$, et d'ajouter la règle $S \longrightarrow \varepsilon$. ⧫

5.4 Arbre de dérivation, ambiguïté

Les grammaires sont fréquemment utilisées pour décrire des langages naturels, des langages de programmation, ou encore des langages de description. Elles permettent de décrire de manière formelle ces langages et donc de construire des compilateurs pour ces langages. Le compilateur vérifie qu'un mot appartient au langage, et il reconstitue en général la structure de ce mot afin de pouvoir lui appliquer des transformations.

Exemple 5.8 Un compilateur pour le langage C doit d'une part valider l'expression $1+2$ et invalider $+(1)2$. Il doit en outre découvrir la structure de l'expression $1+2$: «opérateur $+$ appliqué aux opérandes 1 et 2 », afin de pouvoir la traduire en langage assembleur. ⧫

La structure d'un mot correspond aux différentes règles de la grammaire nécessaires à sa construction. On la représente par un arbre de dérivation.

Définition 5.6 Un **arbre de dérivation** d'un mot w par une grammaire hors-contexte $G = (V, \Sigma, R, S)$ est un arbre fini :

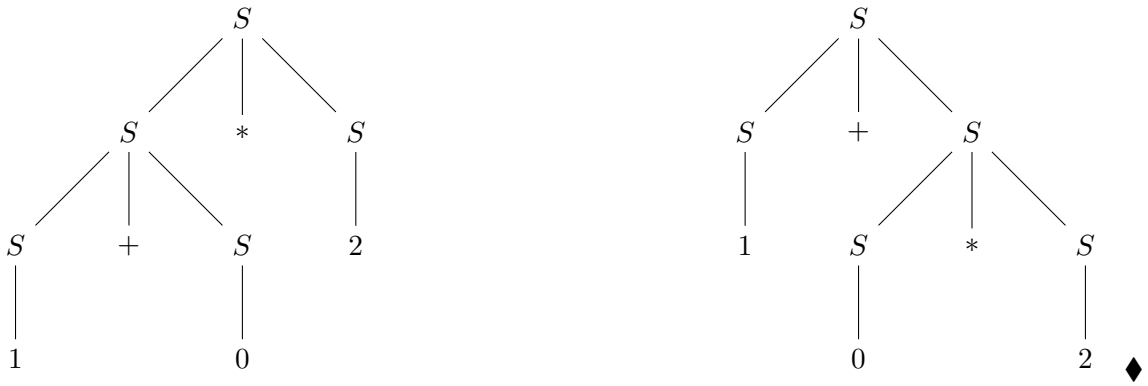
- de racine S,
- dont les feuilles sont étiquetées par des symboles terminaux,
- dont les nœuds intermédiaires sont étiquetés par des symboles non-terminaux,
- tel que si un nœud n étiqueté par un élément non-terminal $A \in V$ possède les fils $n_1, \ldots, n_k$ dans cet ordre alors $A \to n_1 \ldots n_k$ est une règle de G,
- et la concaténation des feuilles de l'arbre de gauche à droite donne le mot w. ⧫

Un arbre de dérivation explicite donc comment un mot est généré par la grammaire en partant du symbole initial S et en remplaçant successivement les symboles non-terminaux à l'aide d'une des règles de la grammaire. Il s'agit d'une description structurée de la dérivation d'un mot par une grammaire tel que défini précédemment (relation $\Rightarrow^*$).

Exemple 5.9 Prenons pour exemple la grammaire G_5 :

$$\begin{aligned} S &\longrightarrow S + S \\ S &\longrightarrow S \times S \\ S &\longrightarrow 0 \,|\, \cdots \,|\, 9 \end{aligned}$$

où $\{0, 1, \ldots, 9, +, \times\}$ est l'ensemble des symboles terminaux et S est l'unique symbole non-terminal. Cette grammaire génère des mots de la forme $1 + 0 \times 2$ constitués d'additions et de multiplications de chiffres. On montre ci-dessous les deux arbres de dérivation possible pour le mot $1 + 0 \times 2$ dans G_5.



⧫

L'une des tâches réalisées par un compilateur consiste à construire un arbre de dérivation associé au mot analysé. Cela pose donc problème lorsque l'arbre de dérivation n'est pas unique puisque le compilateur doit choisir quel arbre construire. Or ces arbres peuvent représenter des informations différentes.

Exemple 5.10 L'exemple 5.9 montre que le mot $1+0\times2$ admet deux arbres de dérivation dans la grammaire G_5. Remarquons que ces deux dérivations correspondent à deux interprétations différentes de l'expression $1 + 0 \times 2$:

- l'arbre de gauche correspond à $(1 + 0) \times 2$ qui donne donc la priorité à l'opérateur $+$ sur l'opérateur $\times$. Cette expression a la valeur 2 ;

— alors que l'arbre de droite correspond à $1 + (0 \times 2)$ qui donne la priorité à l'opérateur $\times$ comme le veut l'usage. Cette expression a la valeur 1. ⧫

Lorsqu'une telle situation se produit, on parle de grammaire ambiguë puisqu'il n'est pas possible de déterminer la structure du mot analysé.

Définition 5.7 Une grammaire $G = (V, \Sigma, R, S)$ est **ambiguë** s'il existe un mot $w \in \Sigma^*$ généré par G pour lequel il existe (au moins) deux arbres de dérivations distincts.

Une grammaire qui admet au plus un arbre de dérivation pour tout mot $w \in \Sigma^*$ est dîte **non-ambiguë**. ⧫

Il n'est pas possible de déterminer si une grammaire est ambiguë ou non. Plus précisément, le problème suivant est indécidable :

ENTREE : une grammaire hors-contexte G

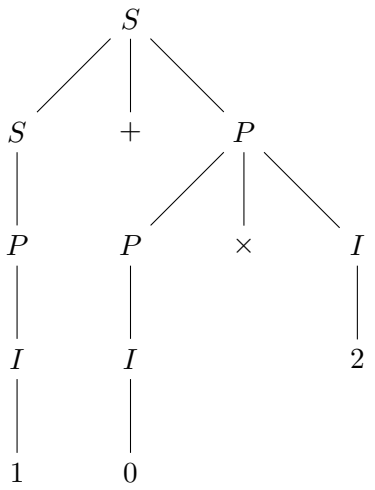
QUESTION : G est-elle ambiguë ?

C'est à dire qu'il n'existe pas (et il ne peut pas exister) d'algorithme qui résout ce problème. On peut décider l'ambiguïté de certaines sous-classes des grammaires hors-contexte (voir le cours de «Compilation » en 2ème année). On peut également parfois résoudre l'ambiguïté comme le montre l'exemple ci-dessous :

Exemple 5.11 La grammaire G_6 ci-dessous génère le même langage que G_5, mais elle est non-ambiguë :

$$\begin{aligned} S &\longrightarrow P \mid S + P \\ P &\longrightarrow I \mid P \times I \\ I &\longrightarrow 0 \mid \cdots \mid 9 \end{aligned}$$

Le mot $1 + 0 \times 2$ admet maintenant un seul arbre de dérivation :



On observe que l'ambiguïté a été résolue en ajoutant de nouveaux symboles non terminaux :

— Le fait que le non terminal S se transforme en P (mais pas l'inverse) donne priorité à l'opérateur $\times$ sur l'opérateur $+$. Ainsi $1 + 0 \times 2$ est interprétée comme $1 + (0 \times 2)$.

— La récursion gauche sur les symboles S (opérateur $+$) et P (opérateur $\times$) choisit l'associativité gauche plutôt que l'associativité droite. C'est à dire $0+1+3$ est interprétée comme $(0+1)+3$ plutôt que $0+(1+3)$. De même $0\times 1\times 3$ est lue comme $(0\times 1)\times 3$. Notons que les deux interprétations (associativité gauche ou droite) donnent le même résultat, mais elles correspondent à deux arbres de dérivation distincts qu'il faut bien distinguer. ♦

▷ Il n'est pas toujours aisé de transformer une grammaire ambiguë en grammaire non-ambiguë. De plus, comme le montre l'exemple précédent, cela complique la grammaire et la rend moins lisible.
L'outil YACC (étudié en TD de «Compilation» en 2ème année) propose un mécanisme pour résoudre les ambiguïtés. Pour la gramaire G_5 de l'exemple 5.9, on spécifierait :

```
%left '+'
%left '*'
```

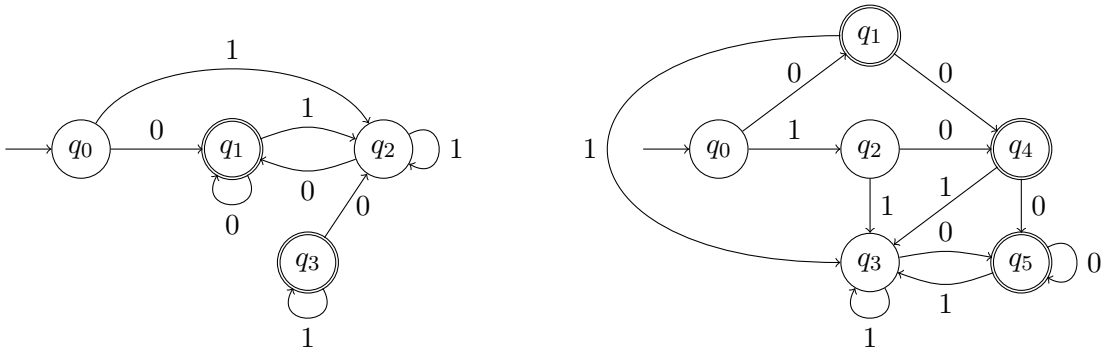
qui indique deux choses :
— choisir l'associativité à gauche pour les opérateurs $+$ et $\times$ (mot-clé `left`) ;
— l'opérateur $\times$ a la priorité sur l'opérateur $+$ (ordre des déclarations `left`).
À l'aide de cette information supplémentaire, YACC construit un analyseur syntaxique qui correspond à la grammaire G_6, à partir de la grammaire G_5.

Chapitre 6

Automate minimal et minimisation

Nous avons vu au chapitre 2 que tout langage régulier est reconnu par un automate fini déterministe. Cependant, il n'y a pas un unique automate fini (déterministe) qui reconnaît un langage régulier donné.

Exemple 6.1 Les automates finis de la figure 6.1 sont des exemples d'automates finis qui reconnaissent l'ensemble des encodages binaires des entiers naturels pairs, tout comme l'automate de la figure 1.1(a) page 10.



(a) A_3 (b) A_4

♦

FIGURE 6.1 – Deux AFD reconnaissant les encodages binaires des entiers naturels pairs.

Comment comparer deux automates finis ? Par exemple, comment décider que les deux AFD de la figure 6.1 reconnaissent le même langage ? Un langage est généralement reconnu par plusieurs automates finis (une infinité en fait). D'un point de vue pratique, il est pourtant important de choisir un «petit» automate, c'est à dire un algorithme avec aussi peu d'instructions que possible.

En section 6.1, nous montrons qu'il existe un **automate minimal** unique pour chaque langage régulier. Ceci va nous permettre de parler de **représentation canonique** de langages réguliers ou encore d'algorithmes minimaux pour les problèmes de décision «réguliers». Puis en section 6.2 nous présentons un algorithme qui calcule, à partir d'un automate fini déterministe A donné, l'automate minimal qui reconnaît $\mathcal{L}(A)$.

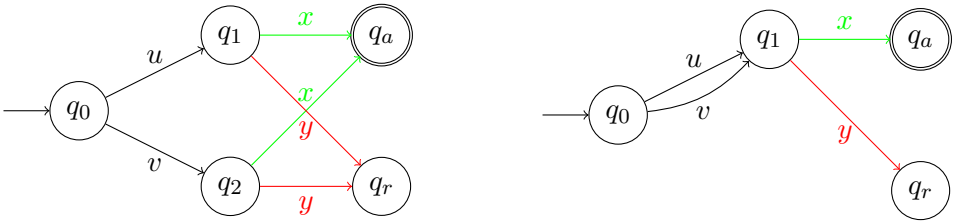


FIGURE 6.2 – Illustration de la congruence droite.

6.1 Représentation canonique des langages réguliers

6.1.1 Congruence associée à un langage

Nous montrons maintenant que l'existence d'une représentation canonique pour les langages réguliers est une propriété intrinsèque à ces langages.

Définition 6.1 (Congruence droite) Soit L un langage sur Σ. Deux mots u et v sont **congrus (à droite)** modulo L, noté $u \sim_L v$, si pour tout mot $x \in \Sigma^*$, $u \cdot x \in L$ si et seulement si $v \cdot x \in L$.
u et v sont dit **indistinguables** pour L. La congruence $\sim_L$ induit une relation d'équivalence. Nous notons $[w]_{\sim_L} = \{w' \mid w \sim_L w'\}$ la classe d'équivalence de w pour $\sim_L$. ⧫

Intuitivement, si u et v sont congrus modulo L, un automate fini qui reconnaît L n'a pas besoin de distinguer u et v puisque $u \cdot x$ et $v \cdot x$ conduisent au même verdict, pour tout mot x. Ceci est illustré sur l'automate de gauche en figure 6.2. On remarque que les langages reconnus depuis les états q_1 et q_2 sont les mêmes puisque $u \sim_L v$. On peut donc ne conserver que l'un des deux états q_1 et q_2 tout en reconnaissant le même langage, comme l'automate de droite en figure 6.2.

Inversement, si deux mots u et v ne sont pas congrus modulo L, alors il existe un mot x tel que $u \cdot x \in L$ et $v \cdot x \notin L$ (ou l'inverse). Ainsi, un automate fini qui reconnaît L, doit différencier u et v en arrivant dans deux états distincts après avoir lu u et v. Nous allons donc chercher à construire un automate fini dont le nombre d'états est le nombre de classes d'équivalence de $\sim_L$.

Définition 6.2 (Langage et co-langage d'un état) Soit $A = (Q, \Sigma, \delta, I, F)$ un automate fini. Le **langage de l'état** $q \in Q$ est défini par :

$$\mathcal{L}(A, q) = \{w \in \Sigma^* \mid q \xrightarrow{w} q' \quad \text{avec } q' \in F\}$$

Le **co-langage de l'état** $q \in Q$ est défini par :

$$\text{co}\mathcal{L}(A, q) = \{w \in \Sigma^* \mid \exists q_0 \in I,\ q_0 \xrightarrow{w} q\}$$ ⧫

▷ La définition du langage de q revient simplement à considérer q comme l'état initial de l'automate. Soit $A = (Q, \Sigma, \delta, I, F)$ un automate, et $q \in Q$ un état. Soit $A' = (Q, \Sigma, \delta, \{q\}, F)$. Nous avons $\mathcal{L}(A, q) = \mathcal{L}(A')$. Par ailleurs, $\mathcal{L}(A) = \bigcup_{q_0 \in I} \mathcal{L}(A, q_0)$.

Théorème 6.1 (Congruence droite et régularité) Un langage L est régulier si et seulement si la congruence $\sim_L$ est d'index fini. ⧫

Preuve. Nous allons montrer que L est reconnu par un automate fini si et seulement si les classes de congruence de L sont en nombre fini. Ce qui équivaut à l'énoncé du théorème 6.1 grâce au théorème 3.2 (page 33).

