Introduction au module

Version : 2.48

Mise en garde sur ce cours

Ce cours est une première version probablement imparfaite. Aussi, nous demandons à tout lecteur de ne pas hésiter à prodiguer remarques et conseils aussi bien sur le fond que sur la forme.

Présentation du cours

Le module "Langages formels" vise à introduire les bases de la théorie des langages, des automates ainsi que les principales notions sur lescompilateurs. Il permet d'appréhender un certain nombre de techniquesfondamentales :

- de nouvelles techniques de programmation (notion de programmation dirigée par la syntaxe)
- le contrôle de validité
- la compréhension et optimisation de nouveaux outils proposant l'utilisation des expressions régulières (Perl, PHP, JDK 1.4...)
- la présentation de bases pour des techniques de CSI (modélisation dynamique en UML) et de compilation (analyseurs lexicaux).
- ...

Ce cours propose aussi d'habituer l'étudiant à des formalisation et démonstrations utiles pour d'autres cours.

Nous y définissons les notions suivantes : vocabulaire, langage, grammaires, classification de Chomsky, langages relationnels, expressions régulières, machine de Turing, automates déterministes et non-déterministes. Nous présentons aussi bien les bases théoriques nécessaires à la bonne compréhension de ces notions que des algorithmes de base permettant de les manipuler. En particulier,nous nous attachons à présenter les outils permettant de construire un analyseur de chaînes de symboles à partir d'une ou plusieurs expressions décrivant leur construction (le langage).

Plan de ce cours

Nous débuterons ce cours par une présentation des bases de la théorie des langages formels. Nous présenterons les notions de base : alphabet, chaîne, mot, langage... ainsi que les opérations associées sur les mots et les langages. Nous insisterons notamment sur une classe de langage particulière : les langages rationnels. Nous donnerons des définitions et des outils permettant de les décrire dont principalement les expressions régulières. Nous présenterons aussi une méthode permettant de déterminer si un langage n'est pas rationnel, basée sur le lemme de la pompe.

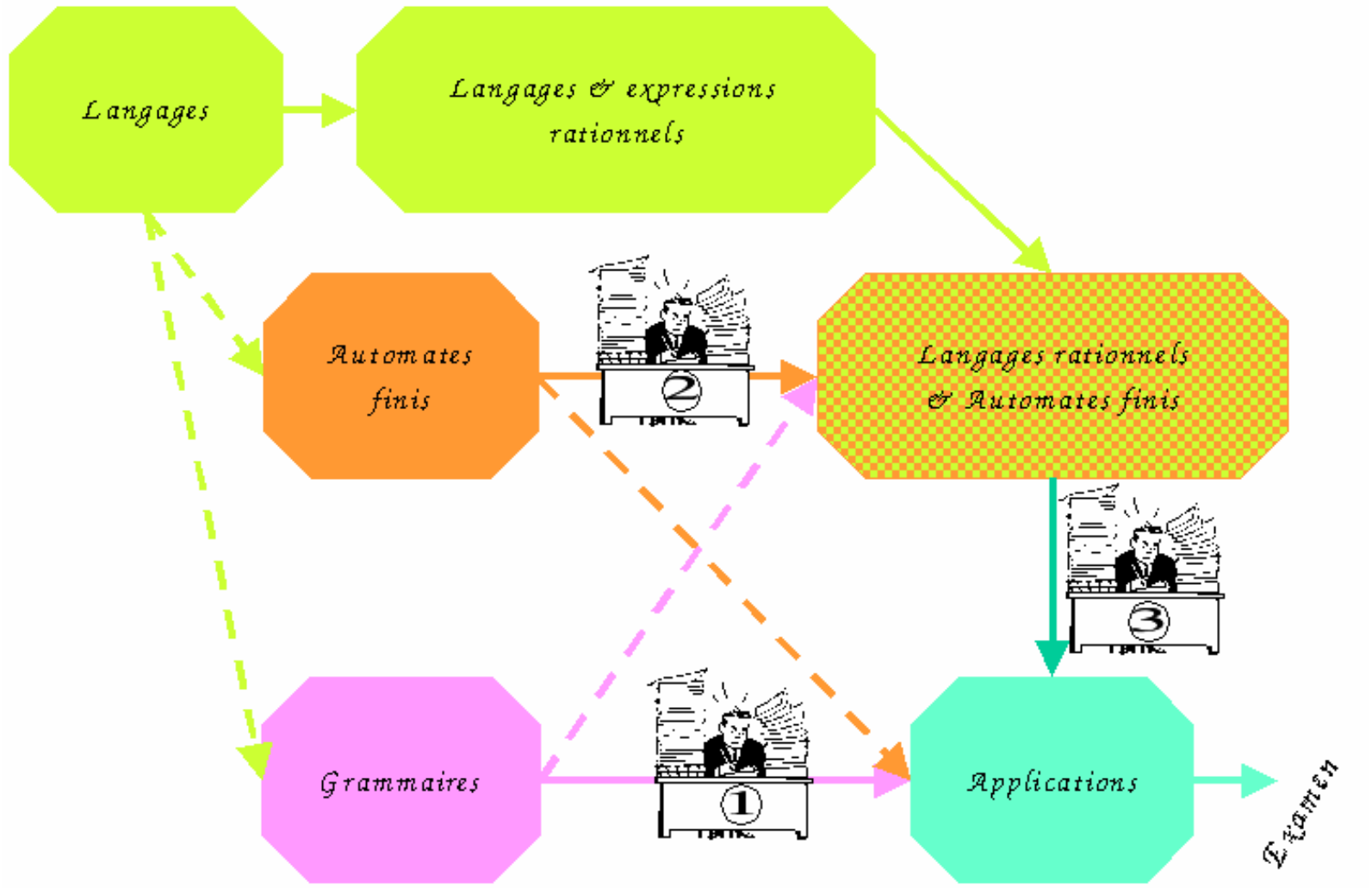
Nous introduirons aussi les grammaires permettant de définir un langage. Une grammaire (ou syntaxe) est un ensemble de règles applicables à un vocabulaire définissant les phrases bien formées du langage. Cette analyse se situe à un niveau purement syntaxique. Une phrase appartenant à un langage peu ne pas avoir de sens d'un point de vue sémantique mais en avoir du point de vue syntaxique. Une grammaire a deux fonctions : produire (des phrases syntaxiquement correctes) et reconnaître (des phrases comme syntaxiquement correctes).

Ensuite, nous introduirons rapidement les machines de Turing avant de nous attarder sur un cas particulier : les automates à nombre fini d'états (appelé aussi Automates d'états finis ou automate fini). Nous donnerons toutes les définitions nécessaires à leur compréhension et à leur utilisation. De plus, nous verrons comment les "optimiser" en les rendant déterministes et minimaux. Nous proposerons aussi une succession d'algorithmes permettant de combiner et transformer ces automates finis.

Après avoir montré l'équivalence entre les langages rationnels et les langages reconnus par les automates finis (théorème de Kleene), nous présenterons aussi des méthodes permettant de passer d'une expression régulière (décrivant un langage rationnel) à un automate fini déterministe et minimal capable de reconnaître des mots de ce langage ainsi que les opérations inverses.

Enfin, nous exposerons des applications possibles des automates finis. Nous verrons en particulier leur utilisation dans le processus de compilation d'un langage informatique et dans le traitement du langage naturel.

Certaines parties sont relativement indépendantes et peuvent donc être abordées en parallèle. Le graphe ci-dessous présente les dépendances entre les grandes parties de ce cours : les flèches pleines représentent des dépendances fortes (il est indispensable d'avoir vu les langages pour aborder les langages rationnels) et les flèches en pointillé sont des dépendances faibles (il n'est pas indispensable mais conseillé de voir les langages avant d'étudier lesgrammaires). Les icônes indiquent les devoirs à effectuer.



Participants

Responsable du module :

- Emmanuel Desmontils (IUP-MIAGE de Nantes)

Participants :

- Emmanuel Desmontils (IUP-MIAGE de Nantes)
- Annie Tartier (IUP-MIAGE de Nantes).

Remerciement :

- à Sophie Coudert pour sa participation à la rédaction du polycopié du module "Automates" en seconde année de l'IUP-MIAGE durant l'année 1998-1999, polycopié qui a servi de base à ce cours.
- à Bengamin Habegger pour sa participation aux TD et TP entre 2000 et 2004 et ses remarques sur les sujets de TD et TP qui sont la base des exercices de ce cours.

- à Alain Vailly (IUP-MIAGE de Nantes) pour ses conseils et son soutien.

Notations

Il y a peu de conventions spécifiques pour ce cours. Cependant, nous essayerons de respecter les notations suivantes :

	indique une définition.
	indique un théorème, parfois suivi d'une démonstration
	indique un lemme, parfois suivi d'une démonstration
	indique une remarque importante.
Exercices et tests	indique une série d'exercices pour tester les connaissances acquises
 ou 	indiquent une aide pour résoudre un exercice
	indique la solution à un exercice

Bibliographie

Les références en gras sont celles qui ont été plus particulièrement utilisées pour la construction de ce cours.

- **A. Aho, R. Sethi & J. Ullman, "Compilateurs : principes, techniques et outils", InterEditions, 1991.**
- P. André, A. Vailly, "Conception des systèmes d'information : Panorama des méthodes et des techniques", série Technosup, Ed. Ellipses, Paris, 2001, ISBN 2-7298-0479-X.
- J.-M. Autebert, "Théorie des langages et des automates", Masson, 1994.
- N. Chomsky, "Three models for the description of language", In Proc. of the Symposium on Information Theory, ED. IRE Trans. Inf. Th., IT-2, 1956
- **Patrick Bellot & Jacques Sakarovitch, "Logique et automates", série Manuel d'Informatique, Ed.**

Ellipses, Paris, 1998, ISBN 2-7298-6894-1
- A. Ginzburg, "Algebraic Theory of Automata", ACM, Academic Press, 1968
- S. C. Kleene, "Representation of events in nerve nets and finite automata", In Automata Studies, C. E. Shannon & J. McCarthy Eds, Princeton University Press, 1956, pp. 3-42.
- **M. Lucas, J.-P. Peyrin & P.-C. Scholl, "Algorithmique et représentation des données : 1 - Files, automates d'états finis", Masson, 1983**
- T. Niemann, "A Guide to LEX & YACC" http://www.informatik.hu-berlin.de/~mueller/codeopt/y_man.pdf
- D. Perrin, "Les débuts de la théorie des automates", TSI, 14:409-443, 1995 www-igm.univ-mlv.fr/~perrin/Recherche/Publications/Loi/copie3.ps
- Charles Platt, "What's It Mean to Be Human, Anyway?", http://www.usyd.edu.au/su/social/papers/platt.html
- **Patrice Séébold, "Théorie de automates : méthodes et exercices corrigés", série Passeport pour l'informatique, Ed. Vuibert, Paris, 1999, ISBN 2-7117-8630-7, www.vuibert.fr**
- K. Thompson, "Regular expression search algorithm", Communication of the ACM, 11(6):419-422, 1968
- A. Turing, "On computable numbers with an application to the entscheidungs problem". Proceedings of the London Math. Soc., 42:230-265, 1936
- B. W. Watson, "A taxonomy of finite automata construction", Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands, 1993 ftp://ftp.win.tue.nl/pub/techreports/pi/automata/taxonomy/2nd.edition/constax.ps.Z
- B. W. Watson, "A taxonomy of finite automata minimization algorithms", Computing Science Note 93/44, Eindhoven University of Technology, The Netherlands, 1993 ftp://ftp.win.tue.nl/pub/techreports/pi/automata/taxonomy/2nd.edition/mintax.ps.Z
- R. Wilhelm & D. Maurer, "Les compilateurs : théorie, construction, génération", Masson, 1994

Cours Web

- G. Coray, "Automates et Calculabilité II", Semestre d'été - 2003, http://lithwww.epfl.ch/teaching/tac/
- G. Falquet, "Outils formels pour les systèmes d'information (hiver 00-01)", http://cui.unige.ch/isi/ofsi00/
- A. Felty,"CSI 3504 Introduction aux langages formels", Hiver 2003, http://www.site.uottawa.ca/~afelty/courses/csi3504/
- S. Gire, " COMPILATION - THÉORIE DES LANGAGES", http://fastnet.univ-brest.fr/~gire/COURS/COMPIL_IUP/POLY.html
- S. Julia, "Automates & Langages", http://deptinfo.unice.fr/%7ejulia/L1/
- L. Lander, "CS573 : Automata Theory and Formal Languages", Thomas J. School of Engineering and Applied Science, Binghamton University, University of New York, 1998 http://bingweb.binghamton.edu/~lander/cs573.html (lien devenu invalide)
- David Matuszek, "Theory of Computation", http://www.netaxs.com/people/nerp/automata/syllabus.html
- M. Pichat & C. Solnon, "Théorie des Langages", http://www710.univ-lyon1.fr/~csolnon/langages.html
- H. Shapiro, "CS500 : Introduction to the Theory of Computation", The University of New Mexico, 1997 ftp://ftp.cs.unm.edu/pub/shapiro/CS500/chapter3.ps (lien devenu invalide)
- A. Tisseran, "QUELQUES ELEMENTS DE THEORIE DES LANGAGES", http://www.mines.u-nancy.fr/~tisseran/cours/langages/
- Turing's World, http://www-csli.stanford.edu/hp/Turing1.html
- ... Et il y en a bien d'autres !

Plus généralement, des bases de cours : http://rangiroa.essi.fr/cours/ & http://rangiroa.essi.fr/specif/spedago/index.html

Outils

- JFLAP, Susan H. Rodger, http://www.cs.duke.edu/~rodger/tools/jflap/
  Cet outil sera régulièrement utilisé pour illustrer cours et exercices. Il est aussi en local ici.
- Applet tracé d'automates, http://lithwww.epfl.ch/teaching/tac/AEF/DFApplet.html
- Eileen Head, http://www.cs.binghamton.edu/~software/

E. Desmontils (IUP-MIAGe de Nantes)
Last modified: Thu Jan 27 11:45:15 CET 2005

Généralités sur les langages

1. Introduction

Les langages peuvent prendre des formes très diverses selon la nature de ce qu'ils expriment : langage naturel, langage mathématique, langage de programmation (Pascal, C...), langages formels...

Une introduction aux langages formels permet, entre autre :

- d'introduire des notions de base pour la compréhension d'outils comme :
  - Lex, Yacc,
  - PERL,
  - PHP...
- la compréhension, l'utilisation et la construction des compilateurs (analyse lexicale et syntaxique)
- l'exploitation de méthodes liées au traitement automatique des langues
- ...

En fin des années 50, le linguiste Noam CHOMSKY propose une première formalisation de la description des langages formels. Un langage formel est constitué d'un vocabulaire et de règles de grammaires entièrement connus. La théorie des langages formels s'occupe soit de générer l'ensemble des phrases (mots) d'un langage soit de déterminer si une phrase appartient ou non au langage. L'analyse ou la génération de phrases est, dans ce contexte, purement syntaxique. Elle est indépendante de la signification des symboles utilisés et des mots produits et/ou reconnus.

Dans cette section nos présentons brièvement la structure de monoïde qui est la base de la représentation mathématique des mots. Ensuite nous définissons les mots comme les éléments de monoïdes particuliers construits sur des ensembles de base appelés alphabets. Enfin, nous définissons les langages comme des ensembles de mots précis. Et, sur les langages, sont définies des opérations de combinaison qui permettent d'en obtenir de nouveaux.

2. Notion de monoïde

monoïde et monoïde libre

Un **monoïde** est un ensemble E muni d'une opération binaire interne associative ⊕.

Un **monoïde libre** est un monoïde possèdant un élément neutre ε. Il est parfois noté <E, ⊕, ε>

Par exemple, <N, +, 0> est un monoïde, où N est l'ensemble des entiers naturels. En effet, + est associative et 0 est neutre pour +. De même, <N, *, 1> est un monoïde.

sous-monoïde

Soit <E, ⊕, ε> un monoïde et T une partie de E. <T, ⊕, ε> est un **sous-monoïde** de <E, ⊕, ε> si c'est un monoïde, c'est-à-dire si ε ∈ T et ∀ t, t'∈ T, t ⊕ t' ∈ T

Par exemple, <P, +, 0> est un sous-monoïde de <N, +, 0>, où P est l'ensemble des entiers naturels pairs.

monoïde engendré

Soit M =<E, ⊕, ε> un monoïde. Pour toute partie A de E on peut définir le plus petit sous-monoïde de M contenant A. On l'appelle le **sous-monoïde** de M **engendré** par A.

Par exemple, le sous-monoïde de <N, +, 0> engendré par {3} est l'ensemble des naturels multiples de 3 (de même que celui engendré par {3,6}).

Remarque : Soit M=<E, ⊕, ε> un monoïde et A une partie de M. On peut montrer que le sous-monoïde de M engendré
par A est $\cup_{n=0..\infty} A^n$, où $A^0=\{\varepsilon\}$ et $\forall n \in N$ alors $A^{n+1} = A \oplus A^n = \{x \oplus y \mid x \in A, y \in A^n\}$

Exercices et tests :

Exercice 2.1. Indiquer si les ensembles suivants sont des monoïdes ? des monoïdes libres ? (N est l'ensemble des entiers naturels, Pair(N) est l'ensemble des entiers pairs et Impair(N) l'ensemble des entiers impairs)

1. <Pair(N), +, 0>

2. <Impair(N), +, 0>
3. <Pair(N), *, 1>
4. <Impair(N), *, 1>

3. Mots et monoïdes libres

Intuitivement, un mot est un ensemble de symboles d'un alphabet placés les uns après les autres donc une suite de lettres (ou chaîne de caractères, dans une terminologie plus informatique).

Symbole, Alphabet, Mots & Mot vide

Un **symbole** est une brique élémentaire, un atome.

Un **alphabet** A est un ensemble fini et non vide de symboles.

Un **mot** (ou **chaîne**) sur un alphabet A est une suite finie de lettres de A. Si [p] désigne la suite d'entiers (1, 2, ..., p), un mot m est une application de [p] dans A (m : [p] → A) pour un certain p de N. p est appelé la **longueur** du mot m, notée |m|. Intuitivement, p est le nombre de symboles du mot.

Le **mot vide** est le mot de longueur 0, c'est-à-dire ne contenant aucun symbole. Il est souvent noté ε.

Remarque : Dans la suite, les alphabets seront décrits comme des ensembles de symboles (entre accolades), éventuellement composés de plusieurs caractères. Les symboles représentés par des signes de ponctuation seront entre quotes.

Par exemple, A_1 = {1}, A_2 = {0, 1}, A_3 = {., -, /}, A_4 = {0, 1, 2, ..., 9}, A_5 = {a, b, ..., z}, A_6={a,b, ch,';',',','} et A={a,b,c} sont des alphabets et "abbc", "aaaa", "bc", "ε" sont des mots sur l'alphabet A, respectivement de longueur 4, 4, 2 et 0.

Soit l'alphabet {a,b,c,h,ch,';',',','}. Les mots "ch" (|ch|=1) et "c ⊕ h" ≡ "ch" (|c ⊕ h|=2) sont théoriquement différents. Il n'y a donc (théoriquement) pas d'ambiguïté sur le mot "ch". En pratique, cette ambiguïté est à lever. Dans le suite de ce cours, nous prendrons volontairement des alphabets qui ne présenterons pas cette ambiguïté.

Par convention (valable dans la suite du document), nous posons :

- Que a, b, c, d, e, f, 0, 1... désignent des symboles.
- Que t, u, v, x, y, z désignent des chaînes.
- La chaîne aaaa est aussi notée a^4. Par conséquent, $a^0=\varepsilon$.
- Le symbole m(i) désigne le i-ème symbole de la chaîne m.

Dans la suite, nous définissons les mots comme les éléments de monoïdes particuliers. On peut définir sur les mots (ou chaînes) une opération particulière : la **concaténation**. Intuitivement, si x et y sont des chaînes, alors le résultat de la concaténation de x et y est la chaîne xy. Cette opération est associative : (xy)z=x(yz)=xyz (Les parenthèses sont ici des symboles mathématiques et n'appartiennent pas à l'alphabet de référence). De plus, la chaîne vide ε est un élément neutre pour la concaténation. L'ensemble des mots sur un alphabet donné peut ainsi être muni d'une structure de monoïde . Ce type de monoïde, un peu particulier, est appelé monoïde libre.

Concaténation & monoïde libre

Étant donné un alphabet A :

- On note A^* l'ensemble de tous les mots sur A.
- Dans A^*, on définit une opération dite **produit de concaténation** ou simplement **concaténation**. Elle est noté ⊕ ou plus souvent implicite (simple juxtaposition des opérandes).
- Cette opération binaire est une loi de composition interne associative définie de la façon suivante : soit u et v deux mots de longueur p et q, la concaténation de u et v (dans cet ordre) est le mot w de longueur p+q défini par w(i)=u (i) si $1 \leq i \leq p$ et w(i)=v(i) si $p+1 \leq i \leq q$. Autrement dit, w est construit par juxtaposition des mots u et v, c'est-à-dire w = u ⊕ v = uv.
- ε est un élément neutre pour la concaténation.
- Le monoïde $\langle A^*, \oplus, \varepsilon \rangle$ est appelé le **monoïde libre engendré** par A.

Par exemple, si x=ab et y=cd, alors x ⊕ y = xy = abcd. De plus, x ⊕ ε = ε ⊕ x = x et, par exemple, (ab ⊕ cd) ⊕ ef = abcd ⊕ ef = abcdef = ab ⊕ (cdef) = ab ⊕ (cd ⊕ ef).

Remarques :

- L'ensemble A^* des mots ainsi construit est infini dénombrable.
- Les symboles de l'alphabet sont atomiques, ie : si $a \in A$, $\forall\ x,y \in A^+ : a \neq xy$
- Il n'y a pas d'inverse pour l'opération de concaténation, ie : si $u \in A^+$, $\forall\ v \in A^+ : uv \neq \varepsilon$

Théorème de la décomposition

Tout mot x (de longueur n=|x|) sur un alphabet A se décompose de façon unique en $x_1 \oplus x_2 \oplus \ldots \oplus x_n$, également noté $x_1x_2\ldots x_n$, où $\forall i \in [1..n]$, $x_i \in A$.

Démonstration

En deux mots, comme les symboles de l'alphabet sont des atomes, il n'est pas possible de diviser un mot plus qu'au niveau de ces symboles. Si deux divisions en symboles étaient possibles, cela voudrait dire qu'un symbole peut être divisé et donc il ne serait pas atomique.