Supposons que L est reconnu par un automate fini déterministe A. Pour tout état $q \in Q$, $\mathsf{co}\mathcal{L}(A, q)$ est inclus dans une classe d'équivalence pour $\sim_L$. En effet, si $w, w' \in \mathsf{co}\mathcal{L}(A, q)$, alors, quel que soit $x \in \Sigma^*$, $w \cdot x \in L$ et $w' \cdot x \in L$ si et seulement si $x \in \mathcal{L}(A, q)$, donc $w \sim_L w'$. Il y a donc au plus autant de classes d'équivalence pour $\sim_L$ qu'il y a d'états dans A. Il vient alors que $\sim_L$ est d'index fini.

Si maintenant la congruence $\sim_L$ est d'index fini, nous définissons en section 6.1.2 un automate fini déterministe et complet qui reconnaît L (preuve au théorème 6.2). ■

6.1.2 Automate minimal

Nous nous intéressons maintenant à une notion d'automate minimal reconnaissant un langage régulier donné. Nous montrons que chaque état de cet automate correspond à une des classes d'équivalence de $\sim_L$. Ses transitions correspondent au passage d'une classe à l'autre : depuis la classe d'équivalence de w, la lecture d'un symbole s conduit à la classe d'équivalence de $w \cdot s$.

Définition 6.3 (Automate minimal) Soit L un langage régulier sur un alphabet Σ. L'**automate fini minimal** $\mathsf{afmin}(L) = (Q, \Sigma, \delta, q_0, F)$ qui reconnaît L est défini par :

- $Q = \big\{[w]_{\sim_L} \,|\, w \in \Sigma^*\big\}$.
- $\delta = \big\{([w]_{\sim_L}, s, [w \cdot s]_{\sim_L}) \,|\, s \in \Sigma,\ w \in \Sigma^*\big\}$.
- $q_0 = [\varepsilon]_{\sim_L}$.
- $F = \big\{[w]_{\sim_L} \,|\, w \in \Sigma^* \quad \text{et} \quad [w]_{\sim_L} \subseteq L\big\}$. ⧫

▷ Notons que $\mathsf{afmin}(L)$ est un automate déterministe et complet.

Nous prouvons maintenant la correction de $\mathsf{afmin}(L)$ au sens où il reconnaît L, et son unicité c'est à dire sa minimalité.

Théorème 6.2 Pour tout langage régulier L, $\mathsf{afmin}(L)$ est le plus petit automate fini déterministe et complet tel que $\mathcal{L}(\mathsf{afmin}(L)) = L$. ⧫

Preuve. Nous prouvons dans un premier temps que $\mathcal{L}(\mathsf{afmin}(L)) = L$. Soit $w \in L$ avec $w = w_1 \cdots w_n$, et $w_i \in \Sigma$ pour tout $i \in [1; n]$. Alors :

$$[\varepsilon]_{\sim_L} \xrightarrow{w_1} [w_1]_{\sim_L} \xrightarrow{w_2} [w_1 \cdot w_2]_{\sim_L} \xrightarrow{w_3} \cdots \xrightarrow{w_n} [w]_{\sim_L} \tag{6.1}$$

est une exécution de $\mathsf{afmin}(L)$. Par ailleurs, $w \in L$ donne $w \cdot \varepsilon \in L$, donc $w' \cdot \varepsilon \in L$ pour tout mot $w' \sim_L w$, et finalement $w' \in L$. Nous en déduisons que $[w]_{\sim_L} \subseteq L$, donc $[w]_{\sim_L} \in F$ et w est accepté par $\mathsf{afmin}(L)$.

Soit maintenant $w \in \mathcal{L}(\mathsf{afmin}(L))$ avec $w = w_1 \cdots w_n$, et $w_i \in \Sigma$ pour tout $i \in [1; n]$. Il existe donc une exécution acceptante de $\mathsf{afmin}(L)$ étiquetée w telle que représentée en équation (6.1). Puisque $[w]_{\sim_L} \in F$, il vient $[w]_{\sim_L} \subseteq L$, et en particulier $w \in L$.

Il reste donc à prouver que $\mathsf{afmin}(L) = (Q, \Sigma, \delta, q_0, F)$ est le plus petit automate fini déterministe et complet qui reconnaît L. Supposons qu'il existe un AFD complet $A' = (Q', \Sigma, \delta', q'_0, F')$ tel que $\mathcal{L}(A') = L$ et $\mathsf{Card}(Q') < \mathsf{Card}(Q)$. Supposons que pour tout couple de mots $w, w' \in \Sigma^*$ tels que $[w]_{\sim_L} \neq [w']_{\sim_L}$, dans A' : $q'_0 \xrightarrow{w} q$ et $q'_0 \xrightarrow{w'} q'$ avec $q \neq q'$. Alors, A' possède au moins autant d'états qu'il y a de classes d'équivalences pour $\sim_L$, ce qui contredit $\mathsf{Card}(Q') < \mathsf{Card}(Q)$. Il existe donc deux mots $w, w' \in \Sigma^*$ et un état $q \in Q$ tels que $[w]_{\sim_L} \neq [w']_{\sim_L}$ mais dans A' : $q'_0 \xrightarrow{w} q$ et $q'_0 \xrightarrow{w'} q$. Alors, pour tout mot $x \in \Sigma^*$, les mots $w \cdot x$ et $w' \cdot x$ sont soit tous les deux acceptés, soit tous les deux rejetés. Puisque $\mathcal{L}(A') = L$, il vient $w \sim_L w'$, ce qui contredit $[w]_{\sim_L} \neq [w']_{\sim_L}$. Nous en déduisons que A' n'existe pas, et donc $\mathsf{afmin}(L)$ est le plus petit automate fini déterministe et complet qui reconnaît L. ■

▷ La preuve précédente montre qu'il n'existe pas d'automate fini déterministe et complet avec moins d'états que afmin(L) qui reconnaît L. Cette contrainte sur le nombre d'états suffit à prouver l'unicité de afmin(L) (au renommage des états près) car étant déterministe et complet, tout autre relation de transition ne conduirait pas à reconnaître L.

6.2 Minimisation des automates finis

La définition 6.3 nous permet de prouver l'existence et l'unicité (théorème 6.2) de l'automate minimal qui reconnaît un langage L donné. Cependant, elle ne permet pas de construire $\mathsf{afmin}(L)$: il faudrait énumérer tous les mots de Σ^* pour connaître l'ensemble de ses états. Nous présentons maintenant un algorithme qui permet de calculer $\mathsf{afmin}(L)$ à partir d'un automate A connu qui reconnaît L.

Soit $A = (Q, \Sigma, \delta, q_0, F)$ un AFD supposé non minimal. Nous pouvons maintenant caractériser la non minimalité en considérant les langages reconnus depuis chacun des états de A. Deux états $q, q' \in Q$ sont dits **équivalents** si $\mathcal{L}(A, q) = \mathcal{L}(A, q')$. On note $\equiv_Q$ la relation d'équivalence ainsi définie.

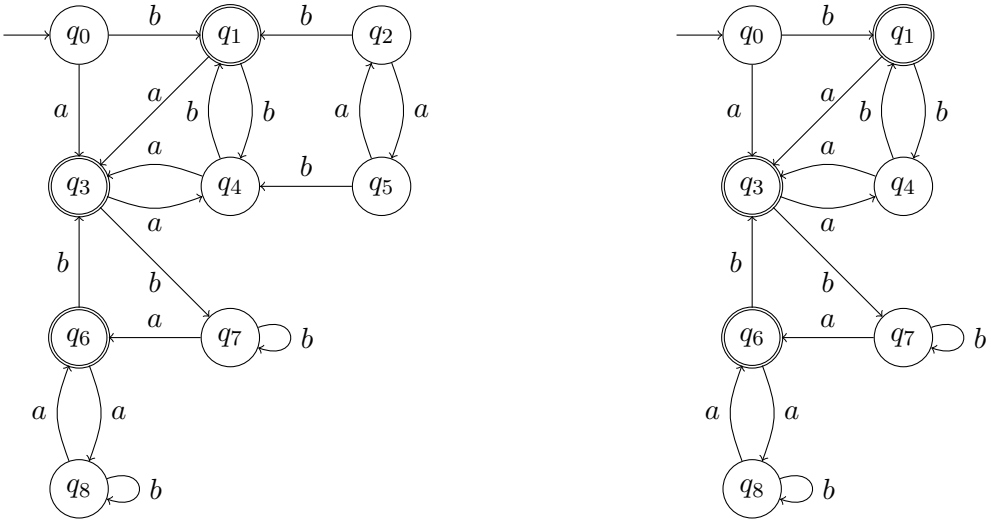
L'équivalence entre états doit être rapprochée de la congruence de $\mathcal{L}(A)$. En effet, supposons que $q_0 \xrightarrow{w} q$ et $q_0 \xrightarrow{w'} q'$ pour deux mots $w, w' \in \Sigma^*$. Alors, puisque $\mathcal{L}(A, q) = \mathcal{L}(A, q')$, pour tout mot $x \in \Sigma^*$, $w \cdot x$ est accepté par A si et seulement $w' \cdot x$ est accepté par A. Il vient alors $w \sim_{\mathcal{L}(A)} w'$. L'équivalence entre $\equiv_Q$ et $\sim_L$ est prouvée au théorème 6.5.

Un automate fini A n'est donc pas minimal si et seulement si :

— il possède (au moins) un état inaccessible : il existe $q \in Q$ tel que pour tout $w \in \Sigma^*$, $q_0 \not\xrightarrow{w} q$

— ou il possède deux états distincts $q, q' \in Q$ qui sont équivalents : $\mathcal{L}(A, q) = \mathcal{L}(A, q')$.

L'algorithme de minimisation d'un automate A procède donc en deux étapes : tout d'abord la suppression des états inaccessibles de A, puis la fusion de ses états équivalents.



(a) A_5 (b) $\mathsf{eqacc}(A_5)$

FIGURE 6.3 – Automate fini déterministe et sa partie accessible.

Exemple 6.2 L'automate A_3 de la figure 6.1(a) n'est pas minimal. En effet, l'état q_3 n'est pas accessible depuis q_0.

L'automate A_4 en figure 6.1(b) n'est lui non plus pas minimal. En effet, le langage reconnu depuis l'état q_4 est constitué de ε, de tous les mots $1 \cdot w$ tels que w est accepté depuis q_3 et de tous les mots $0 \cdot w'$ tels que $w' \in \mathcal{L}(A_4, q_5)$. D'autre part le langage reconnu depuis l'état q_5 est constitué du mot vide ε, de tous les mots $1 \cdot w$ tels que $w \in \mathcal{L}(A_4, q_3)$ et de tous les mots $0 \cdot w'$ tels que $w' \in \mathcal{L}(A_4, q_5)$. Nous avons donc $\mathcal{L}(A_4, q_4) = \mathcal{L}(A_4, q_5)$.

Par contre, l'automate A_1 en figure 1.1(a) (page 10) est l'automate minimal qui reconnaît le langage des nombres naturels pairs encodés en binaire. Les automates A_1, A_4 et A_5 reconnaissent le même langage. L'application de l'algorithme de minimisation présenté ci-après aux automates A_3 et A_4 fournit l'automate A_1 en résultat. ♦

6.2.1 Élimination des états inaccessibles

Définition 6.4 (États accessibles) L'ensemble des **états accessibles** d'un automate fini $A = (Q, \Sigma, \delta, I, F)$ est défini par :

$$\mathsf{access}(A) = \{q \in Q \mid \exists q_0 \in I.\ \exists w \in \Sigma^*.\ q_0 \xrightarrow{w} q\}$$ ♦

Intuitivement, les états accessibles d'un automate fini $A = (Q, \Sigma, \delta, I, F)$ sont les états q pour lesquels il existe une séquence de transitions partant d'un état initial et menant jusqu'à q.

Exemple 6.3 Prenons l'exemple de l'automate A_3 en figure 6.1(a). L'état q_0 est trivialement accessible. L'état q_2 est lui aussi accessible, notamment par $q_0 \xrightarrow{1} q_2$. Mais l'état q_3 n'est pas accessible. Tous les états de l'automate A_4 en figure 6.1(b) sont accessibles.

Considérons maintenant l'automate A_5 en figure 6.3(a). Les états q_0, q_1, q_3, q_4, q_6, q_7 et q_8 sont accessibles. Mais les états q_2 et q_5 ne le sont pas. ♦

Définition 6.5 (Partie accessible) La restriction d'un automate fini $A = (Q, \Sigma, \delta, I, F)$ à sa **partie accessible** est l'automate fini $\mathsf{eqacc}(A) = (Q', \Sigma, \delta', I, F')$ défini par :
— $Q' = \mathsf{access}(A)$,
— $\delta' = \delta \cap (Q' \times \Sigma \times Q')$,
— et $F' = F \cap Q'$. ♦

▷ Remarquons que si A est déterministe (resp. complet) alors $\mathsf{eqacc}(A)$ est déterministe (resp. complet).

Exemple 6.4 La restriction de l'automate A_3 en figure 6.1(a) à sa partie accessible, $\mathsf{eqacc}(A_3)$, revient à supprimer l'état q_3 et ses transitions.

La partie accessible $\mathsf{eqacc}(A_5)$ de l'automate A_5 en figure 6.3(a) est représentée en figure 6.3(b). ♦

La suppression des états inaccessibles d'un automate fini ne modifie par le langage reconnu par celui-ci. Cela vient du fait que, par définition, aucun état inaccessible ne peut être rencontré sur une exécution (acceptante) de l'automate.

Théorème 6.3 Pour tout automate fini A, $\mathcal{L}(\mathsf{eqacc}(A)) = \mathcal{L}(A)$. ♦

Preuve. Toute exécution de $\mathsf{eqacc}(A)$ est une exécution de A (par définition de δ') et $F' \subseteq F$ puisque $\mathsf{access}(A) \subseteq Q$. Donc tout mot $w \in \mathcal{L}(\mathsf{eqacc}(A))$ est accepté par A.

Inversement, soit $w = w_1 \cdots w_n$ avec $w_i \in \Sigma$ pour tout $i \in [1; n]$ un mot tel qu'il existe une exécution acceptante de A :

$$q_0 \xrightarrow{w_1} q_1 \xrightarrow{w_2} \cdots \xrightarrow{w_n} q_n \qquad q_0 \in I \text{ et } q_n \in F$$

Par la définition 6.4, $q_0, q_1, \ldots, q_n \in \mathsf{access}(A)$. Cette exécution est donc une exécution acceptante de $\mathsf{eqacc}(A)$, et $w \in \mathcal{L}(\mathsf{eqacc}(A))$. ■

Le calcul de $\mathsf{access}(A)$ correspond au plus petit point fixe de la fonction $\mathrm{Acc} : X \to X \cup \{q' \in Q \mid \exists q \in X. \exists s \in \Sigma. (q, s, q') \in \delta\}$ contenant les états initiaux I. On peut le calculer par itérations successives :

$$\begin{aligned} &\mathrm{Acc}(0) = I \\ &\mathrm{Acc}(n+1) = \mathrm{Acc}(n) \cup \{q' \in Q \mid \exists q \in \mathrm{Acc}(n). \exists s \in \Sigma. (q, s, q') \in \delta\} \end{aligned}$$

Intuitivement, $\mathrm{Acc}(n)$ est l'ensemble des états pour lesquels il existe une séquence d'au plus n transitions menant d'un état initial jusqu'à eux. Nous calculons donc successivement $\mathrm{Acc}(0)$, puis $\mathrm{Acc}(1)$ en utilisant $\mathrm{Acc}(0)$, puis $\mathrm{Acc}(2)$ à partir de $\mathrm{Acc}(1)$, et ainsi de suite jusqu'au point fixe obtenu lorsque $\mathrm{Acc}(n) = \mathrm{Acc}(n-1)$ pour une valeur donnée de n. L'algorithme EtatsAccessibles de la figure 6.4 calcule le plus petit point fixe de Acc.

```
1   ENTREE: Un automate fini A = (Q, Σ, δ, I, F)
2   SORTIE: access(A) l'ensemble des états accessibles de A
3
4   EtatsAccessibles(A) :
5   debut
6     Acc ← I;   Acc' ← ∅
7     tantque (Acc ≠ Acc') faire
8       Acc' ← Acc
9       Acc ← Acc' ∪ {q' ∈ Q | ∃q ∈ Acc'. ∃s ∈ Σ. (q, s, q') ∈ δ}
10    fintantque
11
12    retourner Acc
13  fin
```

FIGURE 6.4 – Algorithme de calcul des états accessibles.

▷ Remarquons la similitude entre les algorithmes 6.4 et 2.6 (page 24). Ce dernier est un calcul d'accessibilité pour lequel nous ne considérons que les transitions d'étiquette ε.

Exemple 6.5 Pour l'automate A_3 en figure 6.1(a), nous avons les valeurs successives suivantes de Acc et Acc$'$ lors de l'exécution de l'algorithme 6.4 :

Itération	Acc	Acc$'$
0	$\{q_0\}$	$\emptyset$
1	$\{q_0, q_1, q_2\}$	$\{q_0\}$
2	$\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$

Nous obtenons $\mathsf{access}(A_3) = \{q_0, q_1, q_2\}$, donc l'état q_3 n'est pas accessible.

Considérons maintenant l'automate A_5 en figure 6.3(a). Les valeurs successives de Acc et Acc$'$ pour le calcul de $\mathsf{access}(A_5)$ par l'algorithme 6.4 sont :

Itération	Acc	Acc$'$
0	$\{q_0\}$	$\emptyset$
1	$\{q_0, q_1, q_3\}$	$\{q_0\}$
2	$\{q_0, q_1, q_3, q_4, q_7\}$	$\{q_0, q_1, q_3\}$
3	$\{q_0, q_1, q_3, q_4, q_7, q_6\}$	$\{q_0, q_1, q_3, q_4, q_7\}$
4	$\{q_0, q_1, q_3, q_4, q_7, q_6, q_8\}$	$\{q_0, q_1, q_3, q_4, q_7, q_6\}$
5	$\{q_0, q_1, q_3, q_4, q_7, q_6, q_8\}$	$\{q_0, q_1, q_3, q_4, q_7, q_6, q_8\}$

Nous obtenons donc $\mathsf{access}(A_5) = \{q_0, q_1, q_3, q_4, q_6, q_7, q_8\}$: les états q_2 et q_5 de A_5 sont inaccessibles. ♦

Théorème 6.4 Pour tout automate $A = (Q, \Sigma, \delta, I, F)$, l'algorithme 6.4 EtatsAccessibles calcule $\mathsf{access}(A)$ en temps $\mathcal{O}(\mathsf{Card}(Q))$. ♦

Preuve. L'algorithme 6.4 termine. En effet, la suite des valeurs de Acc est strictement croissante, Acc $\subseteq Q$ est invariant, et Q est fini.

L'algorithme 6.4 est correct (i.e il calcule $\mathsf{access}(A)$) par l'invariance de :

1. «tous les états dans Acc sont accessibles»,
2. et «Acc contient l'ensemble des successeurs des états dans Acc$'$».

L'invariant 1 est vrai en ligne 6. Par ailleurs, s'il est vrai en ligne 7, il l'est aussi en ligne 9 puisque nous ne faisons qu'ajouter à Acc des états accessibles. L'invariant 2 est vrai en ligne 6 puisque q_0 est trivialement accessible. Par ailleurs, en ligne 9 nous ajoutons à Acc l'ensemble des successeurs des états de Acc$'$. Sachant qu'à la sortie de la boucle Acc = Acc$'$, nous en déduisons que Acc ne contient que des états accessibles et il est clos par δ, donc il est égal à $\mathsf{access}(A)$.

Enfin, dans le pire des cas où tous les états de A sont accessibles, et ou un seul état est ajouté à Acc (ligne 9) lors de chaque itération, il faut $\mathsf{Card}(Q) - 1$ itérations de la boucle pour calculer $\mathsf{access}(A)$. Donc $\mathcal{O}(\mathsf{Card}(Q))$ opérations si les manipulations (ajout, union, égalité) des ensembles Acc et Acc$'$ sont en temps constant. ■

6.2.2 Fusion des états équivalents

Nous considérons maintenant des automates finis déterministes et complets, sans états inaccessibles. Nous avons vu au chapitre 2 comment déterminiser tout AFN. Tout automate peut être rendu complet par la procédure de complétude vue en section 1.2.2 page 14. Enfin, la section précédente décrit comment calculer l'ensemble des états accessibles d'un automate fini, puis comment le restreindre à sa partie accessible.

Il nous reste maintenant à fusionner les états équivalents de l'automate A, c'est à dire les états q et q' tels que $\mathcal{L}(A, q) = \mathcal{L}(A, q')$. Pour cela, il faut donc calculer quels états de A reconnaissent le même langage, c'est à dire la relation $\equiv_Q$.

Exemple 6.6 Considérons l'automate A_4 en figure 6.1(b) (page 59). Tout mot accepté depuis q_0 est soit un mot de la forme $0 \cdot w$ tel que $w \in \mathcal{L}(A_4, q_1)$, soit un mot de la forme $1 \cdot w'$ tel que $w' \in \mathcal{L}(A_4, q_2)$. Nous pouvons donc écrire les équations définissant le langage reconnus par chaque état de A_4 :

$$\begin{aligned}
\mathcal{L}(A_4, q_0) &= 0 \cdot \mathcal{L}(A_4, q_1) \cup 1 \cdot \mathcal{L}(A_4, q_2) \\
\mathcal{L}(A_4, q_1) &= \varepsilon \cup 0 \cdot \mathcal{L}(A_4, q_4) \cup 1 \cdot \mathcal{L}(A_4, q_3) \\
\mathcal{L}(A_4, q_2) &= 0 \cdot \mathcal{L}(A_4, q_4) \cup 1 \cdot \mathcal{L}(A_4, q_3) \\
\mathcal{L}(A_4, q_3) &= 0 \cdot \mathcal{L}(A_4, q_5) \cup 1 \cdot \mathcal{L}(A_4, q_3) \\
\mathcal{L}(A_4, q_4) &= \varepsilon \cup 0 \cdot \mathcal{L}(A_4, q_5) \cup 1 \cdot \mathcal{L}(A_4, q_3) \\
\mathcal{L}(A_4, q_5) &= \varepsilon \cup 0 \cdot \mathcal{L}(A_4, q_5) \cup 1 \cdot \mathcal{L}(A_4, q_3)
\end{aligned}$$

▷ Les accolades sur les ensembles singletons $\{\varepsilon\}$, $\{0\}$ et $\{1\}$ ont été omises afin de rendre les équations plus lisibles.

D'après les équations précédentes, il vient immédiatement $\mathcal{L}(A_4, q_4) = \mathcal{L}(A_4, q_5)$. Par substitution, nous obtenons alors :

$$\mathcal{L}(A_4, q_4) = \mathcal{L}(A_4, q_5) = \mathcal{L}(A_4, q_1) \qquad \text{et} \qquad \mathcal{L}(A_4, q_2) = \mathcal{L}(A_4, q_3) = \mathcal{L}(A_4, q_0)$$

Pour l'automate A_4, les classes d'équivalence de $\equiv_Q$ sont donc : $\{q_0, q_2, q_3\}$ d'autre part, et $\{q_1, q_4, q_5\}$ d'autre part. ⧫

Une fois connue la relation $\equiv_Q$, nous pouvons définir l'automate fini minimal qui reconnaît le même langage que l'automate A donné.

Définition 6.6 (Automate minimal équivalent) Soit $A = (Q, \Sigma, \delta, q_0, F)$ un automate fini déterministe, complet et sans états inaccessibles. L'**automate minimal équivalent** à A est $\mathsf{eqmin}(A) = (Q', \Sigma, \delta', q'_0, F')$ défini par :
— $Q' = \{[q]_{\equiv_Q} \mid q \in Q\}$,
— $\delta' = \{([q]_{\equiv_Q}, s, [q']_{\equiv_Q}) \mid (q, s, q') \in \delta\}$,
— $q'_0 = [q_0]_{\equiv_Q}$,
— et $F' = \{[q]_{\equiv_Q} \mid [q]_{\equiv_Q} \subseteq F\}$. ⧫

Exemple 6.7 L'automate minimal équivalent à A_4 en figure 6.1(b) est l'automate A_1 en figure 1.1(a) (page 10). ⧫

Nous montrons maintenant que l'automate minimal équivalent à A correspond à $\mathsf{afmin}(\mathcal{L}(A))$, l'automate minimal qui reconnaît le langage reconnu par A.

Théorème 6.5 Pour tout automate fini $A = (Q, \Sigma, \delta, q_0, F)$ déterministe, complet et sans états inaccessibles, $\mathsf{eqmin}(A) = \mathsf{afmin}(\mathcal{L}(A))$. ⧫

Preuve. Nous prouvons dans un premier temps que $\equiv_Q$ et $\sim_{\mathcal{L}(A)}$ coïncident. Formellement, nous prouvons que pour toute classe d'équivalence $[q]_{\equiv_Q}$, $\mathsf{co}\mathcal{L}([q]_{\equiv_Q})$ est une classe d'équivalence pour $\sim_{\mathcal{L}(A)}$ et inversement. Notons que pour tout ensemble d'états $X \subseteq Q$, $\mathsf{co}\mathcal{L}(X) = \bigcup_{q \in X} \mathsf{co}\mathcal{L}(q)$.

Soit $q \in Q$ un état de A, soit $w, w' \in \mathsf{co}\mathcal{L}([q]_{\equiv_Q})$, et soit $q_0 \xrightarrow{w} q$ et $q_0 \xrightarrow{w'} q'$ les exécutions correspondantes dans A. Si $q = q'$, trivialement $w \sim_{\mathcal{L}(A)} w'$. Si $q \neq q'$, alors puisque $q \equiv_Q q'$, $\mathcal{L}(A, q) = \mathcal{L}(A, q')$. Donc pour tout mot $x \in \Sigma^*$, $w \cdot x \in \mathcal{L}(A)$ si et seulement si $w' \cdot x \in \mathcal{L}(A)$, et $w \sim_{\mathcal{L}(A)} w'$.

Considérons maintenant deux mots w, w' tels que $w \sim_{\mathcal{L}(A)} w'$. Soient $q_0 \xrightarrow{w} q$ et $q_0 \xrightarrow{w'} q'$ les exécutions correspondantes de A. Pour tout mot $x \in \Sigma^*$, $w \cdot x \in \mathcal{L}(A)$ si et seulement si $w' \cdot x \in \mathcal{L}(A)$. Nous en déduisons que $x \in \mathcal{L}(A, q)$ si et seulement si $x \in \mathcal{L}(A, q')$, donc $q \equiv_Q q'$.

Il suffit alors de remplacer les classes d'équivalence de $\equiv_Q$ en définition 6.6 par les classes d'équivalence de $\sim_{\mathcal{L}(A)}$ en définition 6.3 (ou inversement) pour constater que les automates $\mathsf{eqmin}(A)$ et $\mathsf{afmin}(\mathcal{L}(A))$ sont égaux (à isomorphisme près). ■

Les relations $\sim_L$ et $\equiv_Q$ sont équivalentes pour $A = (Q, \Sigma, \delta, q_0, F)$ tel que $\mathcal{L}(A) = L$. Mais en pratique, pour calculer $\sim_L$ ou $\equiv_Q$, il est nécessaire de travailler sur une structure finie. L est en général infini, mais nous pouvons travailler soit sur une expression régulière de langage L, soit sur un automate fini qui reconnaît L. Nous présentons donc en figure 6.5 l'algorithme $\mathsf{Partition}_{\equiv_Q}$ de calcul de $\equiv_Q$ pour un automate fini déterministe complet et sans états inaccessibles $A = (Q, \Sigma, \delta, q_0, F)$. L'algorithme $\mathsf{Partition}_{\equiv_Q}$ maintient un ensemble P de couples d'états (q, q') tels que :
— si $(q, q') \notin P$, alors $q \not\equiv_Q q'$;
— et si $(q, q') \in P$, on fait l'**hypothèse** que $q \equiv_Q q'$.

```
1  ENTREE: A = (Q, Σ, δ, q0, F) AFD complet sans états inaccessibles
2  SORTIE: P telle que (q, q') ∈ P si et seulement si q ≡_Q q'
3
4  Partition_≡Q(A) :
5  debut
6    P' ← Q × Q ; P ← Q × Q
7    pour tout q ∈ F et q' ∉ F faire
8      retirer (q, q') et (q', q) de P
9    tantque (P ≠ P') faire
10     P' ← P
11     pour tout (q, q') ∈ P faire
12       pour tout s ∈ Σ faire
13         si (succ(q, s), succ(q', s)) ∉ P' alors
14           retirer (q, q') et (q', q) de P
15         finsi
16       finpour
17     finpour
18   fintantque
19   retourner P
20 fin
```

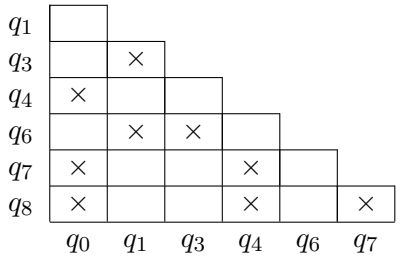
FIGURE 6.5 – Algorithme de calcul de $\equiv_Q$.

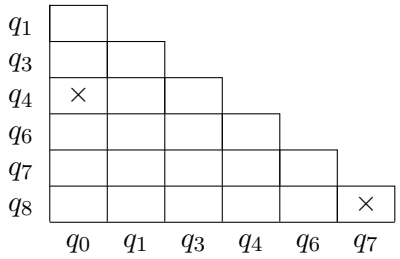
Le principe de l'algorithme est de faire tendre la relation P vers $\equiv_Q$ par raffinements successifs en retirant les couples (q, q') de P dès que l'hypothèse $q \equiv_Q q'$ est invalidée.