Ce théorème sert de base à beaucoup de démonstrations par induction sur la longueur des mots.

Par exemple, avec A = {a,b,c} alors abbcabbc = a ⊕ b ⊕ b ⊕ c ⊕ a ⊕ b ⊕ b ⊕ c.

Remarque : Une chaîne sur un alphabet A peut également être définie récursivement par :

- la chaîne vide ε est une chaîne sur A;
- si x est une chaîne sur A et $a \in A$ alors la chaîne ax et xa sont des chaînes (mots) sur A (Attention, dans le cas général, $ax \neq xa$);
- aucun autre objet n'est une chaîne sur A.

Cette définition sert de base pour la démonstration de nombreuses propriétés à propos des mots, par récurrence (induction) sur leur longueur. L'idée générale est que pour montrer qu'une propriété P est vraie pour tous les mots, il suffit de montrer que P est vraie pour ε (cas de base) et que si P est vraie pour un mot x (hypothèse de récurrence), alors elle est vraie pour ax, ∀ a.

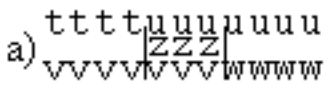
Théorème de Levi

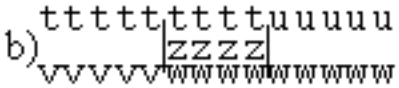
Soit t,u,v,w ∈ A^*, si tu=vw alors il existe un unique (∃!) z ∈ A^* tel que :

a. (v=tz et u=zw)
b. ou (t=vz et zu=w)

Démonstration :

Pour bien comprendre ce théorème, comme souvent, un petit schéma s'impose. t, u, v, w et z correspondent aux situations suivantes :





Pour le cas a), "zzz" correspond à ce qu'il faut ajouter aux "t" pour atteindre la longueur des "v". Le cas b) est symétrique.

Plus formellement, la démonstration se base sur une induction sur la longueur de "tu" (|tu|).

1. **Cas de base** :
    1. **|tu| = 0**, alors |uv| = 0 donc t=u=v=w=z=ε : CQFD
    2. **|tu| > 0 et t =** ε : u = vw et v=z (cas a) : CQFD
    3. **|tu| > 0 et v =** ε : tu = w et t=z (cas b) : CQFD
2. **|tu| > 0 et t** ≠ ε **et v** ≠ ε
    1. **Hypothèse d'induction** : t'u=v'w, ∃ z ∈ A^* tel que (v'=t'z et u = zw)

ou (t'=v'z et zu=w).
2. Par hypothèse, |tu| > 0 et t ≠ ε donc t=xt' (x ∈ A) et tu=xt'u
3. De plus, |tu| > 0 donc |vw| > 0 (tu=vw) et v ≠ ε donc v=yv' (y ∈ A) et vw=yv'w
4. Comme tu=vw alors xt'u=yv'w
5. Selon le théorème d'unicité de la décomposition en symboles de l'alphabet, comme xt'u=yv'w alors x=y et donc t'u=v'w
6. Donc t=xt'=xv'z=vz (cas a) ou v=xv'=xt'z=tz (cas b) : CQFD

Notations :

- L'ensemble des mots (ou chaînes) de longueur n (pour $n \in N$) construits sur A est noté A^n ($A^n = \{x_1x_2...x_n \mid \forall i, 1 \leq i \leq n, x_i \in A\}$) . Par conséquent, toutes les chaînes y de l'ensemble A^3 ont pour longueur |y|=3.
- L'ensemble des chaînes de longueur finie et non nulle définies sur A est noté $A^+ = \cup_{n=1..\infty} A^n = \{x \mid \exists n \in N, n \geq 1 \text{ et } x \in A^n\}$.
  Notons que $A^* = A^+ \cup \{\varepsilon\}$
- Par convention, A^0 est ε et donc $A^* = \cup_{n=0..\infty} A^n$

Par exemple, si l'on considère l'alphabet A= {0,1} alors :

- $A^0 = \{\varepsilon\}$;
- $A^1 = A = \{0, 1\}$;
- $A^2 = \{0,1\}^2 = \{00, 01, 10, 11\}$;
- $A^3 = \{0,1\}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$;
- ...
- $A^* = \{0,1\}^* = \{\varepsilon, 0,1, 00,01,10, 11,000, 001, 010, 011,..., 11111, ..., 1001101110...\}$.
- $A^+ = \{0,1\}^+ = \{0,1, 00,01,10, 11,000, 001, 010, 011,..., 11111, ..., 1001101110...\}$.
- si $S = \{\varepsilon,0, 00, 000, 0000, ...\} = \{0^i\} \subset A^+$ (avec $\varepsilon=0^0$) alors $\langle S,\oplus,\varepsilon\rangle$ est le sous-monoïde de $\langle A^*,\oplus,\varepsilon\rangle$ engendré par {0}.

Remarque : Si $c_1 \in A^n$ et $c_2 \in A^m$ alors $c_1 \oplus c_2 \in A^{n+m}$.

Exercices et tests :

Exercice 3.1. Donner la longueur des mots suivants sur l'alphabet {a,b,c} :

1. abcb
2. abba
3. ε
4. bb

Exercice 3.2. Donner la longueur des mots suivants sur l'alphabet {a,b,ch,','} :

1. a,bbaa
2. achbbaa

Exercice 3.3. Soient a, b des symboles de A et u un mot de A*. Montrer que si ua=bu alors a=b et u ∈ a*. 

4. Relations entre mots

Ici, nous présentons quelques relations classiques entre mots. Tout d'abord, intuitivement, un mot est préfixe (respectivement suffixe puis sous-chaîne) d'un autre mot, s'il en est un début (respectivement une fin, puis un morceau).

Préfixe, suffixe et sous-chaîne

Si x et y sont deux chaînes quelconques sur A, alors

- x est un **préfixe** de y ($x \in Pref(y)$) s'il existe z dans A^* tel que y = xz, autrement dit $x \in Pref(y)$ si et seulement si $\exists z \in A^*, y = xz$;
- x est un **suffixe** de y ($x \in Suff(y)$) s'il existe z dans A^* tel que y = zx, autrement dit $x \in Suff(y)$ si et seulement si $\exists z \in A^*, y = zx$;
- x est une **sous-chaîne** (ou **facteur**) de y s'il existe w et z dans A^* tel que y = wxz, autrement dit $x \in Fact(y)$ si et seulement si $\exists w, z \in A^*, y = wxz$.

Si $x \neq y$ (c'est-à-dire si $z \neq \varepsilon$), alors le préfixe ou le suffixe est dit **propre**.

Par exemple, dans la chaîne 001110, 00 est un préfixe, 10 est un suffixe et 0111 est une sous-chaîne.

Remarques :

- De manière plus générale, x est un préfixe (**facteur gauche**) de $x \oplus y \oplus z$, z en est un suffixe (**facteur droit**) et y en est une sous-chaîne.
- x, comme y et z peuvent très bien être vides,
- ε est préfixe, suffixe et sous-chaîne de toute chaîne.
- $Pref(x) \subseteq Fact(x)$ et $Suff(x) \subseteq Fact(x)$

Occurrence

Une **occurrence** d'une chaîne x dans une chaîne y est une apparition de x à un endroit précis dans la chaîne y.

Par exemple, avec A = {a,b}, le mot y = abababababa comporte quatre occurrences de la sous-chaîne x = aba.

Par ailleurs, notons que dans A^* défini sur $A = \{x_1, x_2, ..., x_n\}$, il est possible de définir plusieurs relations d'ordre :

- l'**ordre préfixiel**, ordre partiel défini par : u < v si et seulement si u est un préfixe propre de v ;

- l'**ordre lexicographique** (ordre du dictionnaire), ordre total défini par : $u < v$ si et seulement si $u = wau_2$ et $v = wbv_2$ tels que $w \in A^*$, $a < b$ avec a et $b \in A$; Autrement dit $u<v$ si et seulement si $u=aw$, $v=bz$ et $(a<b)$ ou $((a=b)$ et $(w<z))$, $a, b \in A$ et $w,z \in A^*$ avec $\varepsilon < t$ $\forall t \in A^+$.
  Tout ceci est valable si et seulement si $\forall x_i, x_j \in A$, $x_i \leq x_j$ si $i \leq j$.
- l'**ordre hiérarchique**, ordre total pour lequel les mots sont classés en premier lieu par longueur, puis pour les mots de même longueur, par ordre lexicographique.

Exercices et tests :

Exercice 4.1. Soit x=abbcc un mot sur l'alphabet A={a,b,c} :

1. Donner l'ensemble Pref(x)
2. Donner l'ensemble Suff(x)

Exercice 4.2. Soient u_1, u_2 et v trois mots de A*. Montrer que si $u_1 \in$ Pref(v) et $u_2 \in$ Pref(v) alors soit $u_1 \in$ Pref(u_2), soit $u_2 \in$ Pref(u_1). 

5. Langages et opérations

Langage
- version 1 -

Un langage L sur un alphabet A est un ensemble de chaînes (ou ensemble de mots) sur A. L est donc un sous-ensemble de A*, autrement dit $L \subseteq A^*$.

Par conséquent, l'ensemble des langages L sur A est l'ensemble $P(A^*)$ des parties de A^*, autrement dit : $L \in P(A^*)$.

Par exemple, si l'on considère l'alphabet A = {0, 1} alors L_1 = {0, 00, 1, 01, 11, 10 } est un langage sur A. De même, pour tout entier naturel n, A^n est un langage sur A, où, rappelons le, A^n est l'ensemble des mots de longueur n sur A. Si de plus, pour toute lettre a, a^n

désigne le mot formé de n symboles a consécutifs, alors $L_2 = \{ 0^n1^n \mid n \geq 0\}$, $L_3 = \{0^n10^m \mid n \geq 0, m \geq 1\}$, $L_4 = \{1^n \mid n \geq 2\}$ et $L_5 = \{0^i \mid i \geq 0\} = \{0\}^*$ sont d'autres exemples de langages sur A.

Parmi tous les langages sur un alphabet A donné, on en distingue quelques uns particuliers, dont par exemple les suivants.

Langage prefixe ou suffixe

Étant donné un alphabet A, parmi tous les langages L de $P(A^*)$:

- Le **langage neutre** est celui dont le seul mot est la chaîne vide : $L = \{\varepsilon\}$.
- Le **langage vide** est celui qui ne contient aucun mot, soit $L = \varnothing$.
- Un **langage fini** est un langage qui contient un nombre fini de mots.
- Un **langage infini** est un langage non vide et non fini.
- Un **langage** L est dit posséder la propriété **préfixe** (resp. **suffixe**) si aucune chaîne de L n'est préfixe propre (resp. suffixe propre) d'une autre chaîne de L.

Par exemple, $L = \{a^ib \mid i \geq 0 \} = \{b, ab, aab, aaab...\}$ est dit préfixe mais n'est pas suffixe. $L = \{a^n \mid n \in N\}$ n'est ni l'un ni l'autre.

Notons par ailleurs que $\varnothing \neq \{\varepsilon\}$.

Comme les langages sont des ensembles, on peut leur appliquer les opérations ensemblistes classiques : union, intersection, complémentation, etc. De plus, par extension de la concaténation des mots aux langages, on peut définir quelques autres opérateurs.

Union, Intersection, Différence, Complémentaire

Soit A un alphabet. On définit sur les langages de $P(A^*)$ les opérateurs suivants : soient L et M deux langages sur A,

Opérateurs ensemblistes classiques :

- **union** : $L \cup M = \{x \mid x \in L \text{ ou } x \in M\}$;
- **intersection** : $L \cap M = \{x \mid x \in L \text{ et } x \in M\}$;
- **différence** (ou exclusion) : $L \setminus M = L-M = \{x \mid x \in L \text{ et } x \notin M\}$;
- **complémentaire** sur A* : $Comp(L)= A^* \setminus L = \{x \mid x \in A^* \text{ et } x \notin L\}$;

Opérateurs induits par la concaténation des mots :

- **produit** des langages : $LM = L \times M = \{xy \mid x \in L \text{ et } y \in M\}$;
- **fermeture** de Kleene : $L^* = \cup_{i=0..\infty} L^i$ où $L^0 = \{\varepsilon\}$ et $L^n = LL^{n-1} = L^{n-1}L$;
- **fermeture positive** : $L^+= \cup_{i=1..\infty} L^i$.

Par extension, le produit est parfois appelé "concaténation de deux langages". Cette concaténation est notée × (et le symbole × est souvent omis), mais il s'agit bien de deux concaténations différentes : l'une entre mots et l'autre entre ensembles de mots. Intuitivement, la concaténation de deux langages est l'ensemble des mots obtenus en concaténant un mot du premier langage avec un mot du second. Par exemple, si L_1={a,bc} et L_2={de,f} alors $L_1 \times L_2$={ade, af, bcde, bcf}.

Sur ces opérateurs entre langages, on a, entre autres, les quelques propriétés suivantes :

- Le langage vide est absorbant pour la concaténation des langages : $\varnothing L = \varnothing = \varnothing L$
- <P(A*), ×, {ε}> est un monoïde libre :
  - Le langage neutre est élément neutre pour la concaténation des langages : $\{\varepsilon\} L = L = L\{\varepsilon\}$
  - La concaténation des langages est associative : $(L_1L_2)L_3 = L_1(L_2L_3)$
- $L^+ = LL^* = L^*L$ et $L^* = \{\varepsilon\} \cup L^+$
- $\varnothing^* = \{\varepsilon\}$ et $\{\varepsilon\}^* = \{\varepsilon\}$

Jusqu'ici, un langage sur un alphabet A pouvait être n'importe quelle partie dans P(A*) , ce qui est très vaste. En particulier, au moment de vérifier si un mot appartient à un

langage, il est utile d'avoir une caractérisation précise de ce langage, ce qui n'est pas toujours évident si l'on considère une partie infinie quelconque dans P(A*) . On va donc, d'un point de vue pragmatique, s'intéresser à des classes de langages particuliers pour lesquels on a des descriptions finies, utilisables pour décider si oui ou non, un mot est dans un langage. Parmi ces types de langages, on s'intéresse en premier lieu aux langages rationnels.

Exercices et tests :

Exercice 5.1. Montrer que le produit de deux langages préfixes est un langage préfixe.

Bibliographie

Les langages rationnels

1. Introduction

Parmi tous les langages constitués par les parties d'un monoïde libre A* engendrés par un alphabet A, on distingue une classe particulière : la classe Rat(A*) des langages rationnels sur A. L'intérêt de cette classe est qu'il existe une méthode simple permettant de décider pour chacun de ses langages si un mot lui appartient ou non : les automates finis sur lesquels nous reviendrons plus loin dans ce cours.

2. Langages rationnels

Les langages rationnels sont une classe de langages (notée Rat(A*)) définie de manière inductive sur P(A*) en utilisant uniquement les trois opérations $\cup$, $\times$ et *.

Langage rationnel/ réguliers

Soit A un alphabet. Les **langages rationnels sur A** (appelés aussi **langages réguliers**) sont les éléments de la classe Rat(A*) définie inductivement de la façon suivante : Rat(A*) est le plus petit sous-ensemble de P(A*) tel que :

1. $\varnothing \in Rat(A^*)$;
2. $\{\varepsilon\} \in Rat(A^*)$;
3. $\forall a \in A, \{a\} \in Rat(A^*)$;
4. Si $L_1 \in Rat(A^*)$ et $L_2 \in Rat(A^*)$ alors $L_1 \cup L_2 \in Rat(A^*)$;
5. Si $L_1 \in Rat(A^*)$ et $L_2 \in Rat(A^*)$ alors $L_1 \times L_2 \in Rat(A^*)$;
6. Si $L_1 \in Rat(A^*)$ alors $L_1^* \in Rat(A^*)$.

Par exemple, si l'on considère l'alphabet A={a,b,c}, alors l'ensemble des mots qui ne sont formés que de a est dans Rat(A*). Ce langage est {a}*. On peut l'obtenir à partir des points 3, puis 6 de la définition qui précède. De même, $\{a\} \cup \{b\} \in Rat(A^*)$, $\{b\} \times \{a\} \in Rat(A^*)$... L'ensemble des mots sur A commençant par un b est également un langage de Rat (A*). Ce langage est $\{b\} \times (\{a\} \cup \{b\} \cup \{c\})^*$. Il est obtenu en appliquant successivement les points 3 (3 fois), 4 (2 fois), 6 (1 fois) et 5 (1 fois).

Théorème des parties finies

Toute partie finie L de A* est dans Rat(A*).

Idée de démonstration :

Comme toute partie finie peut être vue comme la réunion de singletons, L peut être obtenu par un certain nombre d'applications des points 1 à 5 (démonstration par induction des langages rationnels, en particulier sur le cardinal de L) et en utilisant le théorème de décomposition unique d'un mot.

Par exemple, le langage L={0,1,00,11,000,111} sur l'alphabet {0,1,2} est-il un langage rationnel ? Il est construit par l'union de 6 langages rationnels {0}, {1}, {00}, {11}, {000} et {111}. En effet, les deux premiers sont des cas de base pour la définition (point 3) et les 4 suivants correspondent au produits de langages rationnels (par exemple, {00} = {0} × {0}). Or, la définition des langages rationnels indique que l'union de deux langages rationnels est un langage rationnel (point 4) et que le produit de deux langages rationnels est un langage rationnel. Donc L est bien rationnel.

On peut donc construire à l'aide des opérateurs de la définition précédente de nombreux langages désignés par des expressions mathématiques comme par exemple L={(({a}∪{b})*×{a})*×({b}×{c})*}, ce qui, en appliquant les définitions peut se simplifier en L={({a,b}*×{a})*×{bc}*}. Cette écriture toutefois reste lourde, en particulier à cause des nombreuses accolades qu'elle comporte. Il existe donc pour les langages rationnels une écriture simplifiée, plus compacte et pratique : c'est ce qu'on appelle les expressions rationnelles. Chaque expression rationnelle décrit un langage rationnel précis.

Exercices et tests :

Exercice 2.1. Soit l'alphabet A={0,1}, le langage décrivant les symboles de la table ASCII en binaire est-il un langage rationnel ?

Exercice 2.2. Soit l'alphabet A={0,1}, le langage décrivant les chaînes de caractères

(en Pascal ou C par exemple) en binaire est-il un langage rationnel ?

3. Expressions rationnelles

Expression rationnelle

Les **expressions rationnelles** sur A décrivent les **langages rationnels**. Elles sont définies de la façon suivante :

1. $\varnothing$ est une expression rationnelle qui décrit le langage rationnel $\varnothing$;
2. ε est une expression rationnelle qui décrit le langage rationnel $\{\varepsilon\}$;
3. pour tout a de A, a est une expression rationnelle qui décrit le langage rationnel {a} ;
4. Si l_1 et l_2 sont des expressions rationnelles qui décrivent L_1 et $L_2 \in Rat(A^*)$ alors :
   a. $(l_1 \mid l_2)$ et $(l_1 + l_2)$ sont des expressions rationnelles identiques qui décrivent le langage rationnel $L_1 \cup L_2 \in Rat(A^*)$,
   b. $(l_1 l_2)$ et $(l_1.l_2)$ sont des expressions rationnelles identiques qui décrivent le langage rationnel $L_1 \times L_2 \in Rat(A^*)$,
   c. $(l_1)^*$ est une expression rationnelle qui décrit le langage rationnel $(L_1)^*$.

Attention, dans le point "2" de la définition précédente, le symbole ε représente deux notions différentes. Le premier est un symbole d'une expression rationnelle alors que le second symbolyse l'ensemble contenant uniquement le mot vide. Cependant, il y a isomorphisme entre ces deux notions.

Comme la réunion et la concaténation sont associatives, on omettra généralement les parenthèses inutiles. On écrira par exemple (u | v | w) plutôt que ((u |v) | w) ou (u | (v | w)) et de même pour la concaténation. Bien sûr, il est sous-entendu qu'aucun des symboles "|", "*", "ε", "(" et ")" n'appartient à l'alphabet A. Sinon, il faut soit les différencier en renomant les symboles, soit les entourer de guillemets.

Par exemple, le langage L={(({a}∪{b})*×{a})*×({b}×{c})*} peut être désigné par l'expression rationnelle((a|b)*a)*(bc)*.

Théorème des expressions rationnelles

Toute langage dénoté par une expression rationnelle est un langage rationnel.

En résumé, étant donné un alphabet A et C={ |, *, ε, (,)}, tel que A∩C=∅ :

- Un élément w de (A ∪ C)* est une expression rationnelle si et seulement si :
  - w ∈ A* ;
  - w est de la forme w_1*, ou w_1w_2 ou w_1 | w_2
    avec w_1 et w_2 des expressions rationnelles.
- Intuitivement, une expression rationnelle désigne un langage rationnel de la façon suivante :
  - w ∈ A* désigne le langage dont le seul mot est w (w peut être ε)
  - w_1w_2 désigne le langage dont chaque mot est obtenu en concaténant (juxtaposant) un mot du langage désigné par w_1 avec un mot du langage désigné par w_2 (dans l'ordre).
  - w^* désigne le langage dont chaque mot est obtenu en concaténant un nombre quelconque (éventuellement nul) de mots du langage désigné par w.
  - $w_1|w_2$ (parfois notée w_1+w_2) désigne la réunion des langages désignés par w_1 et w_2. Les mots de $w_1|w_2$ sont ceux qui appartiennent à w_1 ou à w_2.
- Et pour des raisons pratiques, on se donne quelques notations complémentaires :
  - $w^+ = w\,w^* = w^*\,w$;
  - w? signifie 0 ou 1 instance de w. Donc w? est une abréviation pour w | ε;
  - la notation [a, b, c], où a, b et c sont des symboles, abrège l'expression rationnelle a|b|c. Une classe de caractères comme [a-z] abrège l'expression rationnelle a | b | ... | z.
  - w^n, pour un n ≥ 0 désigne la chaîne composée de n occurrences du mot du langage désigné par w.
    Si n est quelconque, w^n (n ≥ 0) ≡ w* et w^n (n > 0) ≡ w^+.

Par exemple, 01 désigne le langage {01}, 0* désigne le langage {0}* et (0|1)*011 désigne le langage ne comportant que des chaînes se terminant par 011 sur l'alphabet {0,1}. Comme exemple plus complexe, on peut donner [A-Z,a-z][A-Z,a-z,0-9]* qui décrit la syntaxe des identificateurs en Pascal : des mots commençant par une lettre, majuscule ou minuscule

([A-Z,a-z]) et suivit d'une suite quelconque de caractères parmi les lettres et les chiffres ([A-Z,a-z,0-9]*).