1. $\mathsf{Partition}_{\equiv_Q}$ fait l'hypothèse initiale que tous les états sont équivalents pour $\equiv_Q$ en ligne 6 ;
2. nous savons que pour tous états $q, q' \in Q$ tels que $q \in F$ et $q' \notin F$, $q \not\equiv_Q q'$ puisque $\varepsilon \in \mathcal{L}(A, q)$ alors que $\varepsilon \notin \mathcal{L}(A, q')$. La boucle en ligne 7 retire donc de P toutes les paires d'états distinguables selon ce critère ;
3. enfin, supposons que $(q, q') \in P$ (*i.e.* hypothèse $q \equiv_Q q'$) et qu'il existe un symbole $s \in \Sigma$ et deux états $p, p' \in Q$ tels que $q \xrightarrow{s} p$ et $q' \xrightarrow{s} p'$ avec $(p, p') \notin P$. Alors, nous avons la certitude $q \not\equiv_Q q'$ puisque $\mathcal{L}(A, p) \neq \mathcal{L}(A, p')$ implique que $s \cdot \mathcal{L}(A, p) \neq s \cdot \mathcal{L}(A, p')$ et A est déterministe. De tels couples (q, q') sont détectés et éliminés de P en lignes 9 à 18.

Le point fixe est atteint lorsque $P = P'$ (ligne 9), P' conservant la valeur de P au raffinement précédent (ligne 10), donc lorsque P ne contient plus que des couples (q, q') tels que $q \equiv_Q q'$.

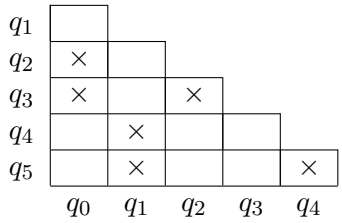
Exemple 6.8 La relation P de l'algorithme $\mathsf{Partition}_{\equiv_Q}$ peut être représentée comme une matrice symétrique de taille $\mathsf{Card}(Q) \times \mathsf{Card}(Q)$. Dans cet exemple, nous ne représentons que la partie triangulaire inférieure de cette matrice. Il y a une croix en ligne q_i, colonne q_j si $(q_i, q_j) \in P$, et une case vide sinon. Appliquons l'algorithme 6.5 $\mathsf{Partition}_{\equiv_Q}$ à l'automate $\mathsf{eqacc}(A_5)$ en figure 6.3(b).





La matrice ci-contre représente P en ligne 9 de l'algorithme 6.5. Les états $\{q_0, q_4, q_7, q_8\}$ ne sont pas équivalents aux états $\{q_1, q_3, q_6\}$ puisque seuls les seconds sont accepteurs. L'hypothèse de départ est que q_0, q_4, q_7 et q_8 sont équivalents d'une part, et que q_1, q_3 et q_6 sont équivalents entre eux d'autre part.
En ligne 9 de l'algorithme 6.5, P est donc la matrice ci-dessus. Pour le couple d'états (q_0, q_7), le symbole a, donne $q_0 \xrightarrow{a} q_3$, $q_7 \xrightarrow{a} q_6$ et $(q_3, q_6) \in P$, et pour le symbole b, $q_0 \xrightarrow{b} q_1$, $q_7 \xrightarrow{b} q_7$, or $(q_1, q_7) \notin P$. q_0 et q_7 ne sont donc pas équivalents, et l'algorithme 6.5 retire (q_0, q_7) de P en ligne 13. La matrice ci-contre présente P après une exécution des lignes 9 à 18 qui est également le point fixe de l'algorithme, c'est à dire $\equiv_Q$.

La matrice ci-dessous représente la relation P pour l'automate A_4 en figure 6.1(b) obtenue en ligne 9 de l'algorithme 6.5. Il s'avère que l'exécution des lignes 9 à 18 laisse P invariante, et qu'il s'agit donc de $\equiv_Q$.



♦

Nous prouvons maintenant que l'algorithme $\mathsf{Partition}_{\equiv_Q}$ termine et qu'il est correct.

Théorème 6.6 Pour tout AFD complet sans états inaccessibles $A = (Q, \Sigma, \delta, q_0, F)$, l'algorithme 6.5 $\mathsf{Partition}_{\equiv_Q}$ calcule $\equiv_Q$ en temps $\mathcal{O}(\mathsf{Card}(\Sigma) \times \mathsf{Card}(Q)^4)$. ♦

Preuve. L'algorithme $\mathsf{Partition}_{\equiv_Q}$ termine puisque P est un ensemble fini qui décroît strictement par exécution de la boucle des lignes 9 à 18.

Nous prouvons la correction de $\mathsf{Partition}_{equivalA}$ grâce à l'invariance des propriétés suivantes :

1. si $(q, q') \notin P$ alors $q \not\equiv_Q q'$;
2. P est l'ensemble des couples $(q, q') \in Q \times Q$ tels que pour tout symbole $s \in \Sigma$, $(\mathrm{succ}(q, s), \mathrm{succ}(q', s)) \in P'$;
3. et si $(q, q') \in P$ alors q et q' sont soient tous les deux acceptants (i.e. $q, q' \in F$), soient tous les deux non acceptants (i.e. $q, q' \notin F$)

Les propriétés 1 et 3 sont vraies en ligne 9 puisque si $(q, q') \notin P$, alors $q \in F$ et $q' \notin F$ ou inversement. La propriété 2 est également vraie puisque $P' = Q \times Q$.

Supposons maintenant que les propriétés 1, 2 et 3 sont vraies en ligne 9 et prouvons qu'elles sont alors vraies en ligne 18. En ligne 18, si $(q, q') \notin P$ soit $(q, q') \notin P'$, soit $(q, q') \in P'$ mais

il existe $s \in \Sigma$ tel que $(\text{succ}(q,s), \text{succ}(q',s)) \notin P'$. Dans ce dernier cas, par induction sur la propriété 1, $\text{succ}(q,s) \not\equiv_Q \text{succ}(q',s)$, et donc $q \not\equiv_Q q'$. Dans les deux cas la propriété 1 est invariante. Considérons maintenant la propriété 2. Si $(q,q') \in P$ en ligne 18, alors pour tout symbole $s \in \Sigma$, $(\text{succ}(q,s), \text{succ}(q',s)) \in P'$ par la ligne 12. La propriété 2 est donc elle aussi invariante. Enfin, si $(q,q') \notin P$ en ligne 9, alors $(q,q') \notin P$ en ligne 18 puisque les instructions de la boucle ne font que retirer des couples d'états de P. Ainsi, la propriété 3 est également invariante.

À la sortie de la boucle en ligne 19, les propriétés 1 2 et 3 étant vraies ainsi que $P = P'$, nous concluons que P est égale à $\equiv_Q$.

La boucle en ligne 7 est itérée $\mathsf{Card}(F) \times \mathsf{Card}(Q) = \mathcal{O}(\mathsf{Card}(Q)^2)$ fois. Dans le pire cas, chaque itération de la boucle en ligne 9 ne retire qu'un couple (q,q') de P et il est nécessaire de tous les retirer pour obtenir $\equiv_Q$. La complexité cumulée de la boucle en ligne 11 est donc $\mathsf{Card}(Q)^2 + (\mathsf{Card}(Q)^2 - 1) + \cdots + 1 = \mathcal{O}(\mathsf{Card}(Q)^4)$ sachant que la taille maximale de P à la première entrée dans la boucle en ligne 9 est $\mathcal{O}(\mathsf{Card}(Q)^2)$. Chaque itération de la boucle en ligne 11 nécessite $\mathcal{O}(\mathsf{Card}(\Sigma))$ opérations pour considérer chaque symbole de l'alphabet (ligne 12). Au total, l'algorithme 6.5 $\mathsf{Partition}_{\equiv_Q}$ a donc une complexité en $\mathcal{O}(\mathsf{Card}(\Sigma) \times \mathsf{Card}(Q)^4)$ puisque la complexité de la boucle en ligne 7 en $\mathcal{O}(\mathsf{Card}(Q)^2)$ s'efface. ■

Nous pouvons donc conclure sur le problème posé en début de ce chapitre : comment décider si deux automates finis reconnaissent le même langage ?

Théorème 6.7 Soient A_1 et A_2 deux AFD complets sans états inaccessibles. $\mathcal{L}(A_1) = \mathcal{L}(A_2)$ si et seulement si $\mathsf{eqmin}(A_1) = \mathsf{eqmin}(A_2)$. ♦

Ce résultat se généralise à deux automates finis quelconques. En effet, il est toujours possible de les rendre déterministes (voir chapitre 2) et complets (section 1.2.2 page 14). Enfin, il est possible d'éliminer les états inaccessibles. La section suivante décrit la procédure complète de minimisation.

6.2.3 Algorithme de minimisation complet

La figure 6.6 présente l'algorithme Minimisation qui calcule, pour un automate fini déterministe et complet A donné, l'automate minimal $\mathsf{eqmin}(A)$.

Exemple 6.9 L'application de l'algorithme Minimisation aux automates A_3 (figure 6.1(a)) et A_4 (figure 6.1(b)) produit l'automate A_1 (figure 1.1(a) page 10). Nous avons donc $A_1 = \mathsf{eqmin}(A_3)$ et $A_1 = \mathsf{eqmin}(A_4)$, et nous pouvons en conclure que $\mathcal{L}(A_1) = \mathcal{L}(A_3) = \mathcal{L}(A_4)$.

L'automate minimal équivalent à A_5 et $\mathsf{eqacc}(A_5)$ de la figure 6.3 est représenté en figure 6.7. Les noms des états représentent les états de A_5 et $\mathsf{eqacc}(A_5)$ qui ont été fusionnés.♦

Nous prouvons maintenant que l'algorithme Minimisation termine, qu'il est correct, et nous donnons une estimation de sa complexité.

Théorème 6.8 Pour tout automate fini déterministe et complet $A = (Q, \Sigma, \delta, q_0, F)$, l'algorithme 6.6 Minimisation calcule $\mathsf{afmin}(\mathcal{L}(A))$ en temps $\mathcal{O}(\mathsf{Card}(\Sigma) \times \mathsf{Card}(Q)^4)$. ♦

Preuve. La terminaison des algorithmes EtatsAccessibles et $\mathsf{Partition}_{\equiv_Q}$ établie aux théorèmes 6.4 et 6.6 respectivement suffit à conclure que l'algorithme Minimisation termine.

```
1  ENTREE: A = (Q, Σ, δ, q0, F) un AFD complet
2  SORTIE: afmin(L(A)) l'automate minimal qui accepte L(A)
3
4  Minimisation(A) :
5  debut
6     // Elimination des états inaccessibles
7     Acc ← EtatsAccessibles(A)
8     A' ← (Acc, Σ, δ ∩ (Acc × Σ × Acc), q0, F ∩ Acc)
9     // Fusion des états équivalents, rappel : P est égale à ≡_Acc
10    P ← Partition_{≡_Acc}(A')
11    Q'' ← {[q]_{≡_P} | q ∈ Acc}
12    δ'' ← {(X, s, X') ∈ Q'' × Σ × Q'' | X' = succ(X, s)}
13    q0'' ← [q0]_{≡_P}
14    F'' ← {X ∈ Q'' | X ⊆ F}
15    A'' ← (Q'', Σ, δ'', q0'', F'')
16    retourner A''
17 fin
```

FIGURE 6.6 – Algorithme de minimisation.

Nous savons par le théorème 6.4 que l'algorithme EtatsAccessibles est correct. Nous en déduisons alors que $A' = \mathsf{eqacc}(A)$ par similarité de la ligne 8 et de la définition 6.5. Nous avons donc $\mathcal{L}(A') = \mathcal{L}(A)$ et A' est un AFD complet sans états inaccessibles. Il est donc valide d'appliquer l'algorithme $\mathsf{Partition}_{\equiv_Q}$ à A' en ligne 10. Par le théorème 6.6, cet algorithme nous retourne $\equiv_{Acc}$. Les lignes 12 à 15 correspondent alors à la définition 6.6. Il vient donc que l'algorithme retourne $\mathsf{eqmin}(A')$, c'est à dire $\mathsf{afmin}(\mathcal{L}(A))$ par le théorème 6.5.

Enfin, la complexité est établie en tenant compte de la complexité de chacune des opérations :

- ligne 7 : $\mathcal{O}(\mathsf{Card}(Q))$ par le théorème 6.4,
- ligne 8 : $\mathcal{O}(\mathsf{Card}(Q) \times \mathsf{Card}(\Sigma))$ au pire car il faut parcourir chaque transition de δ et chaque état de F pour en calculer la restriction,
- ligne 10 : $\mathcal{O}(\mathsf{Card}(\Sigma) \times \mathsf{Card}(Q)^4)$ par le théorème 6.6,
- ligne 11 : $\mathcal{O}(\mathsf{Card}(Q))$ puisqu'il suffit de parcourir une fois l'ensemble des état Q pour les regrouper en classes d'équivalence pour $\equiv_{Acc}$,
- ligne 12 : $\mathcal{O}(\mathsf{Card}(Q) \times \mathsf{Card}(\Sigma))$ puisque, $\equiv_{Acc}$ réalisant une partition de Q, succ sera calculée exactement une fois par état de Q et par symbole de Σ,
- ligne 13 : $\mathcal{O}(1)$ puisqu'il suffit d'une recherche dans Q'',
- ligne 14 : $\mathcal{O}(Q)$ puisqu'il suffit de parcourir Q'' et de tester pour un seul état q de chaque classe d'équivalence $[q]_{\equiv_{Acc}}$ si $q \in F$, or $\mathsf{Card}(Q'') = \mathcal{O}(\mathsf{Card}(Q))$

La complexité de l'algorithme Minimisation est égale à la somme de ces complexités qui est dominée par $\mathcal{O}(\mathsf{Card}(\Sigma) \times \mathsf{Card}(Q)^4)$. ■

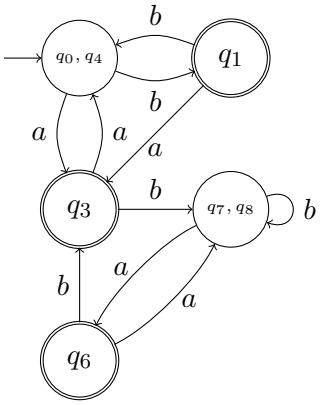


FIGURE 6.7 – Automate minimal $\mathsf{eqmin}(A_5)$.

Chapitre 7

Introduction à l'analyse syntaxique

Les chapitres précédents ont présenté les automates finis comme un modèle de programme avec de bonnes propriétés (déterminisation, forme canonique, etc) et des limites (correspondance à la classe des langages réguliers). Une application fréquente, et immédiate, des automates (finis) est l'**analyse syntaxique**, c'est à dire la reconnaissance d'une structure dans un flux de données, et éventuellement, le calcul d'une valeur sur ce flux. Nous voyons dans ce chapitre comment formaliser l'analyse syntaxique au moyen des automates finis, puis comment utiliser un automate fini pour programmer un analyseur syntaxique. Ce chapitre n'a pas pour but d'être une référence sur l'analyse syntaxique, qui est abordée plus longuement dans les ouvrages traitant de la compilation, mais de présenter comment les automates finis peuvent être utilisés pour programmer des analyses syntaxiques simples.

7.1 Analyse de la structure d'un flux de données

7.1.1 Obtention du flux de données

En informatique, un **flux de données** (ou **file de données**) est une séquence de données, pour laquelle il est possible :

— d'obtenir la première donnée,
— d'obtenir la donnée suivante,
— de détecter la fin du flux.

Formellement, un flux de données peut donc être vu comme un mot sur un alphabet fixé, les données du flux formant les symboles de l'alphabet.

Exemple 7.1
— Un texte saisi au clavier par un utilisateur (par exemple, `"ceci est un texte"`) est un flux de caractères dont la fin est marquée par `\n`. Ici, l'alphabet correspond au code utilisé sur la machine (par exemple, l'ASCII).
— En langage C, une chaîne de caractères est un tableau dont les éléments sont de type `char`, et dont la fin est marquée par le symbole `\0`. Là encore, l'alphabet est le code utilisé sur la machine (par exemple l'ASCII).
— Un programme C peut être vu à la fois comme un flux de caractères terminé par `EOF`, ou comme un flux de mots clés du langage C (`void`, `#include`, `while`, ...) et de noms de variables, de fonctions, de constantes, etc. Un compilateur C utilise cette deuxième vue pour analyser le programme, en vérifier la syntaxe, et le compiler vers du code

exécutable. Dans la première vue, l'alphabet est le code utilisé sur la machine. Dans la seconde vue, l'alphabet est l'ensemble des mots clés du langage C. ⧫

Dans ce chapitre introductif, nous considérons systématiquement que l'alphabet est le code utilisé sur la machine programmée, par exemple le code ASCII. Le flux de données s'obtient donc par une lecture caractère à caractère, comme le montre le programme C en figure 7.1 pour une lecture depuis l'entrée standard. Dans ce programme, le premier appel à la fonction

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char * argv[]) {
5    int c;
6
7    c = getchar();
8    while (c != '\n') {
9      /* Traiter le symbole c */
10     c = getchar();
11   }
12
13   return EXIT_SUCCESS;
14 }
```

FIGURE 7.1 – Programme C de lecture caractère par caractère.

`getchar()` en ligne 7 donne le premier caractère du flux, les suivants étant obtenus lors de chaque appel à la fonction `getchar()` en ligne 10. La fin du flux est détectée en ligne 8 par la lecture de `'\n'` produit par l'utilisateur lorsqu'il a pressé la touche `return`. Le **mot d'entrée** lu par notre programme est la suite de symboles privées de `'\n'`.

Exemple 7.2 Si l'utilisateur entre `abcde`, notre programme lira successivement les caractères `a`, `b`, `c`, `d` et `e`, avant de lire `'\n'`. Le flux de données lu est donc `abcde\n` où `\n` est un seul caractère. Le mot d'entrée lu par notre programme est donc `abcde`. ⧫

7.1.2 Reconnaissance de la structure du flux de données

Nous disposons donc d'un programme permettant d'obtenir un flux de données. Nous voyons désormais le flux comme un mot w sur un alphabet Σ. Le problème de la reconnaissance de la structure du flux de données revient à décider si w possède une bonne structure fixée.

Exemple 7.3 Nous souhaitons écrire un programme qui vérifie si une suite de caractères saisie au clavier forme un entier relatif encodé en base 10. Nous savons qu'un entier est un mot sur l'alphabet $\{-, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ tel que :

— le premier caractère est $-$ ou un chiffre parmi $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$,
— les caractères suivants, s'il y en a, sont des chiffres parmi $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$,
— si le premier caractère est $-$, alors il y a nécessairement un second caractère.

Ainsi, les mots 12 et -2345 sont bien formés. Par contre, $-$, ε et $1-234$ sont mal formés.⧫

Il existe donc un ensemble de mots w bien formés, et un ensemble de mots mal formés ; c'est à dire un langage L des mots bien formés, $\Sigma^* \setminus L$ étant celui des mots mal formés. Donc reconnaître la structure de w consiste à décider si $w \in L$. Nous savons que ce problème admet une solution lorsque L est régulier.

Exemple 7.4 Poursuivons l'exemple 7.3. L'ensemble des mots qui représentent un entier est régulier puisqu'il peut être décrit par une expression régulière :

$$(- + \epsilon)(0+1+2+3+4+5+6+7+8+9)(0+1+2+3+4+5+6+7+8+9)^\bigstar$$

À partir de cette expression régulière nous pouvons calculer l'automate minimal qui reconnaît ce langage. Il est représenté en figure 7.2. Les transitions étiquetées par plusieurs symboles

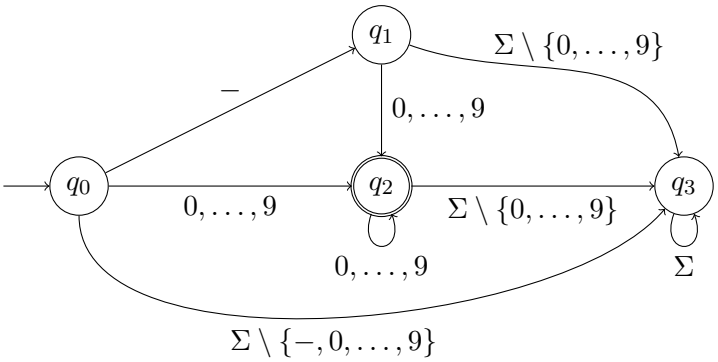


FIGURE 7.2 – Automate minimal qui décide si un mot représente un entier.

(par exemple $0,\ldots,9$) représentent autant de transitions, chacune d'elles étiquetée par l'un des symboles. Σ est l'alphabet défini par l'encodage de la machine utilisé (par exemple le code ASCII). ⧫

Nous supposons dans la suite que L est donné par l'automate fini minimal A_L qui le reconnaît. La minimalité de A_L est importante car elle assure d'une part que A_L est déterministe et complet. C'est à dire que L est programmable d'une part, et robuste d'autre part. La robustesse de l'automate en figure 7.2 se voit dans ses transitions étiquetées Σ (éventuellement privé de certains symboles). Ainsi, que le mot d'entrée soit accepté ou non, l'automate possède une exécution sur laquelle il lit l'intégralité du mot d'entrée, et ne bloque jamais. Enfin, la minimalité garantit également que la programmation de A_L donne un programme aussi petit que possible.

Afin d'obtenir un programme qui décide $w \in L$, il suffit donc de mettre en œuvre l'automate $A_L = (Q, \Sigma, \delta, q_0, F)$ en lieu et place de la ligne 9 dans le programme de la figure 7.1. Cette mise en œuvre consiste à simuler A_L sur le mot d'entrée w pour décider $w \in L$. Il s'agit donc de mémoriser l'état de l'automate, en partant de q_0, et en le faisant évoluer à chaque lettre lue en accord avec δ. Finalement, une fois le mot entièrement lu, il suffit de regarder si l'état mémorisé est accepteur ou non.

Exemple 7.5 La figure 7.3 présente un programme C qui met en œuvre l'automate de la figure 7.2. Nous introduisons en ligne 5 le type `enum Q` pour représenter les états de l'automate.

```
1  #include <ctype.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  enum Q {q0,q1,q2,q3};
6
7  int main(int argc, char * argv[]) {
8    int c;
9    enum Q q = q0;
10
11   c = getchar();
12
13   while (c != '\n') {
14
15     switch (q) {
16       case q0:
17         if (c == '-')
18           q = q1;
19         else if (isdigit(c))
20           q = q2;
21         else
22           q = q3;
23         break;
24       case q1:
25         if (isdigit(c))
26           q = q2;
27         else
28           q = q3;
29         break;
30       case q2:
31         if (isdigit(c))
32           q = q2;
33         else
34           q = q3;
35         break;
36       case q3:
37         q = q3;
38         break;
39     }
40
41     c = getchar();
42   }
43
44   if (q == q2)
45     printf("Mot␣accepté\n");
46   else
47     printf("Mot␣rejeté\n");
48
49   return EXIT_SUCCESS;
50 }
```

FIGURE 7.3 – Programme C d'analyse syntaxique d'un entier saisi au clavier.

Nous aurions pu utiliser un entier, mais un type énuméré nous assure qu'une variable de type `enum Q` ne peut prendre aucune autre valeur que celles énumérées en ligne 5. La variable qui mémorise l'état de l'automate est déclarée en ligne 9 et initialisée à l'état initial `q0`.

L'instruction `switch` des lignes 15 à 39 code la relation de transition δ de l'automate. C'est à dire que pour tout état mémorisé dans `q` et pour tout symbole lu dans `c`, elle met à jour la valeur de `q` conformément à δ. Elle propose un `case` pour chaque valeur possible de `q`. Par exemple, en ligne 16, nous trouvons les transitions issues de l'état `q0` :

— si `c` vaut `'-'`, alors l'automate se déplace dans l'état `q1` ;
— si `c` vaut `'0'`, `'1'`, `'2'`, `'3'`, `'4'`, `'5'`, `'6'`, `'7'`, `'8'` ou `'9'`, l'état suivant de l'automate est `q2` ;
— enfin, dans tout autre cas, c'est à dire, lorsque `c` appartient à $\Sigma \setminus \{-, 0, \ldots, 9\}$, l'automate atteint l'état `q3`.

Les lignes 24, 30 et 36 donnent les transitions pour les états `q1`, `q2` et `q3`. Nous laissons le soin au lecteur de vérifier qu'elles correspondent bien à l'automate de la figure 7.2.

Observons que le `case q3` n'était pas nécessaire. Cependant, le code est plus lisible (on y retrouve précisément l'automate de la figure 7.2) et plus simple à maintenir : cela diminue les risques d'oubli. Notons également que si l'automate n'est pas déterministe, il n'est pas possible de le mettre en œuvre ainsi car la valeur de `q` ne peut pas être déterminée.

Enfin, en ligne 44 le programme teste si, une fois le mot d'entrée lu par l'automate, l'état de celui-ci est accepteur (ici, seul `q2` est accepteur). Il affiche alors un texte correspondant à son verdict. ⧫

L'instruction `switch` reproduit lisiblement la relation de transition de l'automate. Il est (ici) possible de la mettre en œuvre de manière plus compacte en testant d'abord le symbole lu, puis l'état courant, afin de décider de l'état suivant. L'extrait de programme C en figure 7.4 remplace l'instruction `switch` des lignes 15 à 39.

```
1  ...
2      if (c == '-') {
3        if (q == q0)
4          q = q1;
5        else
6          q = q3;
7      }
8      else if (isdigit(c)) {
9        if (q != q3)
10         q = q2;
11       else
12         q = q3;
13     }
14     else
15       q = q3;
16 ...
```

FIGURE 7.4 – Mise en œuvre compacte de la relation de transition.

7.2 Calcul depuis un flux de données

Nous posons maintenant un problème un peu plus complexe. En plus d'avoir reconnu la structure d'un flux, nous souhaitons réaliser un calcul sur celui-ci.

Exemple 7.6 Dans la continuité de l'exemple 7.5, nous voulons maintenant, en plus d'accepter ou de rejeter le mot lu suivant qu'il représente ou non entier encodé en base 10 :
— déterminer si sa valeur déborde du domaine représentable en machine ;
— et en calculer la valeur lorsqu'elle est représentable. ⧫

Une première solution à ce problème consiste à mémoriser le mot lu dans une chaîne de caractères lors de sa lecture par l'automate. Cette solution consiste donc en la mise en œuvre d'un automate fini (comme en figure 7.3), puis un ensemble de traitements classiques en C sur la chaîne de caractères (ou autre) ayant servi à mémoriser le mot d'entrée lorsque celui-ci est accepté. Il existe des problèmes pour lesquels il est nécessaire de stocker le mot d'entrée afin de pouvoir le parcourir plusieurs fois. Cette approche n'est cependant pas efficace :
— elle nécessite de lire deux fois le mot d'entrée ;
— et elle nécessite de stocker le mot d'entrée en mémoire.
De plus un flux de données peut avoir une taille conséquente (plusieurs gigaoctets, voire même quelques teraoctets), et il n'est pas possible ou souhaitable de les stocker en mémoire ou de les parcourir plusieurs fois. Nous proposons donc une seconde solution où les traitements sont réalisés au fur et à mesure de la lecture du mot d'entrée. Nous pouvons donc voir cela comme l'exécution d'instructions en langage C le long des transitions de l'automate.

Exemple 7.7 La figure 7.5 présente un programme qui accepte les mots d'entrée qui codent un entier, et rejette les autres. Lorsqu'un mot est accepté, le programme détecte les débordement de valeur. En l'absence de débordement, le programme calcule la valeur de l'entier représenté par le mot d'entrée. La structure du programme est globalement la même que celui de la figure 7.3 dans lequel la mise en œuvre compacte de la relation de transition (voir figure 7.4) a été substituée. En ligne 11 une variable `signe` a été ajoutée pour mémoriser le signe de l'entier lu. Elle vaut `1` lorsque l'entier est positif (et par défaut), et `-1` lorsque l'entier est négatif. Nous avons ajouté en ligne 12 une variable `n` qui stocke la valeur absolue de l'entier lu. L'entier lu au clavier a donc *in fine* la valeur `signe` $\times$ `n`. Enfin, en ligne 13 la variable `débordement` a été introduite afin de mémoriser un éventuel débordement de capacité.

Sur les transitions de symboles `-`, et `0`,...,`9`, nous avons ajouté du code C afin de tester les débordement de capacité et afin de calculer la valeur de l'entier lu. Sur lecture du symbole `-`, nous devons retenir que l'entier lu est négatif. Nous modifions donc la variable `signe` en conséquence en ligne 21. Sur transition par lecture d'un chiffre `chiffre`, la valeur de `n` devient `10*n+chiffre` puisque nous lisons l'entier en digit de poids fort d'abord. Le chiffre lu dans le caractère `c` vaut `digittoint(c)` (ligne 28), en effet, la valeur de `c` est le code ASCII du chiffre qu'il représente. La valeur de `n` est modifiée en ligne 32 à condition qu'aucun débordement de capacité ne se produise. Un débordement de capacité correspond à l'impossibilité pour la machine de représenter une valeur. Le type de `n` est `int` pour lequel les valeurs maximales représentables sont données par les constantes `INT_MIN` et `INT_MAX` définies dans la bibliothèque `limits.h`. Il y a débordement de capacité lorsque :

$$10 * \texttt{n} + \texttt{chiffre} > \texttt{INT\_MAX} \qquad \text{ou} \qquad 10 * \texttt{n} + \texttt{chiffre} > -\texttt{INT\_MIN}$$