 Notons enfin que si w^n désigne bien un langage rationnel quand n est un entier fixé, par contre, ce n'est pas toujours vrai dans le cas général, quand n est une variable. En effet, par exemple, $(ab)^3$ désigne ababab qui est une expression rationnelle, de même que $[a,b]^2$ qui désigne (a|b)(a|b). Par contre le langage désigné par a^nb^n n'est pas rationnel : il s'agit des mots commençant par un certain nombre de a et suivis **du même nombre** de b, et ce langage ne peut pas être obtenu à partir des parties finies d'un alphabet A par simple application des opérations ×,∪ et *. Ceci illustre bien le fait que tout langage sur A n'est pas nécessairement rationnel. Il est possible de démontrer qu'un **langage n'est pas rationnel** (section sur le lemme de l'étoile).

Expression rationnelle canonique

Une expression rationnelle (rationnelle) est dite **canonique** si et seulement si les seuls opérateurs utilisés sont les opérateurs ×, ∪ et *.

 En conséquence, si un langage peut être décrit par une expression rationnelle canonique, c'est-à-dire utilisant uniquement les opérateur ×,∪ et *, alors ce **langage est** nécessairement **rationnel**. Cependant, l'expression canonique, si elle existe nécessairement, peut s'avérer extrêmement complexe. Une autre façon de prouver qu'un langage est rationnel est, nous le verrons plus loin dans le cours, de construire un automate d'état fini reconnaissant ce langage. D'autres solutions consistent à décomposer le langage comme union, étoile ou concaténation de deux langages rationnels, à présenter une grammaire rationnelle ou d'essayer de décrire le complémentaire du langage. En effet, nous verrons plus loin que le complémentaire d'un langage rationnel est un langage rationnel (théorème). Enfin, nous verrons aussi que l'intersection de deux langages rationnels est un langage rationnel (théorème).

Les expressions rationnelles décrivent des langages, c'est-à-dire des ensembles de mots sur un alphabet A donné (souvent implicitement). De fait, il s'avère que deux expressions différentes peuvent décrire un même langage. Par exemple, a*a et aa* désignent toutes deux l'ensemble des chaînes de n a consécutifs pour n>0. On peut par ce biais définir une équivalence entre expressions rationnelles.

Equivalence d'expressions rationnelles

Deux expressions rationnelles w_1 et w_2 sont dites équivalentes (ou par abus de langage "égales"), noté "$w_1 \equiv w_2$" (ou "$w_1 = w_2$"), si elles décrivent le même langage rationnel.

Soient, u, v et w des expressions rationnelles, on a entre autres les propriétés suivantes :

- L'union
  - u|v = v|u (commutativité)
  - u|u = u (idempotence)
  - u|∅ = u (∅ : élément neutre)
  - u|(v|w) = (u|v)|w (associativité)
- La mise à l'étoile
  - $u^{**} = u^*$ (idempotence)
  - $\emptyset^* = \varepsilon$
- La mise à l'étoile et l'union
  - $u^* = u|u^*$ (absorption)
- La concaténation
  - $u\varepsilon = \varepsilon u = u$ (ε : élément neutre à gauche et à droite)
  - $u \emptyset = \emptyset u = \emptyset$ (∅ : élément absorbant à gauche et à droite)
  - u(vw) = (uv)w (associativité)
- La concaténation et l'union
  - (u|v)w = uw|vw (distributivité à droite de la concaténation par rapport à l'union)
  - u(v|w) = uv|uw (distributivité à gauche de la concaténation par rapport à l'union)

$uw \neq wu$ (la concaténation n'est pas commutative) !... sauf si u = w

Facteur itérant

Un **facteur itérant** est une sous-chaîne non vide pouvant être "étoilée". Autrement dit, v est facteur itérant de $m \in L$ si : $m=uvw$, $|v| \neq 0$, $\forall i$, $uv^iw \in L$ ($uv^*w \subset L$).

Par exemple, dans le mot "bbaab" du langage décrit par $bb(a^2)^*b$, le facteur "aa" est un facteur itérant.

Mot primitif

Un **mot primitif** est un mot u tel que $u=v^n$ si n=1, c'est-à-dire s'il n'est pas puissance d'un autre mot que lui-même.

Par exemple, "abab" n'est pas primitif alors que "ab" l'est sur l'alphabet {a,b}.

Exercices et tests :

Exercice 3.1. Ecrire les expressions rationnelles décrivant les langages suivants :

1. {ab}
2. $\{a^n b a^m, n \geq 0, m \geq 0\}$
3. $\{a^n, n \geq 2\}$

Exercice 3.2. Décrire les langages définis par les expressions rationnelles suivantes (on donne d'abord l'alphabet puis l'expression) :

1. A_1={0,1,2,3,4,5,6,7,8,9,H}, ([1-9][0-9]*(0|2|4|6|8)) | (0|2|4|6|8)
2. A_1, ([0-9]|10|11|12)H[0-5][0-9]
3. A_2={a,b, ..., z}, regarder(ai|as|a|ons|ez|ont)
4. A_3={a,b,c}, ((a|b|c)(a|b|c)(a|b|c))*
5. A_3, (a|b|c)*a(a|b|c)*a(a|b|c)
6. A_3, (b|c)*a(b|c)*a(b|c)*

Exercice 3.3. Donner l'alphabet (si nécessaire) et l'expression rationnelle permettant de décrire :

1. le langage dont les mots sont les nombres décimaux ; 
2. Les mots sur {a,b} qui contiennent au moins un b et ne peuvent avoir deux b consécutifs (c'est à dire que bb ne peut être un facteur des mots du langage) ;
3. l'ensemble des mots sur {a,b,c} tels que toute paire de a est immédiatement suivie d'une paire de b. 

Exercice 3.4. Pour chacune des expressions rationnelles suivantes, donner une expression rationnelle plus simple décrivant le même langage :

1. $a^+a^*bc \mid aa?b?c$ 
2. $([a\text{-}z]^*)^*(c(ab)^* \mid ca(b(ab)^*))$
3. $a(ba)^*b \mid (ab)^+cd$ 
4. $a(ba)^*b(bc|c) \mid ab^*b?c$ 

Exercice 3.5. Expression des chemins sous Unix

On veut représenter des noms complets de fichiers UNIX constitués d'un chemin absolu (c'est à dire partant de la racine) suivi du nom connu localement dans le répertoire. On considère que la longueur des chaînes à représenter n'est pas limitée et on impose les contraintes supplémentaires suivantes : toute sous-chaîne représentant un nom local de répertoire ou de fichier (c'est à dire comprise entre deux '/') ne doit ni être vide, ni se terminer par un '.', ni contenir deux points juxtaposés.

On se donne l'alphabet suivant : {A-Z,a-z,0-9,'.', '/','_'}

Ecrire une expression rationnelle (la plus simple possible) sur cet alphabet qui représente un nom complet de fichier UNIX tel qu'il est spécifié ci-dessus.

- Exemples de chaînes à reconnaître :
  - /home/Oscar/Annee.99/Algonum/exos.tar.gz
  - /.forward
  - /index.htm
  - /home/Oscar/25_mai
  - /A_B_C/Oscar/25_mai
- Exemples de chaînes à ne pas reconnaître :
  - /home/Oscar/Annee.99/Algonum/ (ne peut pas terminer par "/" car ce n'est pas un fichier)
  - /. (ne peut pas terminer pas un ".")
  - / ("/" seul n'est pas un fichier)
  - exos.tar.gz (ce n'est pas un chemin absolu !)

- / forward. (termine pas un ".")

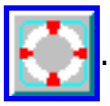
Exercice 3.6. Prouver que le langage suivant est rationnel : les mots sur {a,b} ne contenant pas la chaîne "aba". 

Exercice 3.7. Expression rationnelles positives

Les expressions rationnelles positives sont définies, comme les expressions rationnelles, de manière inductive :

- $\emptyset$ et a, $\forall$ a $\in$ A, sont des expressions rationnelles positives ;
- si e et f sont des expressions rationnelles positives alors e|f, ef et e^+ le sont aussi.

Le langage dénoté par une expression rationnelle positive est défini comme le langage dénoté par une expression rationnelle. On note PRat(A*) la famille des langages dénotés par une expression rationnelle positive.

1. Vérifier que ε n'appartient à aucun langage de PRat(A*). 
2. Montrer que PRat(A*) $\subset$ Rat(A*)  .
3. Montrer que si L $\in$ Rat(A*) alors L\ε $\in$ PRat(A*). 
4. Donner le principe de l'algorithme qui transforme une expression rationnelle en une expression rationnelle positive équivalente (à ε près).

4. Définitions rationnelles

Nous présentons maintenant une méthode pratique pour déclarer des langages rationnels souvent de façon plus simple et donc plus lisible que sous forme d'expressions compliquées.

Pour des raisons de commodité de notation, il est pratique donner des noms aux expressions rationnelles et de définir ensuite de nouvelles expressions rationnelles en utilisant ces noms comme s'ils étaient des symboles. La seule précaution à prendre est d'éviter les confusions en choisissant judicieusement les noms que l'on associe aux expressions rationnelles. Dans la définition qui suit, qui introduit la notion de définition rationnelle, nous utilisons pour cela les caractères gras.

Définition rationnelle

Si A est un alphabet de symboles de base, une **définition rationnelle** est une suite de n définitions ($n \in N$) de la forme $\mathbf{d_i} \rightarrow r_i$ (pour $i \in \{1,..., n\}$), où chaque $\mathbf{d_i}$ est un nom distinct, n'appartenant pas à $(A \cup \{|, *, \varepsilon,(,)\})^*$, et chaque r_i une expression rationnelle sur l'alphabet augmenté $A \cup \{d_1, ..., d_{i-1}\}$.

Exemple : les identificateurs en Pascal sont définis par :

- **lettre** → A | B | ... | Z | a | b | ... | z ;
- **chiffre** → 0 | 1 | ... | 9 ;
- **id** → **lettre** (**lettre**|**chiffre**)*.

NB : Lex est un outil permettant de générer des programmes de reconnaissance de mots (analyseurs lexicaux) à partir d'une suite de définitions rationnelles (Lex est présenté plus loin).

On remarque que chaque membre gauche (r_i) d'une règle ($\mathbf{d_i} \rightarrow r_i$) ne peut faire intervenir que les nouveaux symboles définis **antérieurement** (les $\mathbf{d_j}$ tels que $j<i$) et donc on n'a pas, dans le cas général, de définition récursive, comme le serait par exemple la définition caractérisée par l'unique règle **X** → a**X**b|ab, qui engendre pour **X** le langage $\{a^n b^n \mid n \in N\}$. En effet, si l'on autorisait l'écriture de définitions récursives, on pourrait caractériser des langages qui ne sont pas rationnels, comme par exemple $\{a^n b^n \mid n \in N\}$, ci-dessus. Ces définitions récursives seront d'ailleurs utilisées pour construire d'autres classes de langages, à l'aide de "grammaires", comme nous le verrons dans un prochain chapitre.

Par contre, il est possible d'autoriser une forme de récursivité : la récursivité droite, c'est-à-dire des définitions de la forme **X** → x**X**|y avec x et y des expressions rationnelles. Par exemple, la définition suivant est autorisée : **X** → a**X**|a. Elle engendre pour **X** le langage $\{a^n \mid n \in N\}$. Donc **X** → a**X**|a est équivalente à **X** → a^+.

Exercices et tests :

Exercice 4.1. Donner l'alphabet et les définitions rationnelles permettant de décrire le langage dont les mots sont les nombres décimaux.

Exercice 4.2. Expression des cellules sous Excel

Sous Excel, une cellule est repérée par son numéro de ligne et son numéro de colonne (exprimé en lettres). Par exemple, « L2 » définit la cellule en ligne « 2 » et colonne « L ». De même, « AB23 » définit la cellule en ligne « 23 » et colonne « AB ».

Une zone est indiquée par la cellule en haut à gauche et la cellule en bas à droite, toutes les deux séparées par deux points (« : »). Par exemple, « L2:M6 » indique la zone allant de la cellule « L2 » à la cellule « M6 ». L'ordre des cellules n'importe pas. Par exemple, « M6:L2 » est identique à « L2:M6 ».

Une colonne entière est définie par son nom seul. Par exemple, « L:L » indique la zone contenant toute la colonne « L ». De même, une ligne est définie par son numéro seul.

Une zone peut être aussi une seule cellule ou une suite de zones séparées par « ; ». Par exemple, « M6 ; A3:C18 ; G:I ; 4:18 » indique la zone composée de la cellule « M6 », de la zone « A3:C18 », des colonnes « G » à « I » et des lignes 4 à 18.

Écrire l'expression rationnelle décrivant une zone sous Excel. Vous utiliserez la notation des définitions rationnelles pour simplifier l'expression.

Exercice 4.3. Expression de polynômes

On cherche à écrire le langage des polynômes de la variable x à coefficients entiers. Les polynômes ne sont pas nécessairement réduits (il peut y avoir plusieurs monômes de même degré), ni ordonnés. Le polynôme nul existe et se note 0. Les monômes, quant à eux, doivent être réduits selon les conventions habituelles. Quand ils ne peuvent pas être réduits, ils sont composés d'un coefficient à un chiffre suivi de x et de l'exposant à un chiffre. Le produit entre le coefficient et la variable est une simple concaténation, l'opérateur d'élévation à la puissance est "^".

- Exemples de polynômes valides :
  - -3x^5-7+x^3+2x^5-8x
  - 6x^3
  - x-1
  - x
  - 0
- Exemples de polynômes non valides :
  - -3x^5 - 7 + 1x^3 => -3x^5 - 7 + x^3
  - 2x^5 - 8x^1 => 2x^5 - 8x
  - 33x^10 => 3x^9
  - -7x^0 => 1
  - 3x^2 + 0 => 3x^2

En utilisant les définitions rationnelles, donner une expression rationnelle, la plus simple possible, qui dénote le langage des polynômes tel qu'il est décrit ci-dessus. Précisez l'alphabet.

5. Systèmes d'équations rationnelles

Nous définissons maintenant des "équations" entre expressions rationnelles et montrons comment résoudre des systèmes d'équations de ce type.

Les définitions rationnelles permettent de proposer des expressions rationnelles plus lisibles. Cependant, il est parfois nécessaire d'obtenir une expression rationnelle ne comportant que des symboles du vocabulaire. Il faut donc alors essayer de trouver (ou retrouver) cette expression. La simple substitution n'est pas toujours suffisante, surtout en cas de récursivité. La solution consiste à passer par la résolution d'un système d'équations rationnelles. La transformation s'effectue en utilisant directement les définitions rationnelles. L'opérateur → devient =, l'opérateur | devient +, parfois ε devient 1 et les identificateurs deviennent des variables. La résolution du système ainsi obtenu donne l'expression rationnelle du langage. Nous allons donc décrire ces systèmes d'équations ainsi que la méthode de résolution.

Système d'expressions rationnelles

Les **systèmes d'équations rationnelles** sont des ensembles d'équations dont les coefficients sont des expressions rationnelles.

Par exemple, le système suivant est un système d'équations rationnelles :

$X=a_1X+a_2Y+a_3$ $Y=b_1X+b_2Y+b_3$	avec a_i, b_i (i=1,2,3) des expressions rationnelles.

Alors une solution est :

$X=(a_1|a_2b_2*b_1)*(a_3|a_2b_2*b_3)$

$Y=(b_2|b_1a_1*a_2)(b_3|b_1a_1*a_3)$

Remarque : les solutions ne sont pas uniques, mais, en général, on s'intéresse à la plus petite des solutions (le plus petit point fixe).

Système d'expressions rationnelles en forme standard

Un ensemble d'équations rationnelles d'indéterminés $D=\{X_1, ..., X_n\}$ est en forme standard si $\forall X_i \in D$, il y a une équation de la forme $X_i=a_{i0} + a_{i1}X_1 + ... + a_{in}X_n$ avec a_{ij} des expressions rationnelles sur A telles que $A \cap D = \emptyset$.

Remarques : il est possible d'avoir $a_{ij}=\emptyset$ quand il n'y a pas de terme correspondant à X_j dans l'équation de X_i. $\emptyset$ joue le rôle de 0 et ε celui de 1 (l'équation pour X_i comporte alors X_j sans coefficient à droite). Souvent, on utilise même 1 plutôt que ε dans ces équations.

Pour résoudre un tel système, il suffit, par analogie avec les systèmes linéaires, de procéder par élimination de variables. Pour cela, il est nécessaire de passer par le lemme d'Arden.

Lemme d'Arden

Soient K et L deux langages sur A* ($K \subseteq A^*$ et $L \subseteq A^*$), $\varepsilon \notin K$, alors :

1. K*L est l'unique solution de l'équation X = KX + L.
2. LK* est l'unique solution de l'équation X = XK + L.

Si $\varepsilon \in K$, alors A* est solution et K*L la plus petite solution.

Démonstration :

Les deux cas se traitent de manière similaire. Pour démontrer Arden (par exemple le cas 1) il suffit d'abord de montrer que X=K*L est une solution de X=KX+L. Ensuite, il faut montrer que cette solution est la seule possible. Pour cela, il faut procéder par l'absurde en proposant une autre solution P à l'équation.

Démonstration du cas 1 (le cas 2 est identique) :

1. K*L est une solution pour X=KX+L ?
   K* = {ε}+KK* donc K*L = {ε}L+KK*L = L+KK*L
   Avec X = K*L, on obtient alors X = KX+L : CQFD
2. K*L est l'unique solution ?
   Soit P une autre solution pour X = KX+L.
   Alors P=KP+L (P solution donc X=P)
   Donc $L \subset P$ et donc $KL \subset KP \subset P$...
   Par récurrence, $K^*L \subset P$
   Soit x le mot le plus court de P\K*L (ie, x est dans P mais pas dans K*L).
   $x \notin K^*L$ donc $x \notin L$ (à cause de l'opérateur *) mais $x \in KP$ car $x \in P = KP + L$ et $x \notin L$
   $x \in KP$ donc x = zy avec $z \in K$ et $y \in P$
   Comme $\varepsilon \notin K$ alors $|z|>0$
   Donc comme x=zy, $|x| = |z|+|y|$ et donc $|x| > |y|$ ($y \in P$)
   Donc $y \in K^*L$ (x est le plus court de P sauf K*L, comme y est dans P et est encore plus court que x, il est forcément dans K*L)
   Or x = zy avec $z \in K$ et $y \in K^*L$, donc $x \in KK^*L$
   Or $KK^*L \subseteq K^*L$, donc $x \in K^*L$ or $x \notin K^*L$
   Donc, l'hypothèse d'un x en dehors de K*L est fausse, donc K*L est l'unique solution : CQFD

De ce lemme, il est possible de démontrer un certain nombre d'**identités rationnelles** très

utiles pour démontrer des équivalences d'expressions rationnelles ou pour simplifier des expressions.

En particulier, il assez aisé de démontrer les identités suivantes :

- (E+F)* = E*(FE*)* (e1)
- (E+F)* = (E*F)*E* (e1')
- (EF)* = 1+E(FE)*F (e2)

Pour démontrer e1 par exemple, il suffit de démontrer que (a+b)* = a*(ba*)*. (a+b)*, selon le lemme d'Arden, est l'unique solution de l'équation X = (a+b)X+1. Si (a+b)* = a*(ba*)*, alors a*(ba*)* est aussi solution de cette équation :

(a+b)(**a*(ba*)***)+1 =	$a^+(ba^*)^* + ba^*(ba^*)^*+1$
	$a^+(ba^*)^* + (ba^*)^+1$
	$a^+(ba^*)^* + (ba^*)^*$
	$(a^+ + 1)(ba^*)^*$
(a+b)(**a*(ba*)***)+1 =	**a*(ba*)***

Donc est bien solution de l'équation : CQFD

Revenons maintenant à l'algorithme de résolution d'un système d'équations rationnelles. Soit un système de n équations rationnelles notées X_1, ..., X_n. Cette méthode se déroule en trois phases :

1. Ecrire, quand c'est possible, les équations du système sous la forme $X_i = x_iX_i + Y_i$ où x_i est une expression rationnelle sur A et Y_i est une expression rationnelle de la forme $y_0 + y_1X_1 + ... + y_{i-1}X_{i-1} + y_{i+1}X_{i+1} + ... + y_nX_n$ avec y_i des expressions rationnelles sur A.
2. Pour toutes les équations X_i de X_1 à X_{n-1}, sachant que selon le lemme d'Arden, l'unique solution pour $X_i = x_iX_i+Y_i$ est $x_i^*Y_i$, remplacer dans toutes les équations X_{i+1}... X_n la variable X_i par $x_i^*Y_i$
3. Pour toutes les équations X_i de X_n à X_1, calculer la valeur de X_i en appliquant le Lemme d'Arden et remplacer X_i par cette valeur dans toutes les équations de X_{i-1} à X_1.

 Il faut penser à simplifier continuellement vos expressions rationnelles pour avoir un résultat le plus simple possible. De plus, en fonction des simplifications choisies et de l'ordre adopté, les résultats ne sont pas toujours identiques (du point de vue des expressions obtenues) mais toujours équivalents.

Par exemple, Soit $D=\{X_1, X_2, X_3\}$ avec $A=\{0,1\}$ et le système S :

$X_1 = 0X_2 + 1X_1 + \varepsilon$

$X_2 = 0X_3 + 1X_2$

$X_3 = 0X_1 + 1X_3$

étape 1	$X_1 = 1X_1 + (0X_2 + \varepsilon)$ $X_2 = 1X_2 + (0X_3)$ $X_3 = 1X_3 + (0X_1)$
étape 2	$X_1 = 1X_1 + (0X_2 + \varepsilon)$ $X_2 = 1X_2 + (0X_3)$ $X_3 = 1X_3 + (01^*(01^*0X_3 + \varepsilon))$
étape 3	$X_3= (1+01^+0)X_3 + 01^*=(1+01^*01^*0)^*01^*$ $X_2 = 1^*(0X_3) = 1^*0(1+01^*01^*0)^*01^*$ $X_1 = 1^*(0X_2 + \varepsilon) = 1^*01^*0(1+01^*01^*0)^*01^*+1^*$

Nous verrons dans un prochain chapitre que ces systèmes sont utiles pour obtenir la description par une expression de langages décrits par un automate d'état fini.

Exercices et tests :

Exercice 5.1. a et b sont deux expressions rationnelles. Montrer que :

1. $a(a+ba)^* = (a+ab)^*a$ 
2. $(a+b)^+ = a^+ + a^* (ba^*)^+$ 

Exercice 5.2. Démonter que $a(ba)^*b = (ab)^+$. 

Exercice 5.3. Résoudre les systèmes suivants :

1. $L_1 = aL_2 + bL_3$
   $L_2 = bL_2 + 1$
   $L_3 = bL_3 + aL_2$
2. $L_1 = bL_2 + bL_3$
   $L_2 = aL_2 + aL_4 + 1$
   $L_3 = cL_4$
   $L_4 = cL_4 + 1$
3. $L_1 = (a+b)\, L_2$
   $L_2 = (a+b)\, L_2 + cL_3$
   $L_3 = cL_4$
   $L_4 = cL_3 + (a+b)L_5 + 1$
   $L_5 = (a+b)L_5 + 1$

6. Lemme de l'étoile

Les langages rationnels sont exactement les langages reconnaissables par les automates finis (qui sont présentés plus loin) et donc, pour montrer qu'un langage est rationnel, la méthode la plus courante consiste à produire soit une expression rationnelle canonique qui lui correspond, soit un automate fini qui le reconnaît.

De manière générale, par contre, il n'est pas toujours facile de montrer qu'un langage n'est pas rationnel. Pour cela, on utilise souvent le théorème suivant :

Lemme de l'étoile
Lemme de la pompe
Lemme d'itération
Lemme d'Ogden pour les rationnels

Soit L un langage rationnel sur un alphabet A. Alors il existe un entier naturel n tel que pour tout mot z de L vérifiant $|z|>n$, il existe $u,v,w \in A^*$ tels que $z=uvw$, $v \neq \varepsilon$, $|uv| \leq n$ et pour tout $i \in N$, $uv^iw \in L$.

Intuitivement : pour tout langage rationnel L, il existe une taille de mot n à partir de laquelle tout mot de L plus long possède au moins un facteur itérant dans ses n premières lettres.

Remarque : il existe bien d'autres formulations de ce lemme avec des propriétés parfois un peu différentes. Un début de discussion à ce propos se trouve dans [Séébold 99]. Il présente un lemme de l'étoile "fort" et un lemme de l'étoile "faible". Celui présenté dans ce cours est le second.

En fait, ce lemme peut s'appliquer à des langages qui ne sont pas rationnels mais tout langage rationnel vérifie ce lemme. Autrement dit, $Rat(A^*) \subset Etoile(A^*) \subset P(A^*)$


Ce théorème permet souvent de **démontrer par l'absurde qu'un langage L n'est pas rationnel** : il suffit de montrer que, **pour tout n de N**, **il existe un mot** z de L qui ne vérifie pas la condition **pour un i donné**. Mais attention, un mot peut ne pas fonctionner parce que trop court ! Généralement, il n'est pas nécessaire de trouver le n. C'est vrai pour tout n'>n, il n'est donc pas nécessaire de trouver le n optimal. En effet, si le facteur itérant est dans le n premières lettres alors il est aussi dans les n' premières.


Par contre, **montrer qu'un langage vérifie le lemme de l'étoile ne prouve pas qu'il est rationnel**. Si le lemme de l'étoile ne permet pas de prouver qu'un langage est rationnel, une piste possible est de faire une démonstration par l'absurde en cherchant à démontrer qu'il est rationnel.

Un exemple, si l'on reprend l'exemple de $\{a^mb^m \mid m \in N\}$ (un grand classique !), la démonstration prend la forme suivante :

Hypothèses :	Il existe un entier positif $n \in N$, tel que $\forall z \in L$, $\|z\| > n$, (H1) $\exists u,v,w$, tels que $z=uvw$, (H2) $v \neq \varepsilon$, (H3) $\|uv\| \leq n$, (H4) $\forall i \in N$, $uv^iw \in \{a^mb^m \mid m \in N\}$
Le mot qui va poser problème :	Considérons le mot $z=a^nb^n$ ($\|z\|=2n$, donc $\|z\|>n$). Soient u,v et w tels $z=uvw$ (H1) et vérifiant H2, H3 et H4 (z est plus long que n et par hypothèse, u,v et w existent).
La contradiction :	Tout préfixe de z de longueur $l \leq n$ est nécessairement composé uniquement de a. Comme uv est un préfixe de z (H1 : $z=uvw$) et $\|uv\|<n$ (H3), ceci est vrai en particulier de uv. Donc il existe $k,p \in N$ ($p \neq 0$) tels que $u=a^k$, $v=a^p$ (H2 : $v \neq \varepsilon$) et $k+p \leq n$. Et on a $z=uvw=a^ka^pw$, avec $w=a^{n-(k+p)}b^n$ (car $z=a^nb^n$). H4 s'écrit alors $\forall i \in N$, $a^k(a^p)^ia^{n-(k+p)}b^n \in \{a^mb^m \mid m \in N\}$, or ceci est faux : $a^k(a^p)^ia^{n-(k+p)}b^n = a^ka^p(a^p)^{i-1}a^{n-(k+p)}b^n = a^ka^pa^{n-(k+p)}(a^p)^{i-1}b^n = a^n(a^p)^{i-1}b^n$ et, comme $p \neq 0$, pour n'importe quel $i > 1$, $a^n(a^p)^{i-1}b^n \notin L$.
Conclusion :	Donc l'hypothèse est fausse. Donc, par le théorème de l'étoile, $\{a^mb^m \mid m \in N\}$ n'est pas un langage rationnel.

Remarquons à nouveau que ce lemme ne peut pas servir à démontrer qu'un langage est rationnel : il existe en effet des langages non rationnels qui le vérifient (par exemple $L = \{a^mb^pc^p \mid m,p \in N \text{ et } m > p\}$: il suffit de considérer n=2, car tout mot de L plus long que 2 commence nécessairement par 2 a et l'on peut itérer à volonté le deuxième a, par exemple, sans sortir du langage L).

Notons aussi que, assez souvent, lorsqu'il y a "relation" entre les éléments de l'expression, le langage n'est pas rationnel. Mais attention, ceci est un indice et non une règle universelle.

Exercices et tests :

Exercice 6.1. Montrer que les langages suivants ne sont pas rationnels  :

1. $\{a^n b^p : n < p\}$
2. $\{a^{nn}\}$
3. $\{a^n b^p : n \neq p\}$

Exercice 6.2. Les langages suivants sont-ils rationnels ? Le prouver.

1. $\{a^p : p \text{ premier}\}$
2. $\{a^n b^p : n \equiv p \pmod 2\}$
3. $\{a^n b^p : n \geq p\}$
4. $\{a^3 b^n a^3 : n \bmod 3 = 0\}$
5. L'ensemble des mots sur {a,b} de longueur impaire, dont la lettre médiane (la n ième si le mot est de longueur 2n-1) est un a ;
6. $\{a^m b^n : n + m \leq 1024\}$

Les grammaires de Chomsky

1. Introduction

Un langage est défini par l'ensemble de ses mots, ou, vu autrement, de ses phrases correctes (ce que nous voyions jusque là comme les symboles d'un mot peut alors être vu comme les mots d'une phrase). Il est impossible de donner la liste des phrases (ou mots) correctes d'un langage parce qu'elle est souvent infinie. Pour les langages rationnels que nous venons de voir, nous pouvons les dénoter par des expressions rationnelles. Toutefois, tous les langages ne sont pas rationnels et cette solution n'est donc pas toujours possible. Pour atteindre d'autres classes de langage, on peut utiliser divers types de grammaires, ce que nous présentons dans cette section.

Les grammaires peuvent être vues comme des "algorithmes" d'une part pour construire les mots du langage et d'autre part pour vérifier la validité d'un mot par rapport à un langage.

2. Structure d'une grammaire

Une grammaire à structure de phrase G permet de caractériser un langage L(G), qui est un ensemble de phrases (ou mots), appelé le langage engendré par la grammaire. Une telle grammaire ressemble à une définition rationnelle, en plus riche, plus expressif. Elle repose sur :

1. La connaissance d'un vocabulaire terminal (noté V_T) dont chaque élément est appelé symbole terminal. Ce vocabulaire correspond à l'alphabet pour les langages rationnels et les symboles terminaux aux symboles de l'alphabet. Nous avons donc $L(G) \subseteq V_T^*$. Dans ce cours, nous ne considérerons que les vocabulaires terminaux composés de symboles (lettres, chiffres, caractères spéciaux comme '+', '-', ';'...). En compilation, ce vocabulaire est un ensemble d'unités lexicales (identificateurs réservé ou non, paramètres, nom, opérateurs...).
2. La connaissance d'un vocabulaire non terminal (noté V_N) dont chaque élément est appelé symbole non terminal. Les symboles non terminaux peuvent être vus comme les nouveaux symboles des définitions rationnelles (utilisés comme membres gauches des règles). Ils ont toutefois un rôle un peu plus large. Ils n'appartiennent pas au vocabulaire de base (terminaux) et sont destinés à être éliminés, remplacés pour former les mots du langage.
3. Des règles permettant de déterminer quelles séquences de V_T^* sont légales. Ces

règles ressemblent à celles des définitions rationnelles mais sont plus souples et plus générales.

Grammaire

Une **grammaire formelle**, appelée aussi **grammaire de Chomsky**, G est une description de la forme des symboles et des phrases d'un langage noté L(G). Elle est définie par un quadruplet $G = (V_T, V_N, S, R)$ où :

- V_T est un ensemble fini non vide : le **vocabulaire terminal** ou **alphabet** ;
  $L(G) \subseteq V_T^*$
- V_N est un ensemble fini non vide : le **vocabulaire non terminal** (on parle aussi de **variable** ou de **catégorie syntaxique**) ; On a $V_T \cap V_N = \varnothing$ et on note $V=V_T \cup V_N$ le **vocabulaire** de la grammaire ;
- $S \in V_N$ est un symbole non terminal particulier appelé **l'axiome** ;
- R est un ensemble de **règles**, appelées **règles de production** (ou **règles de réécriture**), de la forme $\alpha \rightarrow \beta$ avec $\alpha \in V^+$ et $\beta \in V^*$.

Une règle de production $\alpha \rightarrow \beta$ se lit "α peut être remplacé par β". α est appelé "partie gauche" et β "partie droite" de la règle. On peut également écrire des règles de la forme $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$ avec $n \geq 1$ et pour tout $1 \leq i \leq n$, $\beta_i \in V^*$ pour abréger n règles ayant α en partie gauche, de la forme $\alpha \rightarrow \beta_i$.

Quelques exemples de grammaires :

- G_0=({0, 1}, {S, X}, S, R) avec R :
  - R1 : $S \rightarrow 0X1$
  - R2 : $0X \rightarrow 00X1$
  - R3 : $X \rightarrow \varepsilon$
- G_1=({0, 1}, {S, X}, S, R) avec :
  - R = {$S \rightarrow 0X1$; $0X \rightarrow 00X1$; $0X \rightarrow 001$}
  - ou R = {$S \rightarrow 0X1$; $0X \rightarrow 00X1 \mid 001$} (équivalent)
- G_2=({0, 1}, {S}, S, R) avec :
  - R = {$S \rightarrow 0S1 \mid 01$}

- G_3=({0, 1}, {S, X}, S, R) avec :
  - R = {S → 0S | OX ; X → 1X | 1}

Intuitivement, la grammaire G_0 définit le langage $L(G_0)=\{0^n1^n \mid n \geq 1\}$ sur le vocabulaire terminal (ou alphabet) A={0,1}. En effet, de façon informelle :

1. Appliquer une règle va consister à remplacer dans un mot une occurrence du membre gauche de la règle par le membre droit correspondant. Sur l'exemple, à partir de 0X1, on peut obtenir 00X11 en appliquant R2 (le lieu du remplacement étant signalé par les lettres en orange et la nouvelle chaîne est soulignée), ce que l'on peut noter :
   0X1 =R2⇒ 00X11.
2. On considérera l'ensemble des mots que l'on peut atteindre en prenant l'axiome S comme mot de départ et en appliquant les règles un nombre fini de fois. Sur l'exemple, on peut ainsi atteindre (entre autres) les mots "0X1" (par S =R1⇒ 0X1), "00X11" (par S =R1⇒ 0X1=R2⇒ 00X11) et "01" (par S=R1⇒0X1=R3⇒01); remarquons que l'on peut également atteindre S, par l'application successive de 0 règles. Les mots ainsi accessibles sont appelés "formes sententielles" et les suites d'applications de règles, "dérivations".
3. Les mots du langage caractérisé par la grammaire sont les mots accessibles (au sens du point précédent) qui ne sont composés que de symboles terminaux. Sur l'exemple, parmi les mots de $L(G_0)$, il y a "01" (S=R1⇒0X1=R3⇒01), mais aussi "0011" (S=R1⇒0X1=R2⇒ 00X11=R3⇒0011), et "000111" ...

On démontrera rigoureusement plus loin que $L(G_0)$ est exactement $\{a^nb^n \mid n \geq 1\}$. Auparavant, nous donnons les définitions rigoureuses correspondant aux idées que nous venons de présenter intuitivement.

Formes sententielles et langage d'une grammaire

Soit $G = (V_T, V_N, S, R)$ une grammaire et $V=V_T \cup V_N$, alors :

- l'ensemble des **formes sententielles** (ou **proto-phrases**) de G est défini récursivement par :
  - L'axiome "S" est une **forme sententielle** de G.
  - Pour tous v, x, y, z ∈ V*, si "xyz" est une forme sententielle de G et si y → v ∈ R, alors "xvz" est une forme sententielle de G.
- Le **langage engendré** par G est $L(G)=\{x \in V^* \mid x \text{ est une forme sententielle de G et } x \in V_T^*\}$.

Une forme sententielle est donc un mot sur V. Les formes sententielles ne contenant aucun symbole non terminal sont donc les phrases de L(G). Par ailleurs, la définition par induction des formes sententielles est la base de la notion d'application de règles que nous avons évoquée ci-dessus. Nous précisons ce point dans la section suivante.

3. Génération de chaînes, fonctionnement et dérivation

Une grammaire G est peut être utilisée soit pour générer les phrases du langage L(G) soit pour déterminer si une phrase donnée appartient ou non au langage L(G) :

- La génération consiste à produire des chaînes de symboles par application des règles de la grammaire à partir de son axiome S.
- La vérification (l'analyse) consiste à retrouver la succession d'applications de règles qui a permis d'obtenir un mot en partant de l'axiome S.

On appelle application de règle, une dérivation directe. On étend cette notion à des suites finies d'applications de règles par les définitions qui suivent.

Dérivations

Soit G = (V_T, V_N, S, R) une grammaire.

- On définit la relation "dérive directement de" sur V*, notée $=G1\Rightarrow$, par : $\forall$ z,z' $\in$ V*, z $=G1\Rightarrow$ z' $\Leftrightarrow \exists$ t, u, v, w $\in$ V*, z=uvw, et z'=utw, et v $\rightarrow$ t $\in$ R
  z $=G1\Rightarrow$ z' est appelé une **dérivation directe** de z' à partir de z.
  De plus, si u $\in$ V_T* (resp. w $\in$ V_T*) alors uvw $=G1\Rightarrow$ utw est une **dérivation à gauche** (resp. à droite).
  Intuitivement, on remplace dans uvw le non terminal v le plus à gauche (resp. à droite) pour obtenir utw.
- On appelle **dérivation de longueur n** (n $\in$ N) toute suite s=u_0, ...,u_n de chaînes de V* telle que u_0 $=G\Rightarrow$ u_1 et ... et u_{n-1} $=G\Rightarrow$ u_n (ce que l'on abrège par u_0 $=G\Rightarrow$ ... $=G\Rightarrow$ u_n)

Notations et terminologie : On écrit $\Rightarrow$ pour $=G\Rightarrow$ quand il n'y a pas d'ambigüité sur la grammaire et $=R\Rightarrow$ pour désigner explicitement la règle utilisée. On adopte en outre les conventions suivantes pour les dérivations, où a est la première chaîne de la dérivation et b la dernière :

- $a =n\Rightarrow b$ désigne une dérivation de longueur n.
- $a =^*\Rightarrow b$ désigne une dérivation de longueur quelconque. Autrement dit, $=^*\Rightarrow$ est la fermeture réflexive et transitive de $=G\Rightarrow$.
- $a =+\Rightarrow b$ désigne une dérivation de longueur supérieure à 1. Autrement dit, $=+\Rightarrow$ est la fermeture transitive de $=G\Rightarrow$.
- Si $a =^*\Rightarrow b$, on dit alors que a **engendre** b ou, symétriquement, que b **dérive de** a. Si de plus $a =+\Rightarrow b$, on dit "engendre" (resp. "dérive") "de façon non triviale".

Pour reprendre les grammaires précédentes, nous avons les dérivations suivantes :

- G_0 : $S \Rightarrow 0X1 \Rightarrow 0\underline{0X1}1 \Rightarrow 0\underline{00X1}11 \Rightarrow 000111$
- G_1 : $S \Rightarrow 0X1 \Rightarrow 0\underline{0X1}1 \Rightarrow 0\underline{001}11$
- G_2 : $S \Rightarrow 0S1 \Rightarrow 0\underline{0S1}1 \Rightarrow 0\underline{001}11$
- G_3 : $S \Rightarrow 0S \Rightarrow 0\underline{0S} \Rightarrow 0\underline{0X} \Rightarrow 0\underline{01X} \Rightarrow 0\underline{011X} \Rightarrow 000111$

En conséquence de ces définitions, les formes sententielles d'une grammaire sont exactement les chaînes que l'on peut dériver à partir de son axiome S. On peut donc proposer une nouvelle définition d'une langage engendré par une grammaire, équivalente à la précédente.

Soit $G = (V_T, V_N, S, R)$ une grammaire :

Langage (version 2)

- Une chaîne a est une forme sententielle pour la grammaire G si et seulement si $S =^*\Rightarrow a$.
- Le langage **engendré** par G est l'ensemble des chaînes de symboles terminaux que l'on peut dériver à partir de l'axiome S :
  $L(G)=\{x \in V_T^* \mid S =+\Rightarrow x\}$.

Cette définition n'est pas très différente de la précédente, mais elle permet de manipuler explicitement les dérivations, ce qui est pratique pour certaines démonstrations.

Si l'on revient sur l'exemple précédent, on peut maintenant montrer que le langage de la grammaire G_0 coïncide exactement avec $\{0^n1^n \mid n \geq 1\}$. Il faut pour cela montrer une double inclusion : $\{0^n1^n \mid n \geq 1\} \subseteq L(G_0)$ et $L(G_0) \subseteq \{0^n1^n \mid n \geq 1\}$.

Démonstration.

$G_0=(\{0, 1\}, \{S, X\}, S, R)$ avec R :

- R1 : $S \rightarrow 0X1$
- R2 : $0X \rightarrow 00X1$
- R3 : $X \rightarrow \varepsilon$

$\{0^n1^n \mid n \geq 1\} \subseteq L(G_0)$

Lemme : $\forall n \in N$, si $S =^* \Rightarrow a^nXb^n$ alors $S =^* \Rightarrow 0^n1^n$.

Démo : Ayant une dérivation $S =^* \Rightarrow 0^nX1^n$, on peut construire la dérivation $S =^* \Rightarrow 0^nX1^n =R3 \Rightarrow 0^n1^n$

Pour tout n, $0^n1^n \in V_T^*$. Il reste à montrer que $\forall n>1 \in N$, $S=^* \Rightarrow 0^n1^n$. Par le lemme, il suffit de montrer que $\forall n>1 \in N$, $S =^* \Rightarrow 0^nX1^n$, par récurrence sur n :

- n=1. On a la dérivation $S =R1 \Rightarrow 0X1$
- n=m+1, $m \geq 1$, et on a $S=^* \Rightarrow 0^{m-1}0X1^m$ (hypothèse de récurrence).
  On peut donc construire une dérivation de 0^n1^n à partir de S :
  $S =^* \Rightarrow 0^{m-1}0X1^m =R2 \Rightarrow 0^{m-1}\underline{00X1}1^m$.

$L(G_0) \subseteq \{0^n1^n \mid n \geq 1\}$

Lemme : $\forall n \in N$, $n \geq 1$, $\forall z \in \{0,1\}^*$, si $S =n \Rightarrow z$ alors $z=0^nX1^n$ ou $0^{n-1}1^{n-1}$

Démo : par induction sur la longueur des dérivations,

- n=1. Une seule dérivation possible $S =R1 \Rightarrow 0X1$ donc $z=0^1X1^1$.
- n=m+1, et on a si $S =m \Rightarrow w$ alors $w=0^mX1^m$ ou $w=0^{m-1}1^{m-1}$ (HR).
  Donc si $S =m \Rightarrow w =R \Rightarrow z$ alors au vu des règles de G_0, $w=0^mX1^m$ et
  - soit R=R2 , $w=0^{m-1}0X1^m$ et $z=0^{m-1}00X11^m=0^nX1^n$
  - soit R=R3 , $w=0^mX1^m$ et $z=0^m\varepsilon1^m=0^{n-1}1^{n-1}$

Donc $\forall z \in \{0,1\}^*$, si $S=+ \Rightarrow z$ et $z \in V_T^*$ alors $\exists n \in N$, $n \geq 1$, $z=0^n1^n$.
D'où par définition $L(G_0) \subseteq \{0^n1^n \mid n \geq 1\}$

On a donc établi et illustré la façon dont une grammaire définit un langage. Il peut arriver qu'un même langage L soit atteint par deux grammaires différentes.

Grammaire équivalente

Deux grammaires G_1 et G_2 sont dites équivalentes si $L(G_1) = L(G_2)$.

Par exemple, soit la grammaire $G_4=(\{0, 1\}, \{S, X\}, S, R)$ avec R :

- R1 : $S \rightarrow 0X1$;
- R2 : $X \rightarrow 0X1$;
- R3 : $X \rightarrow \varepsilon$.

On pourrait montrer que cette grammaire définit le même langage que G_0, c'est-à-dire $L(G_4)=\{a^n b^n \mid n \geq 1\}=L(G_0)$, par une démonstration du même type que la précédente. Autrement dit, G_0 et G_4 sont équivalentes (il en est de même avec G_1 et G_2).