```
#include <ctype.h>
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

enum Q {q0,q1,q2,q3};

int main(int argc, char * argv[]) {
  int c;
  enum Q q = q0;
  int signe = 1;
  int n = 0;
  char debordement = 0;
  int chiffre;

  c = getchar();

  while (c != '\n') {

    if (c == '-') {
      signe = -1;
      if (q == q0)
        q = q1;
      else
        q = q3;
    }
    else if (isdigit(c)) {
      chiffre = digittoint(c);
      if ((n>(INT_MAX-chiffre)/10) || ((n>-((INT_MIN+chiffre)/10))))
        debordement = 1;
      else
        n = 10*n+chiffre;
      if (q != q3)
        q = q2;
      else
        q = q3;
    }
    else
      q = q3;

    c = getchar();
  }

  if (q == q2) {
    printf("Mot␣accepté: ");
    if (debordement)
      printf("débordement de valeur.\n");
    else
      printf("%d\n", signe*n);
  }
  else
    printf("Mot␣rejeté\n");

  return EXIT_SUCCESS;
}
```

FIGURE 7.5 – Programme C d'analyse syntaxique et de calcul d'un entier saisi au clavier.

Cette condition est testée en ligne 29, et si un débordement est détecté, la valeur de la variable `debordement` est mise à jour en conséquence en ligne 30. Notons l'ordre des opérations (négation et division par 10) sur `INT_MIN` en ligne 29 afin de ne pas provoquer de débordement dans le test.

Enfin, lorsque le mot est accepté, nous testons en ligne 46 la valeur de la variable `debordement` pour afficher, soit la valeur de l'entier lu au clavier lorsqu'aucun débordement n'a été constaté, soit un message signalant le débordement de capacité. ♦

7.3 Analyse syntaxique et langages réguliers

Une différence majeure entre les programmes des figures 7.3 et 7.5 est l'utilisation de variables en plus de celles (`q` et `c`) nécessaires à la simulation de l'automate fini. Ces variables sont nécessaires pour réaliser d'autres traitements, mais elles augmentent potentiellement la capacité de l'automate fini puisqu'il dispose d'une autre variable que son état `q` pour mémoriser une information. Dans le programme en figure 7.5, les variables annexes (`signe`, `n`, `debordement` et `chiffre`), sont mises à jour lors des franchissement de transitions, mais leurs valeurs ne sont pas utilisées pour décider quelle transition franchir. Par exemple, dans le programme en figure 7.5, qu'il y ait débordement ou non en ligne 29, ce sont toujours uniquement `q` (ligne 33) et `c` (ligne 27) qui déterminent l'état suivant.

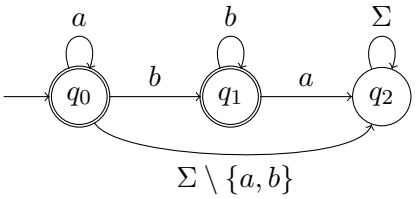


FIGURE 7.6 – Automate qui décide le langage a^*b^* sur $\Sigma \supseteq \{a, b\}$.

La figure 7.7 présente un programme C basé sur l'automate fini en figure 7.6 qui décide le langage a^*b^* sur un alphabet Σ contenant $\{a, b\}$. Le programme en figure 7.7 déclare en outre une variable `n` en ligne 9, initialisée à `0`, puis utilisée en ligne 17 pour compter le nombre de `a` lus (incrémentation de `n`), puis en lignes 21 et 29 pour compter le nombre de `b` lus (décrémentation de `n`).

À la différence des mises en œuvre d'automates précédentes, le programme de la figure 7.7 utilise ensuite la variable `n` pour décider s'il doit accepter le mot lu ou non, en ligne 43. Il accepte les mots du langage a^*b^* pour lesquels $n = 0$, c'est à dire pour lesquels le nombre de a lus est égal au nombre de b lus. En d'autres termes, ce programme décide le langage $\{a^n b^n \mid n \in \mathbb{N}\}$ qui n'est pas régulier (voir exemple 4.3 page 47).

Ainsi donc, l'utilisation de variables autres que `q` et `c` pour déterminer quelle transition franchir, ou pour décider si le mot d'entrée doit être accepté ou non, permet de décider des langages qui ne sont pas réguliers. Le programme de la figure 7.7 met en exergue la limite des langages réguliers : seule une mémoire finie (ici, l'état représenté par `q`) peut être utilisée. Dès qu'une mémoire infinie (ici `n`) est utilisée, nous sortons de la classe des langages réguliers. Le programme de la figure 7.7 ne peut pas être modélisé par un automate fini.

▷ En toute rigueur, le programme de la figure 7.7, lorsqu'il est exécuté sur une machine donnée, ne peut pas décider le langage $\{a^n b^n \mid n \in \mathbb{N}\}$ mais un sous-ensemble fini de celui-ci. En effet, `n` étant de type `int`, sa valeur maximale est `INT_MAX`, et donc le langage décidé par le programme est $\{a^n b^n \mid n \leq \texttt{INT\_MAX}\}$ qui est régulier. Cependant cette limite est une limite de la machine, et non pas une limite du programme. Ainsi, pour une machine fixée (une capacité de mémoire fixée), le langage reconnu est régulier. Mais sur l'ensemble des machines possibles et imaginables, `n` n'est pas bornée, et le langage est irrégulier.

▷ Pour aller plus loin sur l'analyse syntaxique : http://www.complang.org/ragel/

```c
#include <stdio.h>
#include <stdlib.h>

enum Q {q0,q1,q2};

int main(int argc, char * argv[]) {
  int c;
  enum Q q = q0;
  int n = 0;

  c = getchar();

  while (c != '\n') {
    switch (q) {
      case q0:
        if (c == 'a') {
          n++;
          q = q0;
        }
        else if (c == 'b'){
          n--;
          q = q1;
        }
        else
          q = q2;
        break;
      case q1:
        if (c == 'b'){
          n--;
          q = q1;
        }
        else
          q = q2;
        break;
      case q2:
        q = q2;
        break;
    }

    c = getchar();
  }

  if (((q == q0) || (q == q1)) && (n == 0))
    printf("Mot␣accepté\n");
  else
    printf("Mot␣rejeté\n");