Exercices et tests :

Exercice 3.1. Pour chacune des grammaires suivantes :

1. Générer quelques mots à l'aide de dérivations successives ;
2. Préciser leur longueur.

Gex1 = ({a,b,c}, {S}, S, {S → aSbSa | c})

Gex2 = ({a,b,ch,d}, {S,A,B,C}, S, {S → BCaCbbA ; A → CaCbb | ε ; Ca → ba ; Cbb → da ; B → cha})

Gex3 = ({a,b}, {S,A}, S, {S → Aa | bA ; A → Sa | bS})

4. Classification des grammaires

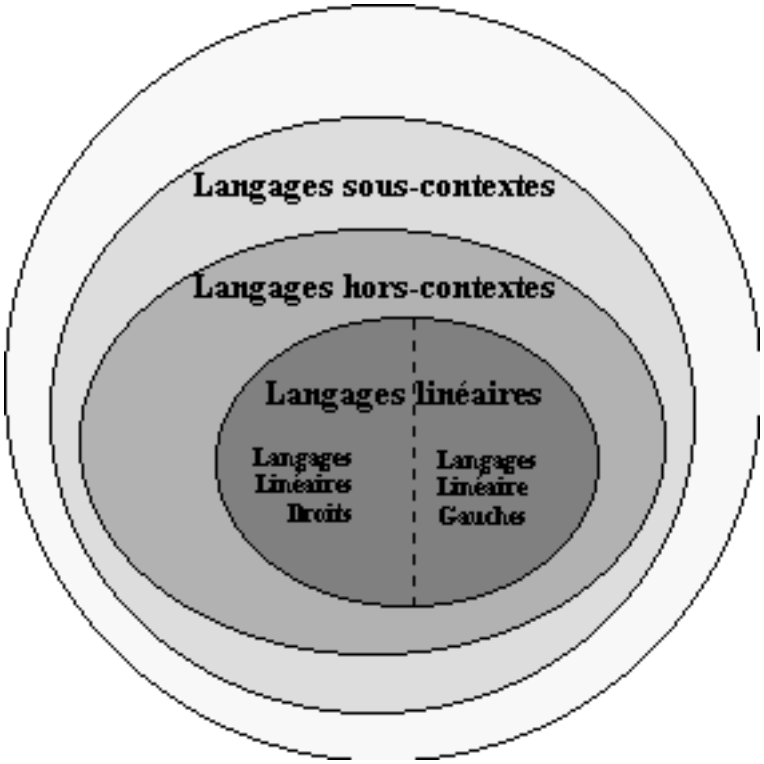
De même que les expressions rationnelles permettent d'atteindre la classe des langages rationnels (ou réguliers), différents types de grammaires permettent d'atteindre différentes classes de langages. La classification de N. Chomsky [Chomsky 56] se fonde sur des contraintes concernant la forme des règles de dérivation. Selon l'ordre croissant des contraintes, nous avons la classification suivante :

Type de la grammaire	Formes des règles
Type 0 : sans contraintes	- $a \rightarrow b$ avec a et $b \in V^*$.
Type 1 : contextuelle (context sensitive, sensible au contexte)	- $a \rightarrow b$ avec a et $b \in V^*$ et $\|a\| \leq \|b\|$.
Type 1 (bis) : contextuelle Autre définition	- $aAb \rightarrow aBb$ avec $a, b \in V^*$, $A \in V_N$ et $B \in V^+$ tel que $\|B\| \geq 1$ (c'est-à-dire que A peut être remplacé par B dans le contexte aAb). Habituellement, on introduit une exception à ces règles : l'axiome, s'il n'est pas en partie droite d'une autre règle que celle qui le définit, peut engendrer ε. Cette exception est aussi valable pour les types suivants
Grammaires **"à structure de phrase"**	- Grammaires de type 1 telles que les règles $a \rightarrow b$ avec $a \in V_N^+$ et $b \in V^+$.
Type 2 : hors-contexte (context free, algébrique)	- $A \rightarrow a$ avec $A \in V_N$ et $a \in V^+$.
Type 3 : linéaires (régulières, d'états finis) [à] droite (resp. [à] gauche)	- uniquement : $A \rightarrow x_1 \mid x_2B$ (resp. $A \rightarrow x_1 \mid Bx_2$) avec A, B $\in V_N$ et $x_1 \in V_T^+$, $x_2 \in V_T^*$.

Par exemple, une grammaire avec comme règles $S \rightarrow 0S \mid 1S \mid \varepsilon$ est linéaire droite. Dans la section précédente, G_0 et G_4 sont de type 0, G_1 est xde type 1, G_2 est de type 2 et G_3 est de type 3.

Les définitions rationnelles peuvent être considérées comme des grammaires hors-contexte particulières (si d_1 est l'axiome et si $\forall$ i, i>1, d_i n'engendre pas ε).

La classification de Chomsky introduit quatre classes de langage engendrées par les quatre types de grammaire. Ces classes de langage satisfont la relation d'inclusion stricte suivante :



$cl3 \subset cl2 \subset cl1 \subset cl0$

Cette relation exprime entre autres que tout langage linéaire est hors-contexte et qu'il existe des langages hors-contextes qui ne sont pas linéaires. On a le même type de relation entre les autres classes de langages. Les langages hors-contextes et linéaires sont les plus courants en informatique, en particulier dans le cadre des langages de programmation; les algorithmes de traitement pour ces deux classes sont les plus efficaces et les plus simples. Dans le cadre de ce cours, nous étudierons plus particulièrement le traitement des langages linéaires (en particulier à l'aide des automates finis). Cependant nous présentons avant quelques outils utiles sur les grammaires hors contexte (habituellement traité à l'aide des automates à pile).

Exercices et tests :

Exercice 4.1. Pour chacune des grammaires suivantes, donner leur type.

Gex1 = ({a,b,c}, {S}, S, {S → aSbSa | c})

Gex2 = ({a,b,ch,d}, {S,A,B,C}, S, {S → BCaCbbA ; A → CaCbb | ε ; Ca → ba ; Cbb → da ; B → cha})

Gex3 = ({a,b}, {S,A}, S, {S → Aa | bA ; A → Sa | bS})

Exercice 4.2. Pour chacune des grammaires suivantes, donner le langage engendré et préciser le type de la grammaire :

Gex4 = ({a,b}, {S,A,B}, S, {S → aAB ; B → SA ; Aa → Sab ; bB → a ; Ab → SBb})

Gex5 = ({/,\}, {S, U, V}, S, {S → UAV | UV ; A → VSU | VU ; U → / ; V → \})

5. Représentation des grammaires hors contexte

A propos des grammaires hors contextes, il existe diverses présentations standard pratiques aussi bien pour les règles des grammaires que pour les dérivations qu'elles permettent. Nous donnons dans cette section deux normes de présentation des règles, puis la représentation usuelle des dérivations sous forme d'arbre.

5.1 Représentations des règles de production

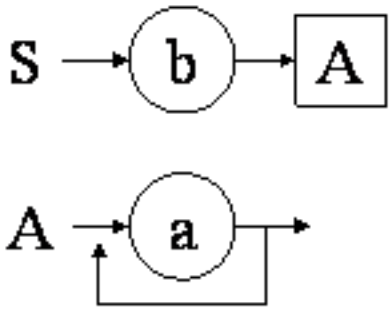
Il existe donc plusieurs manières de représenter les règles d'une grammaire.

Textuellement, la **norme BNF** ("Backus-Naur-Form") est la plus utilisée. Elle est particulièrement utile pour analyser un langage défini par une grammaire hors-contexte (et donc aussi pour une grammaire linéaire). Les symboles non terminaux sont encadrés par des guillemets ("..." ou <<...>>), les symboles terminaux ne sont pas distingués (parfois les non-terminaux sont en majuscule et les terminaux en minuscule). L'axiome est la partie gauche de la première règle énoncée. La disjonction est notée par une barre verticale (|). La flèche des règles est remplacée par le signe "::=". Par exemple, le tableau suivant illustre la forme BNF pour les règles {S → bA ; A → aA ; A → a}.

```
"S" ::= b"A"
```

```
"A" ::= a"A" | a
```

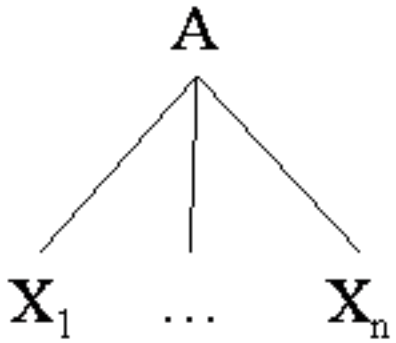
Il existe aussi une représentation graphique des règles en utilisant les **diagrammes de Conway**. Elle décrit graphiquement les grammaires hors-contexte. Il permet de regrouper des règles et de mettre en évidence les phénomènes récursifs. La partie gauche des règles est notée à gauche du schéma. Les symboles terminaux sont écrits dans des cercles et les symboles non terminaux sont écrits dans des rectangles. Une chaîne du langage est formée en parcourant les flèches et en recueillant les symboles rencontrés. La figure suivante montre les diagrammes de Conway pour la grammaire précédente.



Diagrammes de Conway pour les règles {S → bA ; A → aA ; A → a}

5.2 Arbres de dérivations

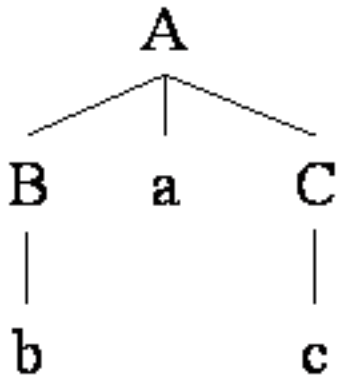
Rappelons que tout mot a du langage engendré par une grammaire G peut être obtenu par une dérivation $S \Rightarrow^{+} a$ avec a uniquement composé de vocabulaire terminal ($a \in L(G) \subseteq V_T^*$). Une telle dérivation peut être représentée par un **arbre de dérivation** (appelé aussi "**arbre syntaxique**" ou "**arbre d'analyse**"). L'idée consiste à associer à chaque règle $R : A \rightarrow X_1 \ldots X_n$ de la grammaire, où $A \in V_N$ et $\forall\ 1 \leq i \leq n$, $X_i \in V$, la structure d'arbre suivante :



La correspondance entre arbres et dérivations est alors la suivante :

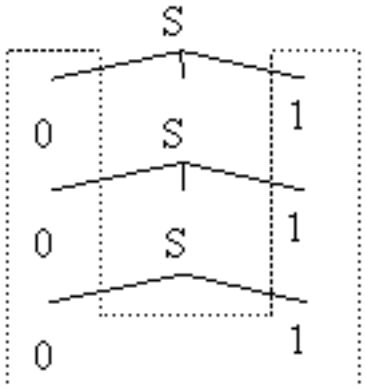
- Une dérivation directe $A =R\Rightarrow X_1 \dots X_n$ est représentée par l'arbre ci-dessus.
- Et une dérivation de la forme $A =n\Rightarrow Y_1 \dots Y_{i-1}\ Y_i\ Y_{i+1} \dots Y_m =R\Rightarrow Y_1 \dots Y_{i-1}X_1 \dots X_mY_{i+1} \dots Y_m$ est représentée par l'arbre associé à $A =n\Rightarrow Y_1 \dots Y_m$, dans lequel on remplace le symbole non terminal correspondant à Y_i par l'arbre associé à R.

Par exemple, en considérant les règles R1 : A → BaC, R2 : B → b et R3 : C → c, à la dérivation $A =R1\Rightarrow BaC =R2\Rightarrow baC =R3\Rightarrow bac$ correspond l'arbre suivant :



On peut remarquer que la représentation des dérivations sous forme d'arbres ne mémorise pas l'ordre dans lequel les règles sont appliquées, lorsque ces applications n'interfèrent pas. Par exemple, l'arbre ci-dessus correspond également à la dérivation $A =R1\Rightarrow BaC =R3\Rightarrow Bac =R2\Rightarrow bac$.

Si l'on prend la grammaire G_2 et le mot "000111", nous avons l'arbre de dérivation suivant :



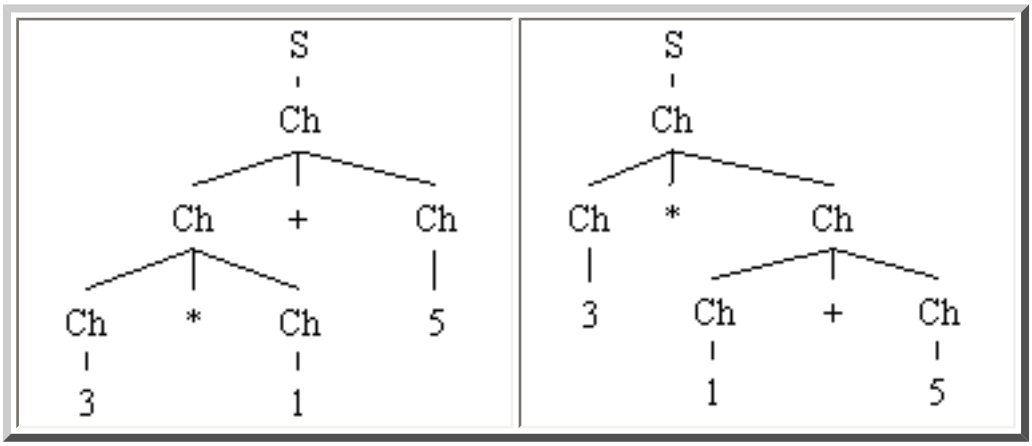
De façon générale, un arbre de dérivation d'un mot du langage engendré par une grammaire représente une façon de dériver ce mot à partir de l'axiome. La racine de l'arbre est l'axiome, ses feuilles sont les symboles (terminaux) du mots, et ses noeuds internes représentent les applications de règles. Notons que l'arbre de dérivation d'un mot n'est pas toujours unique. Une grammaire est dite ambiguë si au moins un mot dans

son langage a plus d'un arbre de dérivation.

Par exemple, soit la grammaire G_6= {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, *}, {S, A}, R}, avec R :

- S → Ch
- Ch → Ch + Ch | Ch * Ch
- Ch → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Par cette grammaire, nous pouvons obtenir pour le mot "3*1+5" les arbres de dérivation suivants :



Notons que ces deux arbres correspondent bien à deux façons différentes d'appliquer les règles qui cette fois, ne sont pas équivalentes. Intuitivement, l'arbre de gauche correspond à l'interprétation (3*1)+5 tandis que l'arbre droit correspond à 3*(1+5). L'ambiguïté dans les grammaires n'est donc pas un problème anodin, mais son étude approfondie sort du cadre de ce cours.

Exercices et tests :

Exercice 5.1. Pour chacune des grammaires suivantes :

1. Les écrire sous forme BNF
2. Donner les arbres de dérivation associés aux mots donnés à l'exercice 3.1. dans le cas où la grammaire est de type 2 ou 3. Les représenter à l'aide des diagrammes de Conway

Gex1 = ({a,b,c}, {S}, S, {S → aSbSa | c})

Gex2 = ({a,b,ch,d}, {S,A,B,C}, S, {S → BCaCbbA ; A → CaCbb | ε ; Ca → ba ;

Cbb → da ; B → cha})

Gex3 = ({a,b}, {S,A}, S, {S → Aa | bA ; A → Sa | bS})

Exercice 5.2. Soit la grammaire ({0,1,2,3,4,5,6,7,8,9,;,-,+},{S,Exp,Ch}, S, R) définie par les règles suivantes :

- S → Exp ; S | Exp
- Exp → Exp + Exp | Exp - Exp | Ch
- Ch → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

1. Ecrire ces règles sous forme BNF
2. Donner les diagrammes de Conway correspondants

Les automates d'état fini

---

1. Introduction

La notion d'automate a été introduite pour formaliser le concept d'algorithme et celui de calculabilité. Les automates sont définis sur la base de la machine de Turing [Turing 36]. Ce sont des objets mathématiques. Ils font donc abstraction des capacités des hommes ou des machines. Un résultat sera calculable s'il est atteint en un nombre fini de pas, ce nombre pouvant être aussi grand que l'on veut. Les notions de calculabilité, de décidabilité et de problème démontrable ne seront pas abordées dans ce cours. Nous allons nous intéresser à la structure des automates et à leur utilisation dans le cadre de la théorie des langages.

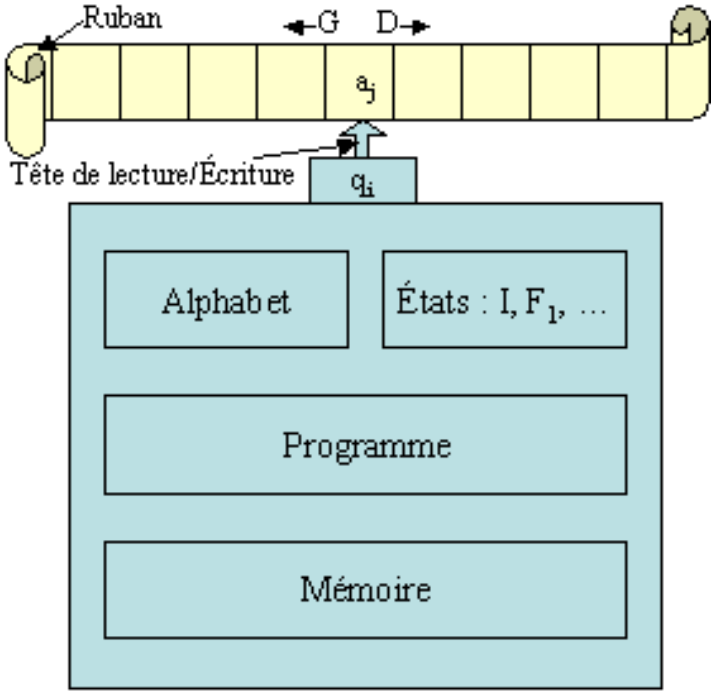
L'objectif de ce chapitre est d'introduire rapidement les automates à travers la présentation des machines de Turing. Puis, nous étudierons plus particulièrement un cas particulier de machine de Turing : les automates finis.

2. La machine de Turing

2.1. Présentation

En 1936, Alan Turing (1912-1954), dans [Turing36], présente sa machine (appelée depuis machine de Turing). Elle est composée :

- d'une **unité centrale** se trouvant dans un **état** q_i, pris parmi l'ensemble fini de ses états internes, noté $Q=\{q_1, q_2, ..., q_n\}$. Parmi les états, on distingue un **état initial** (état au lancement de la machine) et des **états finaux** (états de la machine en situation de terminaison). Le comportement de l'unité centrale est déterminé par son **programme**.
- d'un **ruban** de taille illimitée divisé en cases, contenant au départ les données à traiter, et à l'arrivée les résultats des calculs. Les données de départ tout comme les résultats sont représentés sur le ruban à l'aide de **symboles** pris dans un **alphabet** $A=\{a_0, a_1, a_2, ..., a_m\}$. Chaque case ne contient qu'un symbole.
- d'une **tête de lecture-écriture** qui assure la communication entre l'unité centrale et le ruban. Cette tête n'agit que sur une case à la fois. Elle peut **remplacer le symbole lu** par un autre symbole et **déplacer le ruban** d'une case vers la gauche ou vers la droite.



Structure d'une machine de Turing

Le rôle du programme consiste à déterminer, pour un état q_i de la machine et pour un symbole a_j situé devant la tête de lecture-écriture, le nouvel état dans lequel passe la machine ainsi que l'opération effectuée sur le ruban. On représente un programme par un ensemble fini de quintuplets, appelé **fonction de transition**. Celle-ci définit les règles d'évolution, appelées **règles de transition**, de la forme (q_i,a_j, q_k, a_l, d_m) ou $\bullet(q_i,a_j) = (q_k, a_l, d_m)$ où :

- $q_i \in Q$ est l'état initial ;
- $a_j \in A$ le symbole lu ;
- $q_k \in Q$ le nouvel état ;
- $a_l \in A \cup \{\varepsilon\}$ le caractère à écrire (ε n'écrit rien, c'est-à-dire laisse le caractère a_j sur le ruban) ;
- $d_m \in \{D, G, C\}$ le sens du déplacement de la tête avec respectivement D pour aller d'une case vers la droite, G d'une case vers la gauche et C pour ne pas se déplacer.

Parmi les états de la machine, il en existe un particulier : l'**état initial** (q_1). C'est le premier état pris par la machine. Il existe aussi un ensemble d'états appelés **états finaux**.

Configuration
Configuration initiale
Configuration finale (terminale)

On appelle **configuration** tout couple symbole / état. Une règle détermine donc pour une configuration courante, l'action à effectuer et donc la prochaine configuration.

On appelle **configuration initiale** tout couple symbole / état où l'état est un état initial.

On appelle **configuration finale** tout couple symbole / état où l'état est un état final.

Si la machine de Turing est telle qu'il n'existe pas de règle $\bullet(q_i,a_j)=(x,y,z)$ pour l'état q_i et le symbole a_j de la configuration courante alors le ruban est dit **terminal**. Si l'état courant est un état final alors l'exécution est réussie et le programme s'est déroulé normalement. Le cas contraire est une situation d'échec.

La suite de symboles S du ruban terminal est une chaîne dite terminale (la disposition de départ étant appelée chaîne initiale). Dans cette configuration S(q), la machine de Turing s'arrête. Par ailleurs, le symbole ε permet aussi souvent d'arrêter la machine, c'est-à-dire qu'il n'existe pas de transition de la forme $\bullet(q,\varepsilon)$.

Machine de Turing déterministe

La machine de Turing est dite **déterministe** si pour toute configuration, il n'existe qu'une seule fonction de transition applicable, c'est-à-dire si et seulement si pour $\bullet(x,y)=(z_1,t_1)$ et $\bullet(x,y)=(z_2,t_2)$ deux règles, alors $z_1=z_2$ et $t_1=t_2$. Dans le cas contraire, la machine est dite **non déterministe** (pour une configuration donnée, plusieurs règles sont applicables).

Machine de Turing

Une machine de Turing est définie par un quintruplet (A, Q, I, F, •) tel que :

- A est l'alphabet (fini, non vide) ;
- Q est l'ensemble des états possibles pour la machine (fini et non vide) ;
- $I \in Q$ l'état initial ;
- $F \subseteq Q$ l'ensemble des états finaux ;
- • la fonction de transition telle que $\bullet : Q \times A \rightarrow Q \times A \times \{D, G, C\}$. $\bullet(q_i,a_j)=(q_k,a_l,d_m)$ signifie qu'on effectue les actions de remplacement, de déplacement et de changement d'état en

q_k pour la configuration courante d'état et de symbole.

La suite $S_1(q_1)$, ..., $S_n(q_m)$ d'états (configurations) successifs du ruban et du programme lors de l'exécution d'une machine de Turing $T_.$, est appelée calcul. Si S_1 est la chaîne initiale et S_n une chaîne terminale alors S_n est aussi appelée **résultat** du calcul de S_1 par la machine $T_.$.

2.2. Cas particuliers de machine de Turing

Les différents types d'automates sont des cas particuliers de la machine de Turing. Parmi ceux-ci, on trouve les automates : linéairement bornés, à pile, finis...

Les automates finis sont des machines de Turing très spécifiques. Leurs caractéristiques, leur manipulation et leurs applications sont présentés en détail dans la suite du chapitre.

2.3. Machine de Turing et la théorie des langages

La machine de Turing manipule des symboles. Ceci n'est pas restrictif car tous les problèmes calculables (pour lesquels il existe un algorithme permettant de les résoudre dans un temps fini) peuvent se ramener à la manipulation de symboles et donc peuvent être programmés sur une machine de Turing.

Dans le cadre de la théorie des langages, la correspondance est évidente. Les machines de Turing sont alors appelées **accepteurs** ou **reconnaisseurs** [Aho et al. 91] [Ginzburg 68].

Soit A un alphabet et A* l'ensemble des mots que l'on peut construire avec les symboles de A. Une chaîne (mot) awb (a et b $\in$ A et w $\in$ A*), écrite sur le ruban et entourée de caractères ε, est acceptée si la machine part de la position gauche (i.e. dans la configuration initiale (I,a)) et s'arrête avec la tête de lecture placée après le caractère le plus à droite différent de ε dans un état final (i.e. dans la configuration finale (q_f,b) avec $q_f \in F$). Si elle s'arrête dans un état n'étant pas final alors la chaîne est refusée.

Le langage défini par un accepteur est l'ensemble des chaînes (mots) qu'il accepte. Il existe donc une correspondance entre les grammaires (et les langages) et les machines de Turing (cette correspondance sera étudiée en détail plus loin). Une machine de Turing (ou un automate) est donc alors vue comme l'implémentation d'une fonction f telle que :
$f : A^* \rightarrow \{Accepté, Refusé\}$.

3. Définition des automates d'état fini

3.1. Présentation

Nous allons étudier en détail un cas particulier de machine de Turing : les automates finis. Nous en donnerons d'abord une définition précise. Nous verrons ensuite les différents modes de représentation, leurs propriétés ainsi que les algorithmes permettant de les utiliser.

La théorie des automates finis commence avec l'article de S.C. Kleene [Kleene 56] paru en 1956. Dans cet article tant de fois cité, Kleene démontre le premier résultat de cette théorie. C'est à la fois le premier dans l'ordre chronologique mais aussi dans l'ordre d'importance. Nous verrons un peu plus loin ce théorème.

3.2. Définition

Un **automate fini**, très souvent appelé "**automate d'état fini**" [Perrin 95], ("**finite-state machine**", "**finite automaton**", "**Rabin-Scott Automaton**" [Ginzburg 68]) est une machine de Turing dont le ruban se déplace toujours vers la gauche et case par case. Ce ruban est infini à droite et fini à gauche. Par ailleurs, la tête de lecture de la machine se contente de lire des symboles et n'effectue aucune écriture.

Automate fini
Automate d'état fini
AFN

Un **automate fini** (**AFN**) est composé de la même manière que la machine de Turing. Généralement non déterministe, son comportement est cependant plus restreint. Il est défini par un quintuplet (A, Q, I, F, •) tel que :

- A est **l'alphabet** (vocabulaire), ensemble fini, non vide de symboles ;
- Q est **l'ensemble des états** possibles pour la machine (fini et non vide) ;
- $I \subseteq Q$ est l'ensemble des états initiaux ou états de départ ;
- $F \subseteq Q$ correspond aux états finaux ou états d'acceptation ;
- • la fonction de transition telle que • : $A \cup \{\varepsilon\} \times Q \rightarrow Q$. Pour la configuration courante d'état et de symbole, $\bullet(a_j, q_i)=q_k$ signifie qu'on passe dans l'état q_k et que le ruban est déplacé d'une case vers la gauche.

Remarques :

1. Certains auteurs, dont [Watson 93a], définissent les automates par un 6-uplet (A, Q, E, I, F, •) en ajoutant un ensemble E des ε-transitions. Dans notre définition, ces transitions sont définies par •.
2. Dans beaucoup de définitions, l'ensemble I est réduit à un seul état.
3. • est une fonction partielle, car elle n'est pas définie pour toutes les valeurs du domaine.

Le plus souvent, pour les couples (symbole, état) qui ne sont pas explicités, elle est définie de la façon suivante : •(a,q)=0 où 0 est un état qui n'existe pas. Ces transitions provoquent un arrêt de l'automate ainsi qu'une erreur. L'état 0 peut être considéré comme l'état d'erreur de l'automate.

4. Les transitions peuvent être aussi de la forme :
   $(a_i, q_j, q_k) \equiv (a_i, q_j) \rightarrow q_k \equiv \bullet(a_i, q_j) = q_k$
   avec $a_i \in A$; $q_j, q_k \in Q$;
5. Toujours à propos des transitions, $|(a_i, q_i, q_k)| < 2$

Au départ, une phrase construite sur le vocabulaire de l'automate est inscrite sur le ruban d'entrée et est entourée de symboles ε. L'unité centrale se trouve dans un des états initiaux de I et la tête de lecture est placée devant le symbole le plus à gauche de la phrase. A partir de cette position, l'automate va exécuter son programme. Pour une configuration courante (a_i, q_j), l'automate applique une transition $\bullet(a_i, q_j) \rightarrow q_k$, passe alors dans l'état q_k et déplace le ruban d'une case vers la gauche. L'automate tel qu'il est défini n'est pas forcément déterministe. Plusieurs transitions concurrentes sont donc parfois applicables. L'automate s'exécute jusqu'à ce qu'il arrive à une situation (a_t,q_n) pour laquelle il n'existe pas de transition permettant au programme de poursuivre son calcul. Il suspend alors son exécution. Si la configuration finale est telle que $a_t = \varepsilon$ et $q_n \in F$ alors la phrase inscrite sur le ruban est **acceptée** par l'automate. Dans tous les autres cas, elle est **refusée**.

Remarque : Une façon utile de regarder le non déterminisme est de considérer qu'il permet à un automate de "formuler des hypothèses". Si, dans un état donné, on ne sait pas quoi faire sur un certain caractère d'entrée, il existe plusieurs possibilités pour le choix de l'état suivant. Comme chaque chemin étiqueté par une chaîne de caractères et conduisant à un état d'acceptation est interprété comme acceptable, on fait confiance à l'automate non déterministe dès qu'il fait une hypothèse juste, quel que soit le nombre d'hypothèses fausses faites auparavant.

 Attention, une transition d'un AFN ne peut concerner qu'un seul symbole de l'alphabet et non pas un mot. Par contre, l'automate est indépendant des étiquettes des états. Ils peuvent être numérotés de manière continue ou pas, correspondre à des termes ayant un sens dans le langage courant...

3.2. Représentation des AFN

Les automates, et en particulier leur fonction de transition, peuvent être représentés de plusieurs manières : par des graphes orientés ou des représentations matricielles.

Prenons comme exemple l'automate T={{a,b,c}, {1}, {6,9}, •}. Soit • sa fonction de transition :

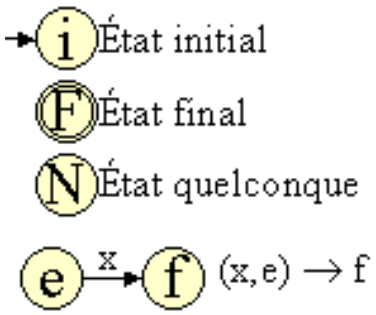
- μ(a,1) = 2 ; μ(b,1) = 4 ; μ(a,2) = 5 ; μ(b,2) = 3 ; μ(c,2) = 2 ; μ(a,3) = 6 ; μ(b,3) = 4 ; μ(c,3) =

9 ; μ(c,4) = 5 ; μ(a,5) = 5 ; μ(a,5) = 6 ; μ(c,7) = 4

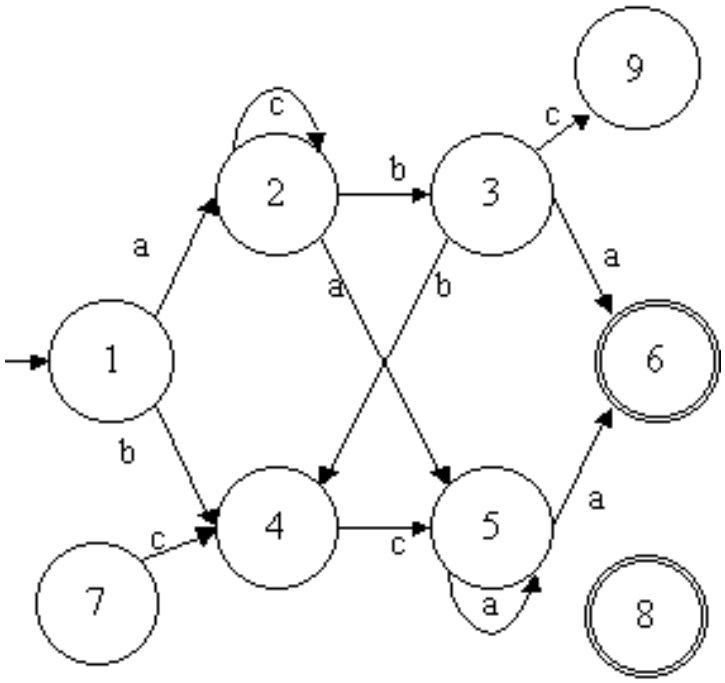
La fonction de transition est très souvent représentée par une "table de transition" (parfois aussi appelée "matrice de transition", "flow table" en anglais). Cette représentation est très commode du point de vue de l'implémentation informatique des AFNs. Les tables de transition sont des tableaux (ou des matrices) dont les colonnes correspondent aux symboles et les lignes aux états. La cellule (l_i,c_j) indique les transitions $(a_i, q_j) \rightarrow q_k$. L'AFN étant susceptible d'être non déterministe, plusieurs états peuvent être présents dans une cellule (l_i,c_j). Les symboles ne donnant aucune transition depuis un noeud dans un certain état sont alors à "0" (transition vers un état inexistant). Cet état correspond à un état d'erreur. C'est cette forme qui est la plus souvent utilisée en programmation. Si on reprend l'AFN servant d'exemple, nous avons alors la table suivante :

•	a	b	c
1	2	4	0
2	5	3	2
3	6	4	9
4	0	0	5
5	5,6	0	0
6	0	0	0
7	0	0	4
8	0	0	0
9	0	0	0

Un AFN peut être aussi représenté par un graphe orienté et valué, appelé "diagramme de transition" ou "graphe de transition". Les noeuds correspondent aux états et les arcs représentent les transitions possibles (calculées par la fonction de transition) entre les états. Ces arcs sont étiquetés par le symbole permettant cette transition. Les états initiaux sont représentés par un noeud possédant une flèche entrante sans origine. Un état final est représenté par un noeud doublement cerclé.

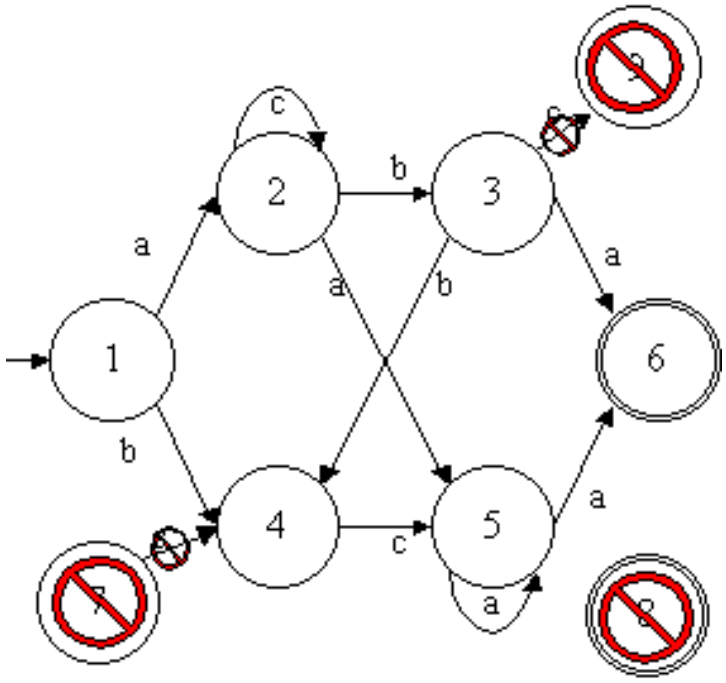


L'automate T, présenté précédemment, peut être représenté par le graphe suivant :



On observant cet automate T, de manière intuitive, on peut noter que certains états ne sont pas particulièrement utiles. Prenons d'abord l'état 7. Cet état n'intervient que dans une seule transition (de 7 vers 4). Aucune transition n'est "dirigée" vers 7. Cela signifie que la machine de Turing ne pourra jamais se trouver en l'état 7. Par conséquent, cet état peut être supprimé (ainsi que la transition de 7 vers 4). Pour l'état 8, c'est pire puisqu'il est totalement isolé. Il peut donc lui aussi être supprimé. Enfin, avec l'état 9, la situation est un peu différente mais le résultat sera le même. Lorsque la machine sera dans l'état 9, plus aucune transition n'étant exploitable, elle sera forcément en échec. Donc, prendre la transition de 3 vers 9 sur "c" sera forcément un échec. Si on supprime 9 et cette dernière transition, le résultat sera totalement identique, c'est-à-dire que étant sur 3, si le ruban est sur un "c", aucune transition ne sera exploitable et donc la machine se mettra aussi en échec. En conclusion, les états 7, 8 et 9 ainsi que les transitions associées peuvent être supprimés. Nous obtenons alors le tableau et le diagramme suivants :

•	a	b	c
1	2	4	0
2	5	3	2
3	6	4	0
4	0	0	5
5	5,6	0	0
6	0	0	0



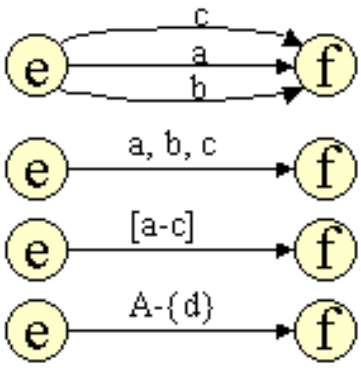
Nous verrons dans le suite que ce procédé de simplification d'automates peut être automatisé. Nous verrons les notions associées (état puit, source, accessible...) et les algorithmes adéquats (élagage...). Nous irons même plus loin en fusionnant les états jouant le même rôle.

Il existe aussi d'autres représentations possibles comme des matrices binaires associées à chacun des symboles...

Remarque. Dans certains cas, il est plus facile de représenter un groupe de transitions par une notation ensembliste. Par exemple, supposons que pour un état donné q_i, nous avons soit un passage à l'état q_j pour les symboles a_p et a_q soit une conservation de l'état courant pour les autres symboles de l'alphabet A. Ce type de transition s'écrit : $(a_p, q_i) \rightarrow q_j$; $(a_q, q_i) \rightarrow q_j$; $(a_1, q_i) \rightarrow q_i$; $(a_2, q_i) \rightarrow q_i$; ... Soit : $(a_j,q_i) \rightarrow q_i$, $\forall$ i $\notin$ {p,q}. Pour simplifier l'écriture, nous aurons : $(\{a_p, a_q\}, q_i) \rightarrow q_j$; $(A-\{a_p, a_q\}, q_i) \rightarrow q\_i$.

Plus généralement, les symboles peuvent être regroupés sous différentes classes et l'automate possède des transitions en fonction de ces classes. Nous retrouvons ici la même simplification que nous avions utilisée pour les expressions rationnelles (nous verrons que cette similitude n'est pas un hasard).

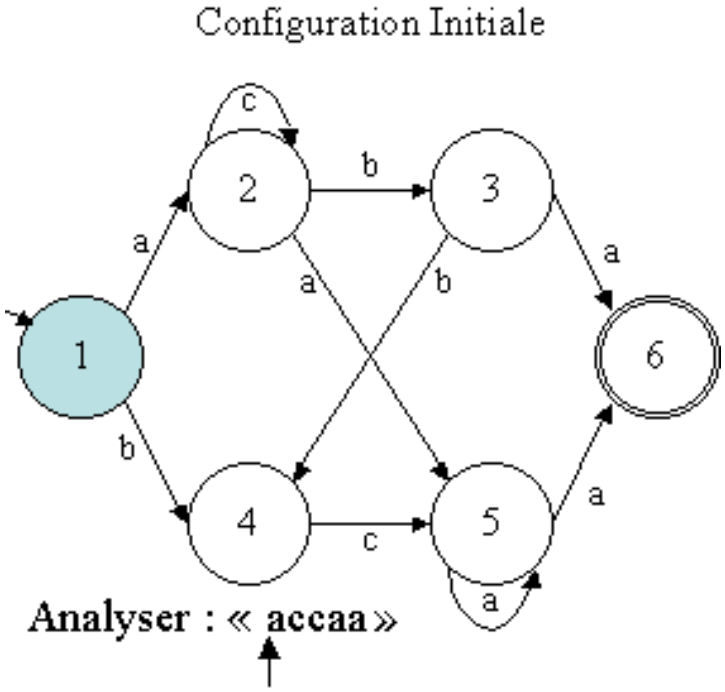
Cette simplification dans la syntaxe se répercute aussi au niveau du diagramme d'états. Les quatre notations suivantes sont équivalentes (en considérant un automate sur l'alphabet {a,b,c, d}) :



Mais attention, dans tous les cas, une transition ne concerne qu'UN SYMBOLE de l'alphabet. Les notations précédentes permettent simplement de simplifier la représentation de l'automate. Dans les 4, les trois dernières sont des "simplifications visuelles" de la première.

3.3. Exemple d'exécution d'un AFN

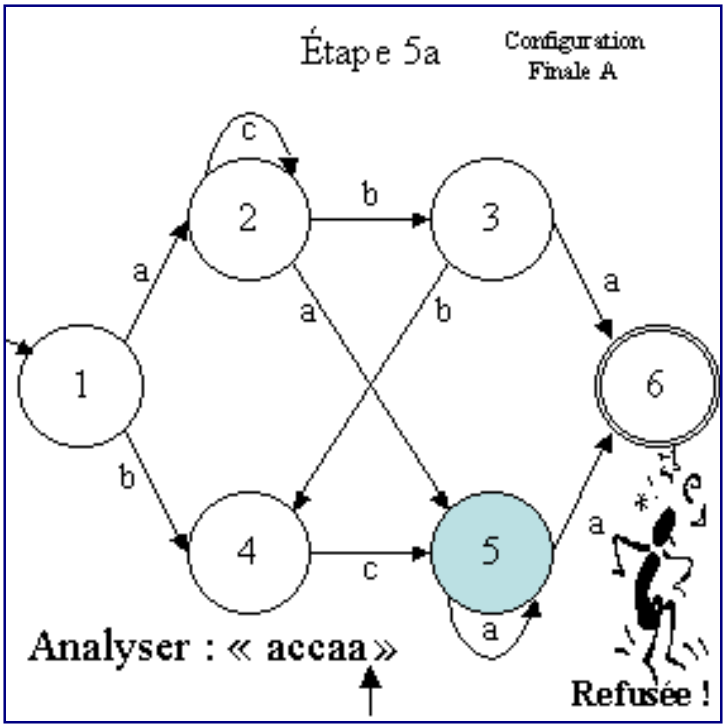
Nous avons vu qu'un automate fini est un cas particulier de machine de Turing. Il est donc possible de simuler le fonctionnement de cette machine. Pour cela, le plus courant est d'utiliser le graphe des transitions. Prenons par exemple l'automate T_1 = {{1,2,3,4,5,6},{1},{6}, {(a,1) → 2, (b,1) → 4, (a,2) → 5, (c,2) → 2, (b,2) → 3, (a,3) → 6, (b,3) → 4, (c,4) → 5, (a,5) → 5, (a,5) → 6}} et étudions le mot "accaa" sur cet automate. N'ayant qu'un seul état initial, la configuration initiale est schématisé par le graphe suivant (en bleu, l'état courant de la machine ; la flèche indique la position de la tête de lecture de la machine sur le mot) :



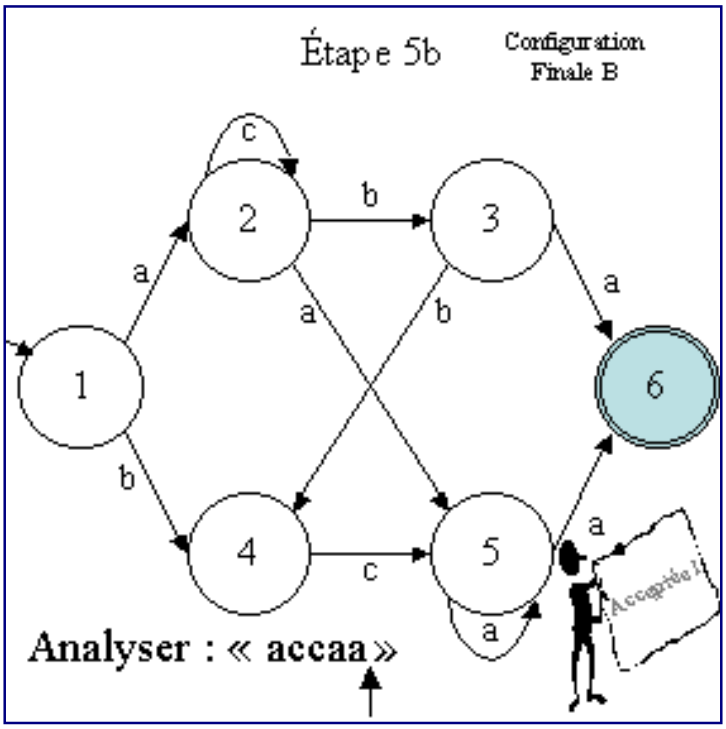
A partir de cette configuration initiale (état 1, symbole "a"), la seule transition possible pour faire évoluer la machine est celle de 1 vers 2 sur "a". Donc, la machine passe dans la

configuration (état 2, symbole "c"). De même, à nouveau, la seule transition possible est celle de 2 vers 2 sur "c". Donc, la machine passe dans la configuration (état 2, symbole "c"). De nouveau, la seule transition possible est celle de 2 vers 2 sur "c". Donc, la machine passe dans la configuration (état 2, symbole "a"). Maintenant, la seule transition possible est celle de 2 vers 5 sur "a". Donc, la machine passe dans la configuration suivante (état 5, symbole "a").