  return EXIT_SUCCESS;
}
```

FIGURE 7.7 – Programme C qui décide le langage $\{a^n b^n \mid n \in \mathbb{N}\}$.

Index

Automates finis - TD "langages et automates finis"
Département informatique ENSEIRB-MATMECA 27 février 2024

Voir la page `http://herbrete.vvv.enseirb-matmeca.fr/IF114` pour toute information complémentaire sur cet enseignement.

1 Langages et automates finis

Pour les exercices suivants, vous pourrez utiliser l'outil JFLAP [1] afin de créer et de simuler les automates finis.

Exercice 1 (Premiers langages et automates)
Pour chaque langage ci-dessous sur l'alphabet $\{\mathtt{a}, \mathtt{b}\}$, donnez un automate fini déterministe qui le reconnaît :

1. le nombre de `a` est un multiple de 4
2. un nombre pair de `a` et impair de `b`
3. tout symbole `a` est précédé et suivi d'au moins un symbole `b`

Exercice 2 (Langages et encodages d'entiers)
Pour chaque langage ci-dessous sur l'alphabet binaire $\{0, 1\}$, donnez un automate fini déterministe qui le reconnaît :

1. les entiers pairs
2. les entiers multiples de 3

Exercice 3 (Automates non-déterministes)
Pour chaque langage ci-dessous sur l'alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, donnez un automate fini non-déterministe qui le reconnaît :

1. les mots qui se terminent par `aa`
2. les mots dont la dernière lettre apparaît précédemment dans le mot (ex : `aba`, mais pas `aab`)
3. les mots dont la dernière lettre n'apparaît pas précédemment dans le mot

Exercice 4 (Algorithme de simulation)
Écrire un algorithme qui indique si un automate fini *déterministe et complet* $A = (Q, \Sigma, \delta, q_0, F)$ accepte un mot $w \in \Sigma^*$.

Exercice 5 (État initial)
Soit $A = (Q, \Sigma, \delta, I, F)$ un automate fini avec au moins deux états initiaux (i.e. $|I| > 1$). Construire un automate $A' = (Q', \Sigma', \delta', q_0, F')$ ayant un seul état initial et tel que $\mathcal{L}(A) = \mathcal{L}(A')$. Justifiez.

Exercice 6 (Miroir)
Soit $w = a_0 \cdots a_n$ un mot sur Σ (i.e. $a_0, \ldots, a_n \in \Sigma$). On note $w^R = a_n \cdots a_0$ le mot *miroir* de w obtenu en lisant w de droite à gauche. On généralise cette construction à un langage : $\mathcal{L}^R = \{w^R \mid w \in \mathcal{L}\}$.

Soit A un automate fini. Construire un automate A^R qui reconnaît le langage $\mathcal{L}(A)^R$. Justifiez.

1. Voir `http://www.enseirb-matmeca.fr/~herbrete/IF114` pour l'utilisation de JFLAP.

2 Jeu des portes bascule

On considère le jeu [2] représenté en figure 1.

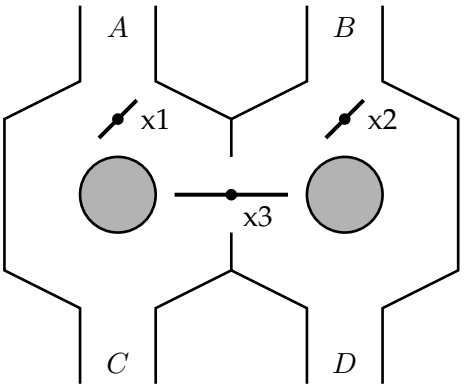


FIGURE 1 – Le jeu des portes-bascule.

On dispose pour ce jeu d'un nombre donné de billes que l'on insère soit en A, soit en B. Les billes ressortent soit en C, soit en D selon l'orientation des portes-bascule x1, x2 et x3. Lorsqu'une bille heurte une porte, cette dernière dévie la trajectoire de la bille *puis* elle change de position. Les portes x1 et x2 font passer les billes à gauche ou à droite des obstacles. Les portes x1 et x2 peuvent être orientées à gauche ou à droite. La porte x3 fait passer les billes de la partie gauche du jeu à sa partie droite, et inversement. Elle peut être orientée horizontalement, forçant alors les billes à changer de côté, ou verticalement, bloquant alors le passage des billes.

On gagne ce jeu lorsque la dernière bille jouée sort par D.

Les portes sont initialement dans la position représentée en figure 1. On cherche à définir les parties pour lesquelles on gagne ce jeu. Il s'agira donc, connaissant le nombre de billes à jouer, de définir où chaque bille doit être introduite (A ou B) pour l'emporter.

Exercice 7 (Un aperçu des parties gagnantes)
Donnez les parties gagnantes pour 3 billes.

Exercice 8 (Formalisation du problème)
On cherche maintenant une méthode permettant de calculer les parties gagnantes pour un nombre quelconque de billes. Afin de déterminer si le jeu d'une bille est gagnant, nous devons savoir si elle sort en C ou en D.

1. Quels paramètres définissent la sortie (C ou D) empruntée par une bille ? En déduire ce qui définit un *état* du jeu.
2. Quel est l'*ensemble des états* du jeu ? Quel est l'*état initial* du jeu ?
3. Quels paramètres influent sur le changement d'état du jeu lors de l'insertion d'une bille ? Définir l'*alphabet* et la *fonction de transition* du jeu.

2. D'après "Introduction to Automata Theory, Languages, and Computation", J. E. Hopcroft, R. Motwani et J. D. Ullman.

4. Comment définit-on le gain du jeu ?

Exercice 9 (Simulation du modèle)
Créez l'automate fini obtenu dans JFLAP et simulez le pour les séquences d'entrée à trois billes. Indiquez si la séquence jouée est gagnante ou non en vérifiant par rapport aux résultats de l'exercice 6.

Exercice 10 (Calcul des parties gagnantes)
1. Définissez formellement l'ensemble des parties gagnantes du jeu des portes-bascule
2. Calculez l'ensemble des parties gagnantes pour ce jeu
3. Vérifiez que les parties gagnantes pour 3 billes sont bien celles obtenues au premier exercice

Automates finis - TD "déterminisation"
Département informatique ENSEIRB-MATMECA 27 février 2024

Voir la page `http://herbrete.vvv.enseirb-matmeca.fr/IF114` pour toute information complémentaire sur cet enseignement.

L'algorithme de déterminisation des automates finis vu en cours et étudié dans cette feuille de TD est mis en œuvre dans l'outil JFLAP (voir la page www de l'enseignement pour son utilisation). Il est accessible via le menu `Convert/Convert to DFA`. Nous recommandons d'utiliser cet outil pour renforcer la compréhension de l'algorithme de déterminisation et pour s'exercer.

1 Automates finis non-déterministes

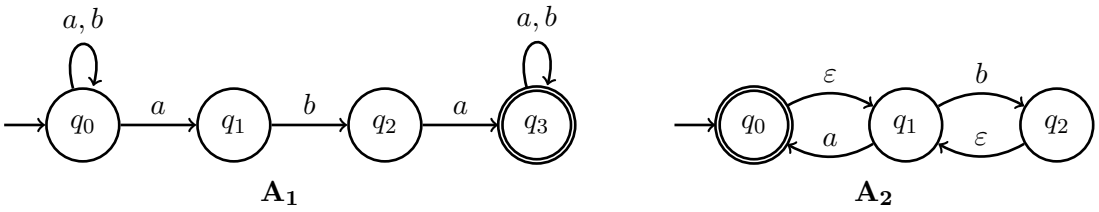


FIGURE 1 – Deux automates non-déterministes.

Exercice 1 (Mots acceptés)
Indiquez si les mots suivants sont acceptés par les automates A_1 et A_2 en figure 1.

1. a
2. aba
3. $baba$
4. $baabab$
5. $bbbaabba$

Exercice 2 (Simulation)
Donner un algorithme permettant de décider si $w \in \Sigma^*$ est accepté par A, automate fini non-déterministe.

2 Algorithme de déterminisation

Nous avons vu que pour simuler un automate non-déterministe, il suffit de calculer l'ensemble des états atteints après avoir lu un préfixe du mot d'entrée, et de maintenir cet ensemble à jour à chaque lecture d'une lettre.

Le tableau de la figure 2 généralise ce calcul en prenant en compte non pas la lettre lue, mais toutes les lettres possibles de Σ. On commence naturellement par l'ensemble $\{q_0\}$ (colonne E), et l'on calcule les ensembles d'états atteints par une transition a et b (colonnes l et F). Lorsqu'un ensemble d'états apparaît dans F mais pas dans E, il y est recopié, et le même calcul est effectué sur cet ensemble (cas de $\{q_0, q_1\}$ ici).

E	l	F
$\{q_0\}$	a b	$\{q_0, q_1\}$ $\{q_0\}$
$\{q_0, q_1\}$	a b	
	a b	
	a b	
	a b	
	a b	

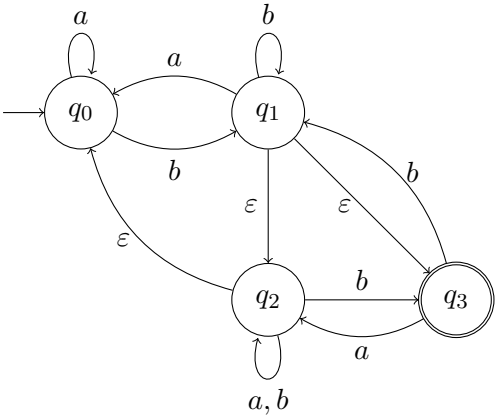
FIGURE 2 – Power-set construction.

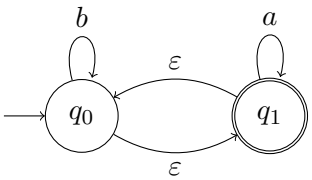
Exercice 3 (Power-set construction)
Complétez le tableau de la figure 2 pour l'automate fini non-déterministeA_1 de la figure 1.
Nota bene : il y a exactement le bon nombre de lignes.

Exercice 4 (Propriétés de la construction)
Dessinez l'automate dont les états sont les ensembles représentés en colonne E du tableau de la figure 2, et donc les transitions sont données par les lignes de ce tableau.

1. D'après la construction de eqdet(A) vue en cours, quel est l'état initial de l'automate et quels sont ses états accepteurs ?
2. Quelle(s) propriété(s) remarquable(s) cet automate possède-t-il ?

Exercice 5 (Application de l'algorithme)
À l'aide de l'algorithme vu en cours, calculez les automates déterministes correspondant à l'automate A_2 de la figure 1, ainsi qu'aux automates suivants.





3 Algorithmes sur les automates non-déterministes

Exercice 6 (Langage vide)
Donner un algorithme permettant de décider si le langage reconnu par un automate fini déterministe A est vide ou non. Cet algorithme est-il correct pour les automate non-déterministes ? Quelle est sa complexité ?

Exercice 7 (Langage universel)
Le langage d'automate A est universel s'il contient tous les mots, c'est à dire $L(A) = \Sigma^*$.

1. Donner un algorithme permettant de décider si le langage de A est universel pour un automate déterministe A. Quelle est sa complexité ?
2. Expliquer pourquoi l'algorithme précédent n'est pas correct si A est non-déterministe ? Donner un algorithme pour les automates non-déterministes. Quelle est sa complexité ?

Exercice 8 (Inclusion des langages)
Soient A et B deux automates finis. On cherche à décider si $L(A) \subseteq L(B)$.

1. Donner un algorithme dans le cas où B est déterministe et A quelconque. Quelle est sa complexité ?
2. Expliquer pourquoi cet algorithme n'est pas correct dans le cas où B n'est pas déterministe. Donenr un algorithme dans ce cas. Quelle est sa complexité ?

4 Complexité de la déterminisation

On s'intéresse à la taille de l'automate déterministe équivalent à un automate non-déterministe donné obtenu par la *power-set construction*. Le nombre d'itérations de la boucle `Tant que` de l'algorithme est égal au nombre d'états de l'automate déterministe. Nous cherchons donc à estimer ce nombre.

Exercice 9 (Des automates non-déterministes particuliers)
On définit le langage L_i comme celui des mots sur $\{a, b\}$ dont la i-ème lettre en partant de la fin est un a. Donnez 3 automates finis qui reconnaissent respectivement L_1, L_2 et L_3. (Indication : le non-déterminisme est une aide précieuse)

Exercice 10 (Taille de l'automate déterministe)
Remplissez le tableau suivant. Pour obtenir les automates déterministes demandés, utilisez JFLAP (menu `Convert/Convert to DFA`). Vous extrapolerez le cas de L_n.

Langage	Nb. états AND	Nb. états AFD
L_1		
L_2		
L_3		
L_n ?		

Exercice 11 (Expression régulière et taille de l'AFD)
On considère l'alphabet $\Sigma = \{a, b\}$.

1. Donner une famille d'automates finis non-déterministes pour la famille des langages représentés par les expressions régulières $\Sigma^* a \Sigma^n$ pour $n \geq 0$.

2. À partir de la construction précédente et en utilisant le théorème de Kleene, donner une famille d'automates finis non-déterministes A_n pour les expressions régulières E_n définie par $E_n = \Sigma^* a \Sigma^n + \Sigma^* b \Sigma^n$. Quelle est la taille de A_n et de $\mathsf{eqdet}(A_n)$ pour n valant 1, 2 et 3 ? Extrapolez le cas général.

n	Nb. états A_n	Nb. états $\mathsf{eqdet}(A_n)$
1		
2		
3		
cas général		

3. On considère maintenant la famille d'expressions régulières F_n définie par $F_n = \Sigma^* \Sigma^{n+1}$. Donner une famille d'automates finis non-déterministes B_n qui reconnaissent le langage des F_n. Quelle est la taille de B_n ? Quelle est la taille de $\mathsf{eqdet}(B_n)$ pour n valant 1, 2 et 3 ? Extrapolez le cas général.

n	Nb. états B_n	Nb. états $\mathsf{eqdet}(B_n)$
1		
2		
3		
cas général		

4. Comparer les langages de E_n et F_n ainsi que les tailles de $\mathsf{eqdet}(A_n)$ et $\mathsf{eqdet}(B_n)$.

Exercice 12
Complexité au pire de la déterminisation Si A est un automate non-déterministe à n états, quel est au pire cas le nombre d'états de l'automate déterministe équivalent obtenu par la "power-set construction" ? En supposant que l'on dispose d'un programme ne nécessitant qu'une milliseconde pour calculer un état, calculez le temps nécessaire pour rendre déterministe des automates à 5, 10, 50 et 100 états.

Automates finis - TD "Expressions régulières et théorème de Kleene"
Département informatique ENSEIRB-MATMECA 27 février 2024

Voir la page `http://herbrete.vvv.enseirb-matmeca.fr/IF114` pour toute information complémentaire sur cet enseignement.

1 Expressions régulières

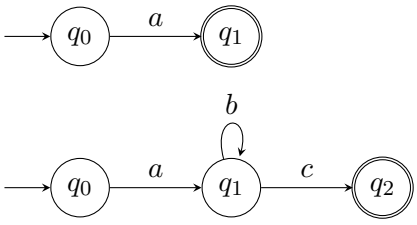
Exercice 1 (Syntaxe des expressions régulières)
Indiquer, parmi les expressions suivantes, lesquelles sont effectivement des expressions régulières sur l'alphabet $\{a, b\}$:

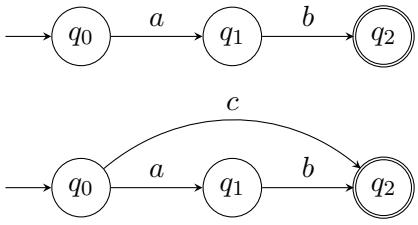
1. a
2. c
3. $a + b^*$
4. $(a + b)^*$
5. a^*b
6. $(abba + baba)^* = bababa$

Exercice 2 (Langage défini par une expression régulière)
Donner les langages définis par les expressions régulières suivantes sur l'alphabet $\{a, b\}$, ainsi que des exemples de mots appartenant à ces langages :

1. a
2. ab^*
3. $a(a + b)^*$
4. $(abba + baba)^*$

Exercice 3 (Expression régulière associée à un chemin)
Donnez des expressions régulières pour les langages reconnus par les automates suivants.





2 Théorème de Kleene

Le théorème de Kleene stipule que :

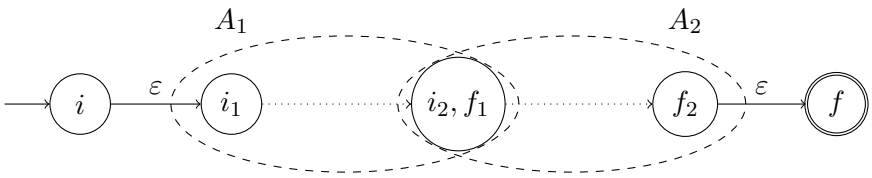
"Les langages réguliers sont les langages décidés par les automates finis"

Nous allons maintenant illustrer la partie de ce théorème qui permet de construire des automates. Pour toute expression régulière α, nous pouvons construire un automate fini qui décide le langage défini par α. Vous vous réfèrerez au polycopié pour la construction de Kleene qui associe un automate fini A_α à toute expression régulière α.

Exercice 4 (Construction de Kleene)
Appliquer la construction de Kleene à l'expression régulière suivante :

$$((a + b)^*.c)^*$$

Exercice 5 (Simplifier la concaténation ?)
La construction de Kleene pour l'expression $\alpha_1\alpha_2$ ajoute une transitions ε allant de l'état final f_1 de l'automate qui reconnaît $\mathcal{L}(\alpha_1)$ à l'état initial i_2 de l'automate qui reconnaît $\mathcal{L}(\alpha_2)$. Cette transition peut sembler superflue, on propose donc de simplement fusionner les deux états f_1 et i_2 :



Cette modification est incorrecte. Trouvez deux automates A_1 et A_2 tels que l'automate obtenu par la construction ci-dessus ne reconnaît pas le langage $\mathcal{L}(A_1).\mathcal{L}(A_2)$.

Justifiez que l'automate obtenu par la construction de Kleene appliquée à A_1 et A_2 reconnaît bien le langage $\mathcal{L}(A_1).\mathcal{L}(A_2)$.

3 Outils UNIX

De nombreux outils de l'environnement UNIX utilisent les expressions régulières afin de permettre de faire des recherches dans des chaînes de caractères. L'utilisation la plus courante est celle des *wildcard expansions* dans les shells comme `bash`, mais la plupart des outils ont leur propre idée de la façon de représenter ces expressions régulières (cf. figure 1). Des exemples de ces expressions régulières sont donnés en figure 2. Ici, nous présentons deux de ces outils, `grep` et `sed`, utilisés pour les recherches et les transformations de chaînes de caractères.

Symbole :	`emacs`	`sed`	`awk`	`grep`	`grep -E`	Signification :
`.`	✓	✓	✓	✓	✓	un caractère quelconque
`[ ]`	✓	✓	✓	✓	✓	un ensemble de caractères
`[^ ]`	✓	✓	✓	✓	✓	le complément de ce même ensemble
`?`	✓		✓		✓	Répétition zéro ou une fois
`*`	✓	✓	✓	✓	✓	Étoile de Kleene : répétition 0 ou plusieurs fois
`|`			✓		✓	Somme de deux expressions régulières
`|`	✓	✓		✓		
`^`	✓	✓	✓	✓	✓	début de ligne
`$`	✓	✓	✓	✓	✓	fin de ligne
`\`	✓	✓	✓	✓	✓	protection du caractère suivant
`\( \)`	✓	✓		✓		grouper les sous-expressions
`( )`			✓		✓	

FIGURE 1 – Expressions régulières dans plusieurs outils UNIX

Exercice 6 (Utilisation de grep)
La commande `grep` permet de rechercher dans une chaîne de caractères[1], dans un fichier

1. Et le plus souvent dans un flot de caractères (*stream*), ce qui inclut les redirections des sorties standard.

Motif :	Signification :
`mer*`	*me, mer, merr, merrr, ...*
`b[aeiuo]g`	la deuxième lettre est une voyelle
`^...$`	une ligne contenant exactement trois caractères
`[A-Za-z]`	une lettre de l'alphabet
`[^0-9a-zA-Z]`	tout symbole qui n'est ni une lettre ni un chiffre
`^[^a]`	une ligne dont le premier caractère n'est pas un 'a'

FIGURE 2 – Exemples d'expressions régulières UNIX

et même dans un ensemble de fichiers, la présence d'un motif (*pattern*) représenté par une expression régulière :

`grep -e '⟨regexp⟩' '⟨string⟩'`
`grep -e '⟨regexp⟩ ⟨file1⟩ ...⟨filen⟩`

Exemples :
- `grep alias ~/.bash_profile`
- `cat ~/.bash_profile | grep alias`

Ne pas hésiter à aller lire la page de manuel de grep pour voir les options disponibles pour `grep`. Pour chacune des questions suivantes, spécifier une expression régulière répondant à la question, et tester la commande sous votre environnement de travail.

1. Rechercher dans votre fichier `.bash_export` les lignes contenant le mot `PATH`. Faire la même chose à l'aide de la commande `less` en utilisant le raccourci `'/'`.
2. Rechercher dans le résultat de la commande `last` les lignes contenant votre login.
3. Rechercher à l'aide de la commande `find` l'ensemble des fichiers `.c` sur votre compte.

Exercice 7 (Utilisation de sed)
La commande `sed` (*stream editor*) est particulièrement utile pour transformer des flots de caractères. Il existe de très nombreuses options dans le programme `sed`, mais on se limitera ici à la plus utilisée, celle qui permet de remplacer des chaînes de caractères :

`sed -e 's★⟨regexp1⟩★⟨regexp2⟩★g' ⟨file⟩`

Exemples :
- `sed -e 's/a/b/g' ~/.emacs`
- `cat ~/.emacs | sed -e 's/a/b/g'`

Dans la commande précédente, le caractère ★ peut être remplacé par n'importe quel caractère *tant que c'est toujours le même et qu'il n'apparaît pas dans les expressions régulières*. Ne pas hésiter à aller lire la page de manuel de grep pour voir les options disponibles pour `sed`. Pour chacune des questions suivantes, spécifier une expression régulière répondant à la question, et tester la commande sous votre environnement de travail.

1. Écrire une commande qui recherche tous les fichiers `.c` dans votre répertoire, et qui écrit une commande de compilation de ce fichier en `.o` à l'écran (à l'aide du compilateur `gcc` et de l'option `-c`).

2. Écrire une commande qui transforme la sortie de la fonction `date` (à l'aide de la commande `sed`) pour afficher l'heure avant le jour du mois :

```
                                    Fri Sep 11 10:44:51 MEST 2009 →
10:48:11 Fri Sep 11 MEST 2009
```

   Indice : Pour cela, il est utile d'utiliser les commandes de groupage :

```
    echo '18Apr' | sed -e 's/\([0-9]*\)\([A-Za-z]*\)/\2\1/g' →
Apr18
```

   Dans la commande précédente, les expressions régulières entre parenthèses (`\(` et `\)`) sont sauvegardées dans des registres qui peuvent être réutilisés dans la seconde expression avec `\1` et `\2`.

Automates finis - TD "preuve de non-régularité"
Département informatique ENSEIRB-MATMECA 27 février 2024

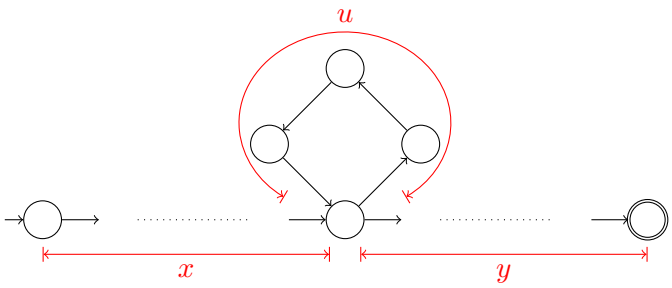
Voir la page `http://herbrete.vvv.enseirb-matmeca.fr/IF114` pour toute information complémentaire sur cet enseignement.

1 Lemme de l'étoile

Soit $A = (Q, q_o, \Sigma, \delta, F)$ un automate déterministe et L le langage régulier reconnu par A. Le lemme de l'étoile donné en cours est le suivant :

Lemme de l'étoile : Il existe un entier $N \in \mathbb{N}$ tel que pour tout mot $w \in L$ de longueur $\geq N$, on peut décomposer le mot w en trois parties $w = xuy$ telles que $|u| > 0$ et $|xu| \leq N$, de manière à assurer que :

$$\forall n \geq 0, \qquad xu^n y \in L$$



Exercice 1 (Preuve du lemme de l'étoile)

1. Donner un schéma de preuve du lemme de l'étoile. En particulier, donner une valeur adéquate pour N et expliquer votre choix.
2. Quel est l'intérêt de la condition $|u| > 0$? Et de la condition $|xu| \leq N$?

En pratique, on veut déterminer si un langage $L \subset \Sigma^\star$ donné est un langage régulier *ou non*.

Exercice 2 (Preuve de régularité)
Quelle est la démarche à adopter pour prouver qu'un langage L donné est régulier ?

Le lemme de l'étoile ne permet donc pas de prouver qu'un langage est régulier ! Nous allons donc l'utiliser pour prouver qu'un langage donné n'est pas régulier.

Exercice 3 (Contraposée du lemme de l'étoile)

1. Écrire la contraposée du lemme de l'étoile.
2. La proposition suivante étant vraie :

   "L est un langage régulier $\Rightarrow$ Le lemme de l'étoile est vrai pour L"

   en déduire une méthode permettant de prouver qu'un langage n'est pas régulier.

Exercice 4 (Régulier...ou pas)
Pour chacun des langages suivants, exhiber une preuve affirmant si le langage est un langage régulier ou non :

1. Les mots de longueur paire sur un alphabet donné.

$$L_1 = \{u \in \{a,b\}^\star, \exists n \in \mathbb{N}, |u| = 2n\}$$

2. Le langage fait des paires de mots, dont le premier est plus court que le second

$$L_2 = \{a^n b^m, n \leq m\}$$

3. L'ensemble des nombres premiers encodé en base unaire :

$$L_3 = \{a^p, p \text{ premier}\}$$

4. Le langage miroir d'un langage régulier L donné.

$$L_4 = \{w^R, w \in L\}$$

où $w^R = a_n \cdots a_0$ est le miroir de $w = a_0 \cdots a_n$.

5. Une petite variation sur le langage miroir :

$$L_5 = \{w\#w^R, w \in \{a,b\}^*\}$$

2 Raisonnement par construction

Exercice 5 (Vers l'utilisation du lemme de l'étoile)
Que penser du raisonnement ci-dessous ?

(i) Le langage $L = a^\star b^\star$ est régulier.
(ii) Le langage $L' = \{a^n b^p, n + p \text{ est pair}\}$ est inclus dans L.
(iii) Or tout sous-ensemble d'un langage régulier est régulier.
(iv) Donc L' est régulier.

Exercice 6 (Une alternative au lemme de l'étoile)
Et de ce raisonnement-ci ?

(i) Considérons le langage $L = \{a^n c^p b^{n+p}, n, p \in \mathbb{N}\}$.
(ii) Le langage $L' = \{a^n b^n, n \in \mathbb{N}\}$ n'est pas régulier.
(iii) Or l'égalité suivante est vraie : $L \cap \{a,b\}^\star = L'$.
(iv) Et l'intersection de deux langages réguliers est un langage régulier.
(v) Donc L n'est pas un langage régulier.

Exercice 7 (Régulier...ou pas)
Pour chacun des langages suivants, exhiber une preuve affirmant si le langage est un langage régulier ou non. Pour certaines preuves de non régularité, il est préférable d'utiliser les deux exercices précédents.

1. Le langage fait des paires de mots, dont le second est plus court que le premier

$$L_6 = \{a^n b^m, n \geq m\}$$

2. Les expressions bien parenthésées.

$$L_7 = \{u \in \{a; b\}^\star, \text{ tel que } |u|_a = |u|_b \text{ et } \forall v \text{ préfixe de } u, |v|_a \geq |v|_b\}$$

3. Le langage préfixe d'un langage régulier L donné.

$$L_8 = \{u \in \Sigma^\star, \exists x \in \Sigma^\star, ux \in L\}$$

4. Les palindromes de longueur paire d'un alphabet donné.

$$L_9 = \{uu^R, u \in \Sigma^\star\}$$

Automates finis - TD "grammaires"
Département informatique ENSEIRB-MATMECA 27 février 2024

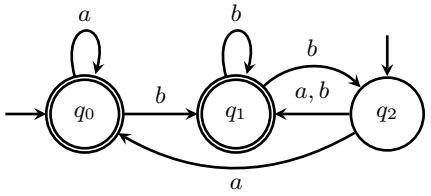
Voir la page `http://herbrete.vvv.enseirb-matmeca.fr/IF114` pour toute information complémentaire sur cet enseignement.

1 $\mathcal{L}(A)$ par la voie algébrique : le lemme d'Arden

Soit $A = (Q, \Sigma, \delta, I, F)$ un automate avec les états $Q = \{q_0, \dots, q_n\}$. On note X_i le langage *reconnu dans l'état* q_i, c'est à dire l'ensemble des mots w tels qu'il existe une exécution $q_i \xrightarrow{w} q$ menant de l'état q_i à un état acceptant $q \in F$.

Considérons l'automate A_0 ci-dessous. On déduit de sa relation de transition qu'un mot accepté depuis l'état q_0 est : soit vide (ε), soit un mot $a \cdot w$ et w est accepté depuis q_0, soit un mot $b \cdot w'$ et w' est accepté depuis q_1. On peut donc exprimer X_0 en fonction de X_0 et X_1 sous la forme d'une équation $X_i = \gamma_0 \cdot X_0 \cup \cdots \cup \gamma_n \cdot X_n \cup \chi$ où $\gamma_0, \dots, \gamma_n, \chi$ sont des langages. Le système d'équations S_{A_0} pour l'automate A_0 est donné ci-dessous.

Automate A_0.



Système d'équations S_{A_0}.

$$X_0 = \{a\} \cdot X_0 \cup \{b\} \cdot X_1 \cup \{\varepsilon\}$$
$$X_1 = \{b\} \cdot X_1 \cup \{b\} \cdot X_2 \cup \{\varepsilon\}$$
$$X_2 = \{a\} \cdot X_0 \cup \{a, b\} \cdot X_1$$

Le système d'équation S_{A_0} se résout grâce au lemme d'Arden :

Lemme d'Arden. Soient A et B deux langages. Le langage $A^* \cdot B$ est le plus petit langage (pour l'inclusion) qui est solution de l'équation $X = A \cdot X \cup B$.
$A^* \cdot B$ est l'unique solution lorsque A ne contient pas le mot vide ε.

Exercice 1 (Lemme d'Arden)

1. Calculez les langages X_0, X_1 et X_2 pour l'automate A_0 en utilisant le Lemme d'Arden. Comment $\mathcal{L}(A_0)$ est-il défini à partir de X_0, X_1 et X_2 ?
2. Démontrez que Σ^* est également solution de l'équation $X = A \cdot X \cup B$ lorsque A contient le mot vide ε.
3. Prouvez le lemme d'Arden

2 Grammaires

Exercice 2 (À gauche, à droite)
Une grammaire est linéaire droite si toutes ses productions sont de la forme $A \to wB$ ou $A \to w$. Inversement, elle est linéaire gauche si toutes ses productions ont la forme : $A \to Bw$ ou $A \to w$.

Montrez que toute grammaire linéaire droite est équivalente à une grammaire linéaire gauche et inversement.

Exercice 3 (Linéarisation)
Soit la grammaire G :

$$S \to A1B$$
$$A \to 0A \mid \varepsilon$$
$$B \to 0B \mid 1B \mid \varepsilon$$

où $V = \{S, A, B\}$ et $\Sigma = \{0, 1\}$.

1. Donnez une grammaire linéaire droite équivalente.
2. Appliquez la construction vue en cours pour construire un automate fini A_G qui reconnaît le langage $\mathcal{L}(G)$.

Exercice 4 (Au-delà des grammaires régulières)
Pour chaque langage *non régulier* ci-dessous, donnez une grammaire hors contexte aussi simple que possible qui le génère :

1. $\{a^n b^n \mid n \in \mathbb{N}\}$
2. $\{a^n b^m \mid n, m \in \mathbb{N} \text{ et } m \leq n\}$
3. $\{ww^R \mid w \in \{a, b\}^*\}$ (rappel : w^R est le mot mirroir de w)
4. le langage des mots de Dyck (expressions bien parenthésées) sur les parenthèses : { }, ( ) et [ ].
5. variante du langage précédent où les parenthèses : { } apparaissent au 1er niveau, ( ) au 2ème niveau et [ ] au 3ème niveau (par exemple : {()}{([])()} OK, mais ({}) KO car non respect des niveaux). Ce langage est-il régulier ?

Exercice 5 (Grammaire régulière)
La grammaire suivante n'est pas régulière :

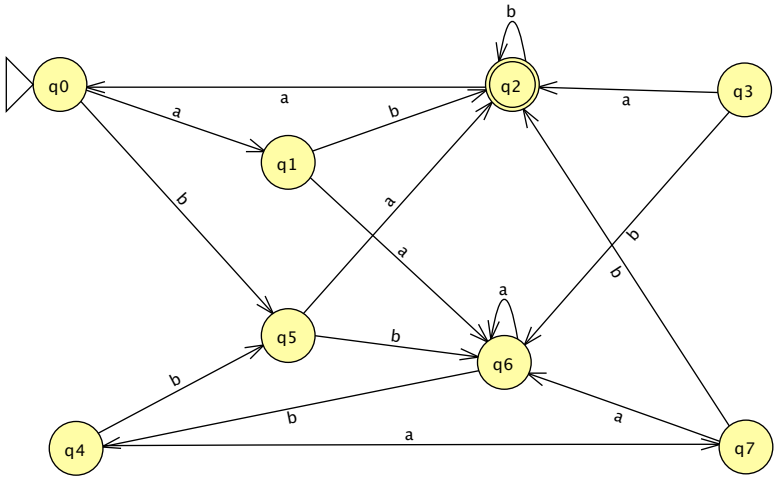
$$S \to S + S \mid S * S$$
$$S \to 0 \mid 1$$

1. Cette grammaire est-elle hors contexte ?
2. Montrer que cette grammaire est ambiguë. Quel problème cela pose-t-il ?
3. Donner une grammaire régulière qui génère le même langage. Est-elle ambiguë ?
4. Est-ce toujours possible lorsqu'on ajoute la règle $S \to (S)$?

Automates finis - TD "minimisation"
Département informatique ENSEIRB-MATMECA 27 février 2024

Voir la page `http://herbrete.vvv.enseirb-matmeca.fr/IF114` pour toute information complémentaire sur cet enseignement.

L'algorithme de minimisation des automates finis vu en cours et étudié dans cette feuille de TD est mis en œuvre dans l'outil JFLAP (voir la page www de l'enseignement pour son utilisation). Il est accessible via le menu `Convert/Minimize DFA`. Nous recommandons d'utiliser cet outil pour renforcer la compréhension de l'algorithme de minimisation et pour s'exercer.

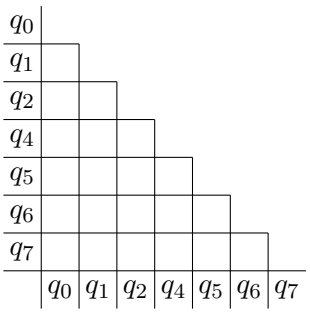


Exercice 1 (Calcul d'automate minimal)
Répondre aux questions ci-dessous pour l'automate ci-dessus.

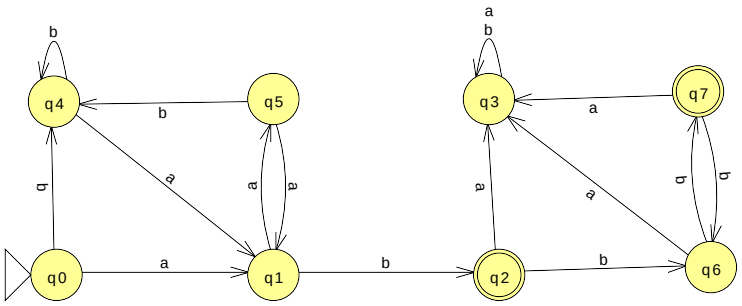
1. Calcul de la partie accessible

Itér.	0	1	2	3	4
Acc	$\{q_0\}$				

2. Calcul des classes d'équivalence $\equiv_Q$



3. Dessin de l'automate minimal



Exercice 2 (Calcul de l'automate minimal)
Répondre aux questions ci-dessous pour l'automate ci-dessus.

1. Calcul de la partie accessible

Itér.	0	1	2	3	4
Acc	$\{q_0\}$				

2. Calcul des classes d'équivalence $\equiv_Q$

q_0								
q_1								
q_2								
q_3								
q_4								
q_5								
q_6								
q_7								
	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7

3. Dessin de l'automate minimal

Automates finis - TD "analyse syntaxique"
Département informatique ENSEIRB-MATMECA 27 février 2024

Voir la page `http://herbrete.vvv.enseirb-matmeca.fr/IF114` pour toute information complémentaire sur cet enseignement.

1 Nombres flottants

Nous souhaitons décider si un entrée saisie au clavier par l'utilisateur représente un nombre flottant ou non. La définition des nombres flottants dans la grammaire du langage C est la suivante :

$$
\begin{aligned}
\mathit{floatnum} &::= \mathit{sign}\ \mathit{value}\ \mathit{exponent} \\
\mathit{sign} &::= \epsilon \mid - \mid + \\
\mathit{value} &::= \mathit{num} \mid \mathit{num}\ .\ \mathit{num} \\
\mathit{num} &::= \mathit{digit} \mid \mathit{digit}\ \mathit{num} \\
\mathit{digit} &::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\mathit{exponent} &::= \epsilon \mid \mathit{expsym}\ \mathit{sign}\ \mathit{num} \\
\mathit{expsym} &::= \texttt{E} \mid \texttt{e}
\end{aligned}
$$

Exercice 1 (Formalisation)

1. Quel est l'alphabet de travail ? Quels sont les symboles qui peuvent apparaître dans la représentation d'un nombre flottant ?
2. Donner une expression régulière qui décrit le langage des nombres flottants
3. Donner l'automate minimal qui reconnaît le langage décrit par l'expression régulière précédemment obtenue.

Exercice 2 (Mise en œuvre)
Mettre en œuvre l'automate fini par un programme C qui décide si une entrée saisie au clavier par l'utilisateur représente un nombre flottant.

Exercice 3 (Évaluation)
Enrichir votre programme pour qu'en plus de décider si l'entrée saisie est bien formée, il calcule le nombre flottant correspondant.

2 Un fichier indexé

Nous souhaitons écrire un programme qui lit en entrée un fichier saisi au clavier par l'utilisateur, puis décide s'il a la structure suivante :

```
A2    12.45
B1    2.0
A1    13.47
C4    1.45
```

```
B27   23.78
C2    4.0
B3    6.8
...
```

Les lignes du fichier définissent les entrées des tableaux `A`, `B` et `C` (dans cet exemple). Chaque ligne a la structure suivante :

- une lettre en première colonne,
- immédiatement suivie d'un nombre (entier naturel),
- puis une séquence non vide d'espaces (`' '` et `'\t'`),
- un nombre flottant,
- et enfin, le caractère "fin de ligne" `'\n'`.

Des espaces supplémentaires peuvent se trouver en début ou en fin de ligne (*i.e.* immédiatement avant `'\n'`. Ils doivent être ignorés. Enfin, la fin du fichier est marquée par la lecture du caractère "Fin de fichier" représenté par la constante symbolique `EOF` (définit dans `stdlib.h` pour le langage C).

Exercice 4 (Formalisation)
Donnez un l'automate fini minimal qui reconnaît le langage des fichiers (vus comme des mots) qui ont la structure ci-dessus.

Exercice 5 (Mise en œuvre)
Mettez en œuvre l'automate fini minimal obtenu à l'exercice précédent par un programme C qui lit le fichier sur son entrée standard et qui indique s'il a la bonne structure ou non.

Exercice 6 (Mémorisation des données)
Enrichissez votre programme pour qu'il mémorise les valeurs des tableaux `A` et `B` uniquement. Sur l'exemple ci-dessus, le tableau `A` contient 12.45 en case d'indice 2 et 13.47 en case d'indice 1. Les cases du tableau pour lesquelles aucune valeur n'a été fournie doivent contenir 0.0. Avant de stocker un élément dans un tableau, vous vérifierez que l'indice donné dans le fichier ne provoque pas de débordement de capacité du tableau. Une fois l'intégralité du fichier lue, votre programme affichera les contenus des tableaux `A` et `B`.