La configuration courante nous pose un problème. En effet, étant dans la configuration (état 5, symbole "a"), deux transitions peuvent être exploitées : celle de 5 vers 5 sur "a" ou celle de 5 vers 6 sur "a". Cette situation est une situation de non-déterminisme. Il faut donc choisir une transition en prévoyant éventuellement de revenir dans cette configuration pour en reprendre une autre. Prenons donc la transition de 5 vers 5. La machine passe alors dans la configuration (état 5, symbole "ε"). Or, 5 n'est pas un état final. La machine est donc dans une situation d'échec (plus de symbole et état non final).



Il convient donc de revenir en arrière pour reprendre la transition de 5 vers 6 dans la configuration (état 5, symbole "a"). La machine passe alors dans la configuration finale (état 6, symbole "ε").



Le mot proposé "accaa" est alors accepté par l'automate T_1. Il en est de même pour les mots "aba", "bcaaa", "acba", acbbcaaa"... Par contre, pour le mot "abaa" n'est pas accepté. En effet, au bout d'un certain nombre d'étapes, la machine se retrouve dans la configuration (état 6, symbole "a"). A ce moment, plus aucune transition n'est exploitable et il reste un symbole à étudier. La machine est en échec et le mot est refusé. De même, pour "abbb", la machine va se retrouver dans la configuration (état 4, symbole "b"). Dans cette configuration, à partir de 4, aucune transition sur "b" n'est disponible.

3.4. Configuration et action

Pour pouvoir exprimer de manière plus formelle le déroulement d'un programme sur un automate fini, il est nécessaire d'introduire les notions de configuration et d'action.

Configuration d'un AFN
Configuration d'arrêt
Configuration non-

Si T=(A, Q, I, F, •) est un AFN, alors tout couple (a, q) avec $a \in A^*$ et $q \in Q$ est une configuration de T. Contrairement à la notion de configuration d'une machine de Turing, a n'est pas un symbole mais un mot contenant les symboles restant à parcourir.

Une **configuration d'arrêt** est une configuration soit de la forme (ε, q) $\forall q \in Q$, $\forall a \in A^*$ soit de la forme (a, q) $\forall q \in Q$ telle qu'il n'existe pas de transition $(a, q) \rightarrow q'$ $\forall q' \in Q$, $\forall a \in A^*$.

Il existe trois ensembles de configurations particulières :

- les configurations (a, q_I) $\forall a \in A^*$ et $q_I \in I$ appelées

déterministe
Configuration déterministe

configurations initiales ou **configurations de départ** ;
- les configurations d'arrêt de la forme (ε, q_F) avec $q_F \in F$ appelées **configurations finales** ou **configurations terminales**.
- les autres configurations appelées **configurations intermédaires**.

Une **configuration non déterministe** est une configuration à partir de laquelle plusieurs transitions sont possibles. Dans le cas contraire, la configuration est dite **configuration déterministe**.

Les machines exécutant un automate fini qui, pour un mot m, terminent dans une configuration d'arrêt qui est une configuration finale sont en situation d'acceptation (le mot m est accepté par l'automate). Dans tous les autres cas de configuration d'arrêt, la reconnaissance échoue (le mot m est rejeté). En cas de non déterminisme, un mot est reconnu s'il existe au moins une configuration finale. Par contre, si toutes les configurations d'arrêt ne sont pas finales, la reconnaissance échoue.

Exemples, avec T_1 :

- ("accaa", 1) est une configuration initiale ;
- $(\varepsilon, 6)$ est une configuration finale ;
- ("caa", 2) est une configuration intermédiaire.

Une action représente le passage d'une configuration donnée d'un AFN à une autre par l'application d'une transition de l'automate. Bien évidemment, pour les AFNs, il peut exister plusieurs actions différentes pour une même configuration de départ (non déterminisme).

Action
Calcul

Une **action** (ou un **calcul**, un **déplacement**) de l'automate T=(A, Q, I, F, •) est représentée par une relation binaire sur les configurations notée $\text{—}T\text{→}$ ou $\text{—}1/T\text{→}$ (→ s'il n'y a pas d'ambiguïté sur l'automate). Alors la transition $\bullet(a, q_i)=q_j$ peut s'écrire $(aw, q_i) \text{—}1/T\text{→} (w, q_j)$.

On dit que $C \text{—}0/T\text{→} C'$ si et seulement si C=C' et $C \text{—}k/T\text{→} C_k$, $k \geq 1$ si et seulement si $\exists\ C_1, ..., C_{k-1}$ tels que $C_i \text{—}T\text{→} C_{i+1}\ \forall\ i, 0 \leq i < k$ et $C_0 = C$

Notations : $C \text{—}+/T\text{→} C'$ signifie $C \text{—}k/T\text{→} C'$ avec $k \geq 1$ et $C \text{—}*/T\text{→} C'$ signifie $C \text{—}k/T\text{→} C'$ avec $k \geq 0$.

suite d'actions
chemin d'actions

Nous appellerons **suite d'actions** (ou de configurations) ou **chemin d'actions** les différentes actions consécutives permettant de passer d'une configuration à une autre.

Par exemple, si on reprend l'automate T1 avec le mot "accaa", nous obtenons les suites d'actions suivantes :

- (« accaa »,1) → (« ccaa »,2) → (« caa »,2) → (« aa »,2) → (« a »,5) configuration non-déterministe :
  - (« a »,5) → (ε,5) cette configuration d'arrêt n'étant pas une configuration finale, le mot n'est pas, pour l'instant, reconnu ;
  - (« a »,5) → (ε,6) cette configuration d'arrêt étant une configuration finale, le mot est reconnu.

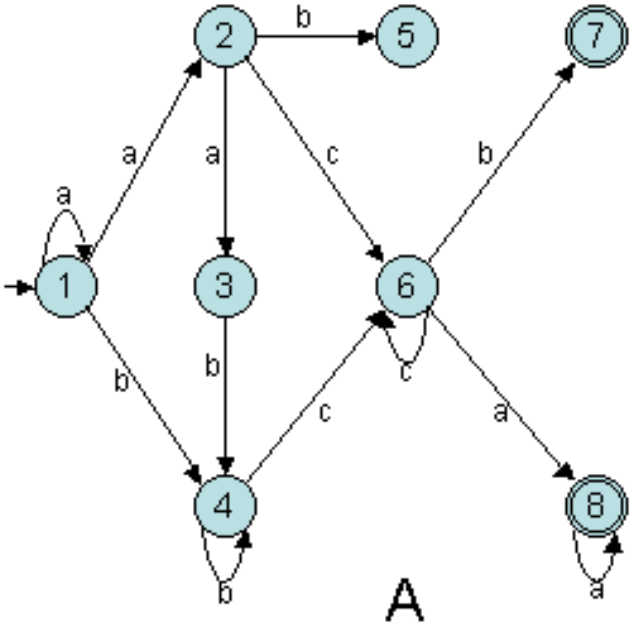
Donc (« accaa »,1) $—*/T_1→$ (ε,6) (OK) et (« accaa »,1) $—*/T_1→$ (ε,5) (ECHEC)

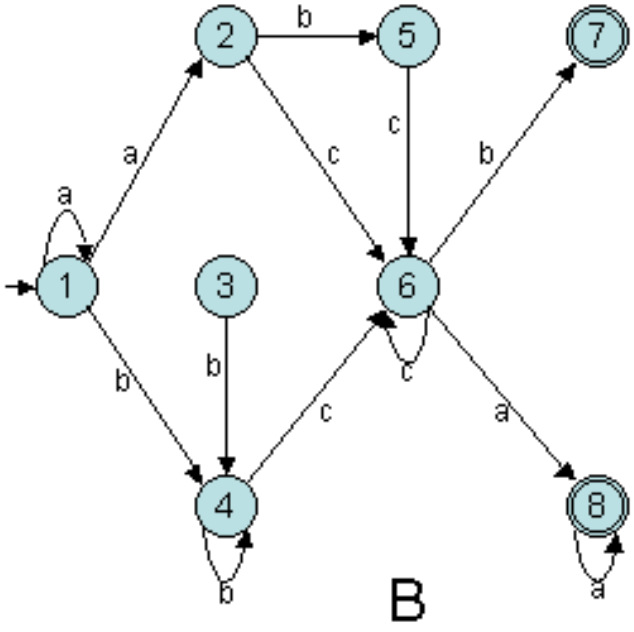
Remarques :

- $—+/T→$ est la fermeture transitive de $—T→$ et $—*/T→$ est la fermeture réflexive et transitive de $—T→$.
- Parfois, il est possible de trouver l'utilisation d'une fonction partielle récursive $•*$ définie à partir de la fonction de transition • sur $A^* \times Q \to Q$ telle que : $•*(ε,q)=q$ et $•*(aw, q) = •*(w,•(a,q))$.
  Cette fonction est équivalente à la notion d'action : avec $a \in A$ et $w \in A^*$,
  $•*(aw,q) = q' \equiv (aw,q) —1/T→ (w,q')$
  et $•*(w,I) = q_f \equiv (w,I) —*/T→ (ε,q_f)$.

Exercices et tests :

Exercice 3.1. Donnez toutes les suites d'actions possibles pour chacun des mots suivants avec les automates finis A et B (figures ci-dessous, alphabet {a,b,c}) et en déduire s'ils sont reconnus : "ε", "a", "aab", "bcb", "abca", "acac" et "aabbcb".

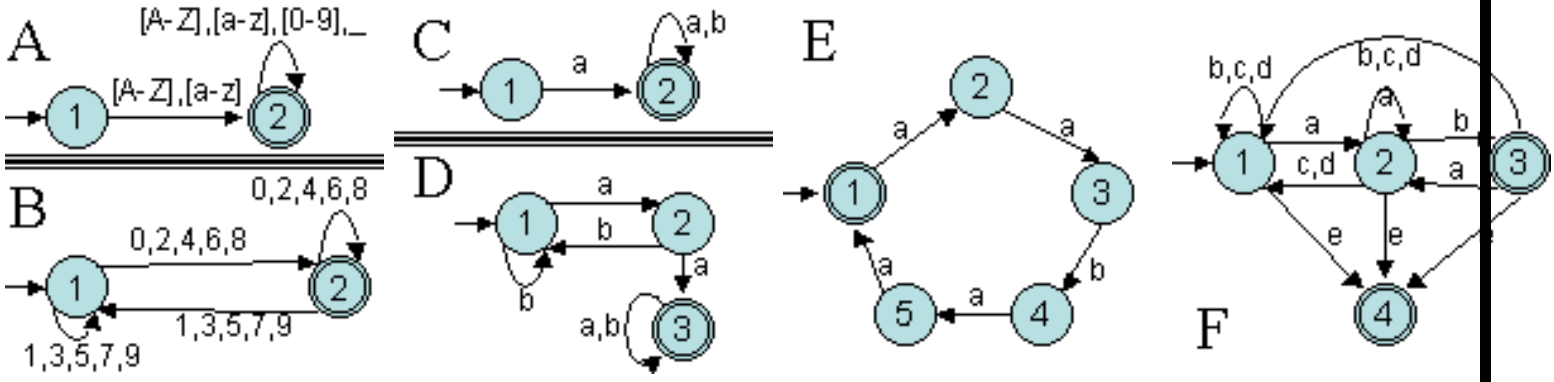




Exercice 3.2. Donner les automates finis (alphabet, liste des états, états initiaux, états finaux et transitions sous forme de table ou de diagramme) qui reconnaissent les mots suivants :

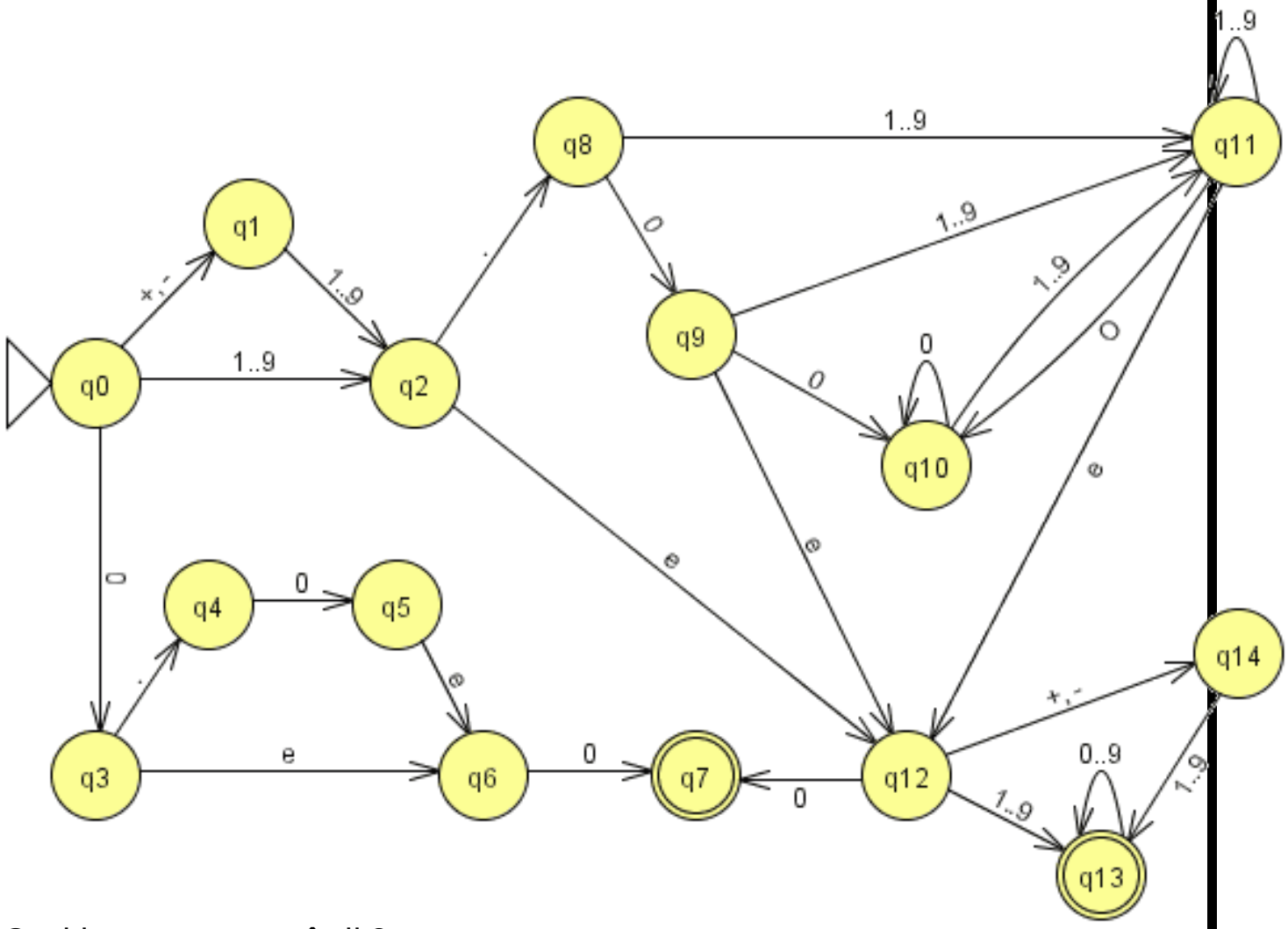
1. les mots terminés par "er", "é", "ée", "és" et "ées" ; 
2. les mots contenant au moins deux "a" sur l'alphabet {a,b,c} :
    a. consécutifs
    b. par forcément consécutifs

Exercice 3.3. Décrire les mots reconnus par les automates suivants :



Exercice 3.4. Écrire un programme permettant de simuler le fonctionnement d'une machine de Turing basée sur un AFD :

1. Proposer l'algorithme général de reconnaissance par un automate d'état fini déterministe quelconque dont le nombre d'états ne dépasse pas 50 états. .
2. Étudier l'automate suivant :



3. Quel langage reconnaît-il ?
4. Appliquer l'algorithme proposé en 1 à cet automate (l'automate sera "en dur" dans le programme ou chargé à partir d'un fichier) pour obtenir un programme permettant de reconnaitre les mots du langage décrit par l'automate.

Exercice 3.5. Donner les automates finis qui reconnaissent les mots des langages suivants sur le vocabulaire {a,b} :

1. le langage des mots avec un nombre pair de "a" et un nombre impair de "b" ;
2. le langage des mots ne contenant pas deux occurrences de "a" successives.

Exercice 3.6. Donner un automate reconnaissant le langage dont les mots sont des entiers multiples de 3 ({w ∈ A* tq w mod 3 = 0}).

4. Les automates finis généralisés

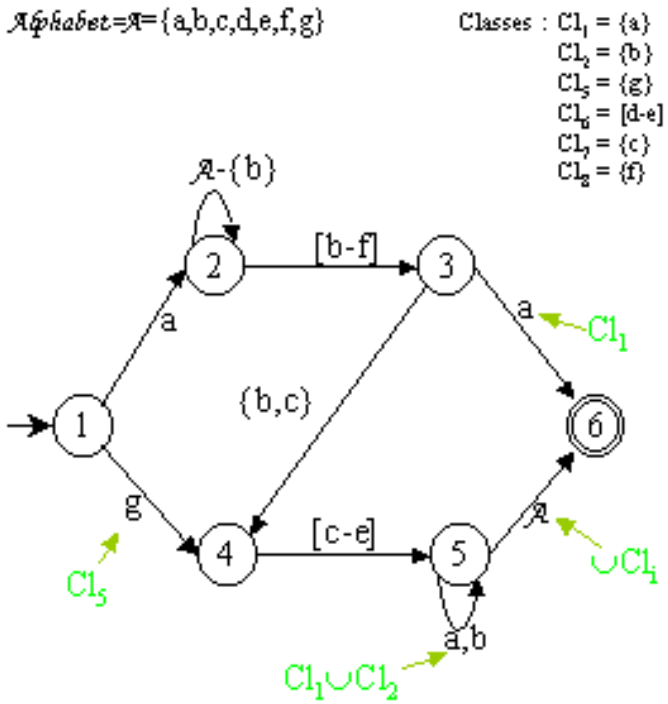
4.1. Principes

Les AFNs sont très utilisés pour les traitements séquentiels sur des suites de symboles (mots, phrases, suites d'actions...) à cause de la facilité de mise en oeuvre du procédé et des avantages qu'ils offrent dans le cas de modifications ou d'extensions d'un traitement.

Cependant, on utilise souvent des AFNs généralisés qui ne se contentent pas de reconnaître des chaînes de symboles, mais éventuellement qui les traitent au passage. Les généralisations classiques sont les suivantes :

1. la tête de lecture reconnaît une classe de symboles (au minimum, une par symbole et toutes disjointes), et pas seulement un symbole (comme nous l'avons vu précédemment) ;
2. un traitement particulier peut être associé à un état de la machine, ou à une transition, visant soit à produire un résultat, soit à modifier l'état de l'automate ;
3. un traitement d'erreur est souvent prévu en cas de mauvaise analyse d'un symbole.

Dans le cadre des automates généralisés (cas 1), il est donc possible de simplifier les transitions en introduisant la notion de classe de symboles. La figure suivante montre comment les utiliser :



Automate avec classes de symboles

Une application illustrant cette généralisation concerne les "transducteurs" ("Machines séquentielles"). Un transducteur permet de réaliser une transduction de l'alphabet d'entrée vers un alphabet de sortie. Un transducteur est aussi appelé "Machine de Moore" lorsque la sortie est produite pour chaque état et "Machine de Mealy" lorsqu'elle est produite pour chaque transition.

4.2. Machine de Moore

Une machine de Moore est donc définie de la manière suivante :

Machine de Moore

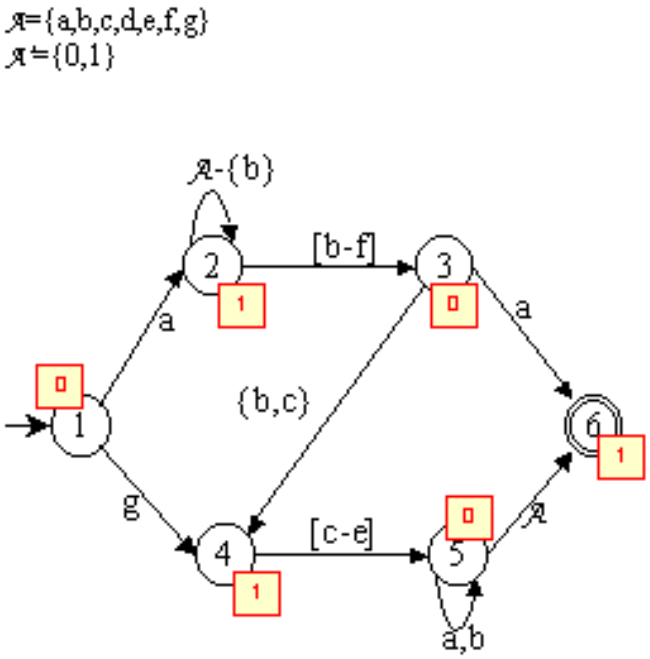
Une **machine de Moore** est composée de la même manière qu'un automate d'état fini. Elle est défini par (A, A', Q, I, F, •, •') tel que :

- A est **l'alphabet d'entrée**, ensemble fini, non vide de symboles ;
- A' est **l'alphabet de sortie**, ensemble fini de symboles ;
- Q est **l'ensemble des états** possibles pour la machine (fini et non vide) ;
- $I \subseteq Q$ est l'ensemble des états initiaux ou états de départ ;
- $F \subseteq Q$ correspond aux états finaux ou états d'acceptation ;
- • la fonction de transition telle que • : $A \cup \{\varepsilon\} \times Q \rightarrow Q$. Pour la configuration courante d'état et de symbole, •(a_j,

$q_i)=q_k$ signifie qu'on passe dans l'état q_k et que le ruban est déplacé d'une case vers la gauche.

- •' une fonction de sortie telle que •' : Q → A'. •'$(q_i)=a_j$ indique que le caractère a_j est émis (ou imprimé) en entrant dans l'état q_i.

Prenons comme exemple la machine suivante ({a,b,c,d,e,f,g},{1,2,3,4,5,6},{1},{6}, •, {•'(1)=0, •'(2)) 1, •'(3)=0, •'(4)= 1, •'(5)=0, •'(6)=1}) avec • décrite par le graphe de transitions suivant :



Machine de Moore

Sur cette machine, les mots suivants sont acceptés et donnent :

- "abccaa" = "0101001"
- "aeccaa" = "0101001"
- "geba" = "01001"

4.3. Machine de Mealy

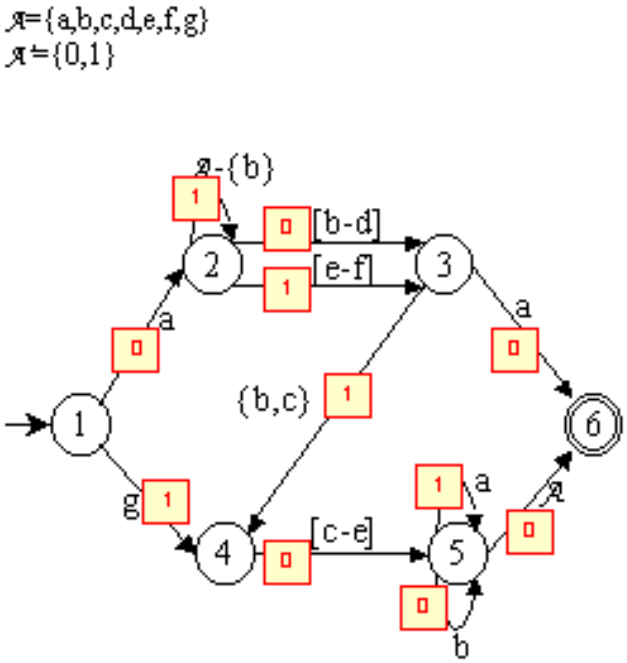
Une machine de Mealy est définie de la manière suivante :

Machine de Mealy

Une **machine de Mealy** est composée de la même manière qu'un automate d'état fini. Elle est défini par (A, A', Q, I, F, •) tel que :

- A est **l'alphabet d'entrée**, ensemble fini, non vide de symboles ;
- A' est **l'alphabet de sortie**, ensemble fini de symboles ;
- Q est **l'ensemble des états** possibles pour la machine (fini et non vide) ;
- $I \subseteq Q$ est l'ensemble des états initiaux ou états de départ ;
- $F \subseteq Q$ correspond aux états finaux ou états d'acceptation ;
- • la fonction de transition telle que • : $A \cup \{\varepsilon\} \times Q \rightarrow Q \times A'$. Pour la configuration courante d'état et de symbole, • $(a_j,q_i)=(q_k,a_l')$ signifie qu'on passe dans l'état q_k, que le symbole a_l' est généré (imprimé) et que le ruban est déplacé d'une case vers la gauche.

Prenons comme exemple la machine suivante ({a,b,c,d,e,f,g},{1,2,3,4,5,6},{1},{6}, •) avec • décrite par le graphe de transitions suivant :



Machine de Mealy

Sur cette machine, les mots suivants sont acceptés et donnent :

- "abccaa" = "001010"
- "aeccaa" = "011010"
- "geba" = "1000"

4.4 Equivalence des machines

Equivalence entre Machine de Mealy et Machine de Moore

Soit une machine de Moore, notée Mo, qui imprime x au départ et une machine de Mealy, notée Me. Les **deux machines sont équivalentes**, Mo ≡ Me, si pour tout mot d'entrée, Me imprime w et Mo imprime xw.

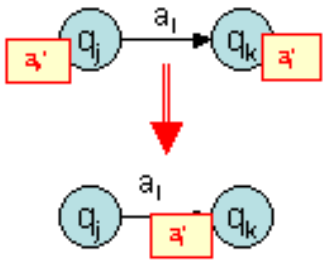
A partir de cette définition, deux théorèmes sont démontrables. tout d'abord :

Théorème Moore->Mealy

Pour toute machine de Moore, il existe une machine de Mealy tel que les deux machines sont équivalentes.

Démonstration :

Soit Mo= (A, A', Q, I, F, $\bullet_o$, $\bullet_o$') et Me = (A, A', Q, I, F, $\bullet_e$).
Si $\exists a_i \in A$ et $q_j, q_k \in Q$ tels que $\bullet_o(a_i,q_j)= q_k$ et $\bullet_o'(q_k) = a_i'$ avec $a_i' \in A'$ alors $\exists \bullet_e(a_i,q_j) = (q_k, a_i')$ dans Me .



Puis :

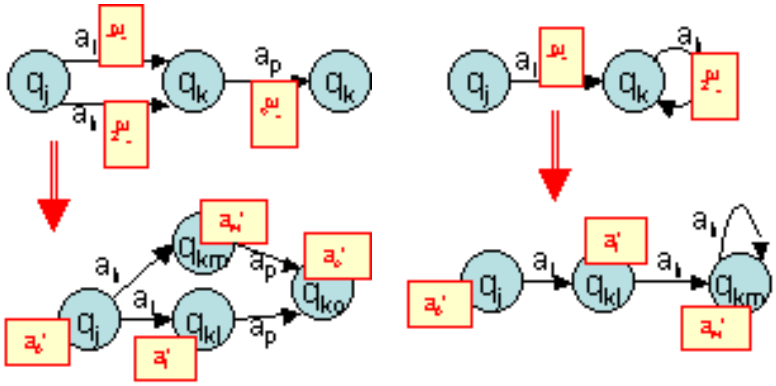
Théorème Mealy->Moore

Pour toute machine de Mealy, il existe une machine de Moore tel que les deux machines sont équivalentes.

Démonstration :

Soit Mo= (A, A', Q, I, F, $\bullet_o$, $\bullet_o$') et Me = (A, A', Q', I', F', $\bullet_e$).
Si $\exists a_i \in A, a_l \in A'$et $q_j, q_k \in Q'$ tels que $\bullet_e(a_i,q_j)= (q_k,a_l)$ dans Me alors :

- $\forall\ q_k \in Q', \forall$ l telle que $\bullet_e(a_i,q_j)= (q_k,a_l)$ on fait correspondre $\{q_{kl}\} \subseteq Q$
- si q_k tst tel que $\forall$ j, $\bullet_e(a_i,q_j) \neq (q_k,a_l)$ alors $q_{k0} \in Q$ et $\bullet_o'(q_{k0}) = a_0$ avec $a_0 \in A'$ dans Mo
- si $\exists a_i \in A, a_l \in A'$et $q_j, q_k \in Q'$ tels que $\bullet_e(a_i,q_j)= (q_k,a_l)$ dans Me$\forall x$ $\bullet_o(a_i,q_{jx}) =q_{kl}$ et $\bullet_o'(q_{kl}) = a_l$ dans Mo.
- si $q_j \in F'$ alors $\forall x\ q_{jx} \in F$.
- si $q_j \in I'$ alors $\forall x\ q_{jx} \in I$.



Exercices et tests :

Exercice 4.1. Donner la machine de Mealy permettant de calculer le complément binaire. Par exemple, "101" donne "010", "001010" donne "110101"...

Exercice 4.2. Ecrire l'automate généralisé permettant de compter le nombre d'occurrences du facteur "ac" pour des mots de l'alphabet A={a,b,c} contenant au moins deux "a" consécutifs.

5. AFN et langages

Les machines de Turing (et donc les AFNs), nous l'avons vu, sont aussi appelées "accepteurs" ("Language Acceptor") lorsqu'elles sont utilisées pour reconnaître un langage donné en Théorie de Langages. Un automate fini sert à reconnaître un langage bien particulier. Il est donc possible de définir le **langage reconnu par un automate donné**. Pour cela, nous utilisons les notions de **langage entre deux états**, de **langage gauche** et de **langage droit**.

Un langage entre deux états correspond aux différentes successions de symboles (aux différents mots) permettant de passer d'un état donné à un autre.

Langage entre état

Un **langage entre deux états** quelconques q_0 et q_1 de T=(A, Q, I, F, •) est défini par : $l_T(q_0, q_1) = \{w \mid w \in A^*, \forall x \in A^*, (wx,q_0) —*/T\rightarrow (x,q_1)\}$.

Le langage gauche d'un état est l'ensemble des suites de symboles (l'ensemble des mots) permettant d'atteindre cet état à partir des états initiaux. Le langage droit d'un état est celui permettant, à partir de cet état, d'atteindre les états finaux.

Langage droit & Langage gauche

Le **langage gauche** d'un état $q \in Q$ d'un AFN T=(A, Q, I, F, •) est donné par la fonction $LG_T : Q \rightarrow A^*$ où $LG_T(q) = \cup_{i\in I}\ \underline{l_T}(i,q) = \{w \mid w \in A^*, x \in A^*, \exists i \in I, (wx,i) —*/T\rightarrow (x,q)\}$

Le **langage droit** d'un état $q \in Q$ d'un AFN T=(A, Q, I, F, •) est donné par la fonction $LD_T : Q \rightarrow A^*$ où $LD_T(q) = \cup_{f\in F}\ \underline{l_T}(q,f) = \{w \mid w \in A^*, \exists f \in F, (w,q) —*/T\rightarrow (\varepsilon,f)\}$.

Un langage reconnu par un AFN est donc l'ensemble des suites de symboles (l'ensemble des mots) permettant d'atteindre un état final à partir d'un état initial.

Langage d'un AFN
Langage reconnaissable
Rec(A*)

Un langage reconnu (accepté, défini) par un AFN T=(A, Q, I, F, •) est donné par la fonction $L : AFN \rightarrow A^*$ où $L(T) = \cup_{f\in F}\ \underline{LG_T}(f) = \cup_{i\in I}\ \underline{LD_T}(i) = \cup_{i\in I, f\in F}\ \underline{l_T}(i,f)$. L(T) est donc l'ensemble des chaînes acceptées par T, c'est-à-dire : $L(T) = \{w \mid w \in A^*, \exists\ f \in F, i \in I : (w,i) —*/T\rightarrow (\varepsilon,f)\}$.

Un langage L sur un alphabet A est dit **langage reconnaissable**, noté **L ∈ Rec(A*)**, s'il existe un automate fini T tel que L = L(T).

Attention, un langage est reconnaissable si l'on peut construire un AFN qui reconnait TOUS les mots du langage et UNIQUEMENT les mots du langage.

Il est alors possible alors de comparer deux automates à partir des langages qu'ils reconnaissent.

Automates équivalents

Deux **automates** sont **équivalents** s'ils acceptent le même langage, c'est-à-dire :
$T_1 \equiv T_2$ si et seulement si $L(T_1) = L(T_2)$.

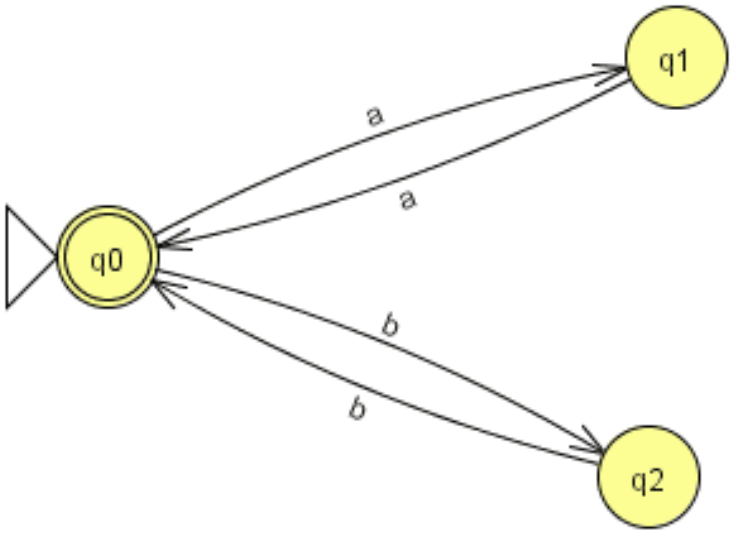
Exercices et tests :

Exercice 5.1. Montrer que les langages suivants sont reconnaissables

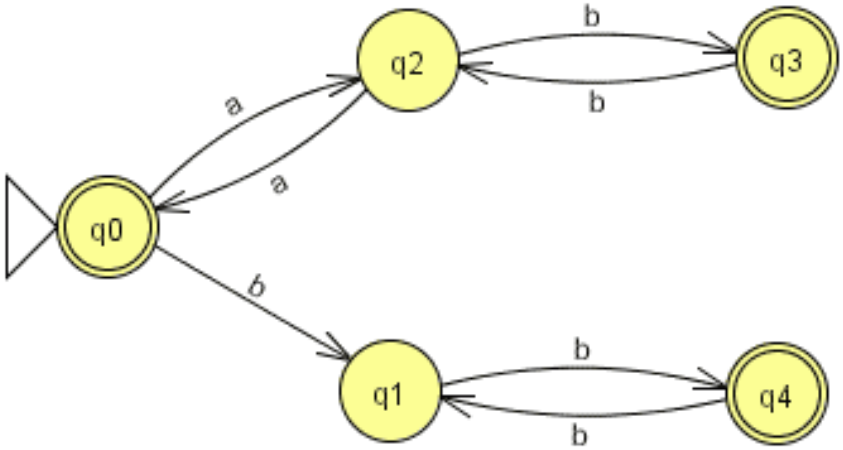
- Le langage des entiers signés ;
- Le langage des mots qui contiennent au moins un facteur "ab" et un facteur "ba" sur l'alphabet {a,b}.

Exercice 5.2. Parmi les automates suivants, quel est celui reconnaissant les mots du langage $\{a^n b^m \mid n \bmod 2 = m \bmod 2\}$, autrement dit le langage avec d'abord des "a" et ensuite des "b" et tel que le nombre de "a" et le nombre de "b" ont même parité (images issues de JFLAG) :

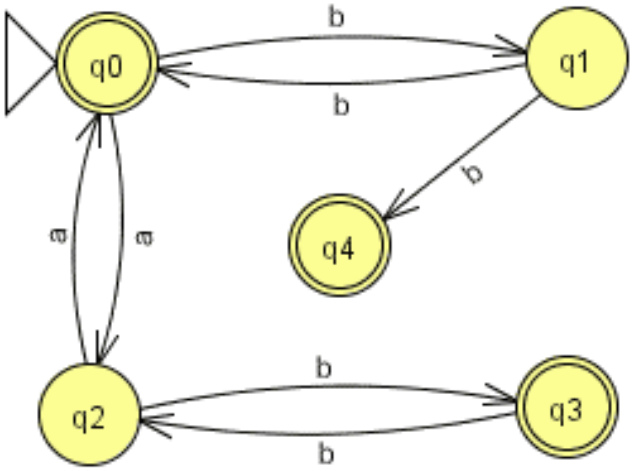
1.



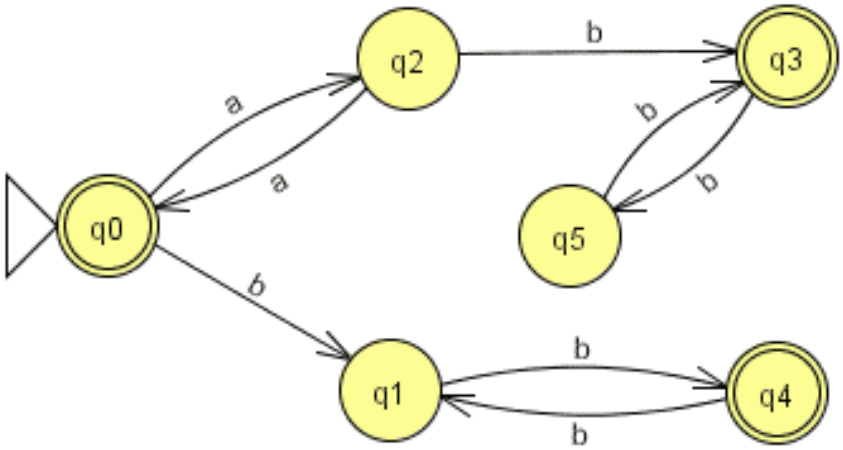
2.

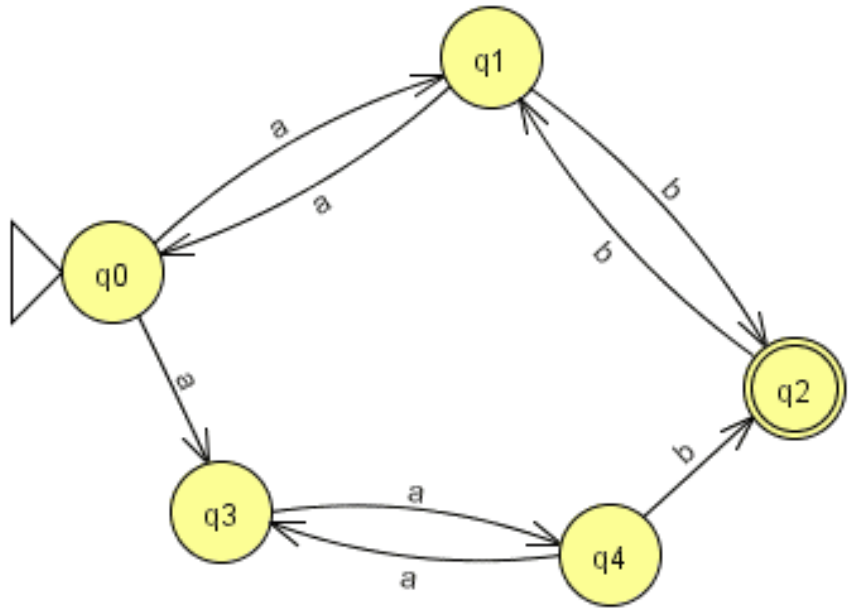

q0
q1
q2
q3
q4
a
a
b
b
b
b
b

3.


q0
q1
q2
q3
q4
a
a
b
b
b
b
b

4.


q0
q1
q2
q3
q4
q5
a
a
b
b
b
b
b
b



5.

6. Propriétés des états

Outre les états finaux et l'état initial, il existe d'autres catégories d'états spécifiques : les états stériles, accessibles, isolés...

Etat stérile
Etat co-accessible

Un **état stérile** (état mort ou <<dead state>>, <<trap state>>) e est un état pour lequel il n'est pas possible d'atteindre un état final, c'est-à-dire si : LD(e) = ∅. En théorie des graphes, ils sont aussi appelés <<noeuds puits>>.

Un état qui n'est pas stérile (ie LD(e) ≠ ∅) est dit **état co-accessible**.

Bien sûr, les états stériles sont à éviter lors de la construction d'un automate. De même, les noeuds non initiaux pour lesquels aucune transition n'aboutit (appelés "noeuds sources" en théorie des graphes) ne sont pas utiles, et sont donc à éviter.

Etat accessible
Etat inaccessible

Un état e est dit **accessible depuis un état** f pour un AFN T=(A, Q, I, F, •) si ∃ wx ∈ A* telle que (wx,f) —*/T→ (x,e).

Un état e est dit **accessible** pour un AFN T=(A, Q, I, F, •) si ∃ wx ∈ A* et i ∈ I tels que (wx, i) —*/T→ (x,e) (il existe au moins un chemin de i à e, c'est-à-dire LG(e) ≠ ∅).

Dans le cas contraire, l'état e est dit **inaccessible** (il n'existe pas de chemin de i ∈ I à e, c'est-à-dire LG(e) = ∅).

L'état ne faisant l'objet d'aucune transition est dit isolé et n'est, lui aussi, d'aucune utilité.

Etat isolé

Un état e est dit **isolé** s'il ne participe à aucune transition.

Plus généralement, tout état qui ne peut être atteint depuis l'état initial en suivant les transitions ou qui est stérile est à supprimer. Autrement dit, un état (et les transitions qui le concernent) peut être supprimé de l'automate s'il est inutile, c'est-à-dire inaccessible ou stérile.

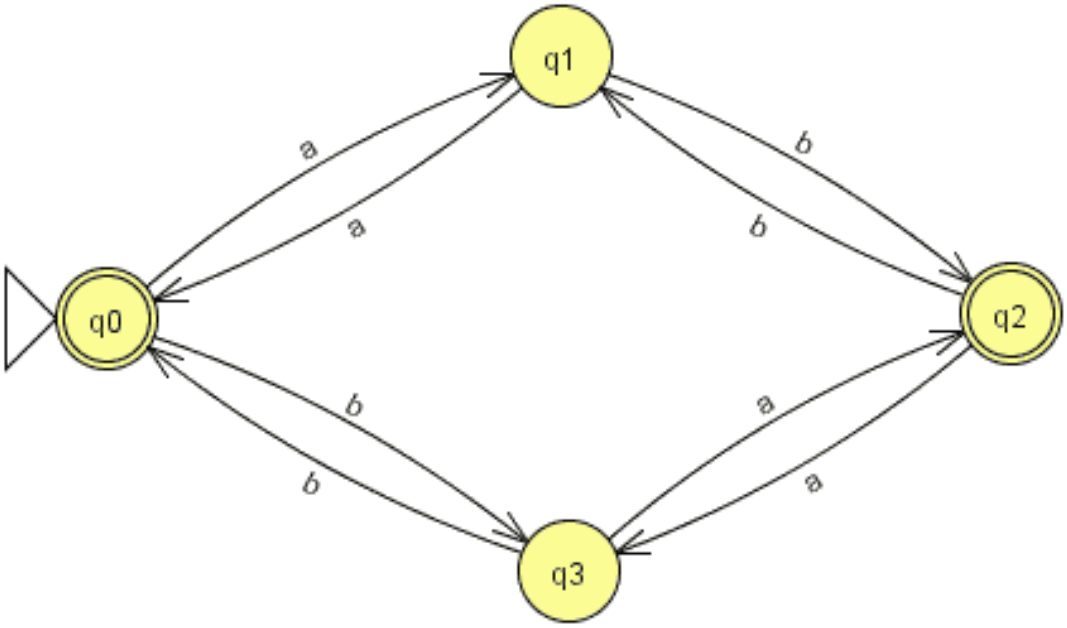
Etat utile

Soit un AFN T=(A, Q, I, F, •) et un état e ∈ Q :

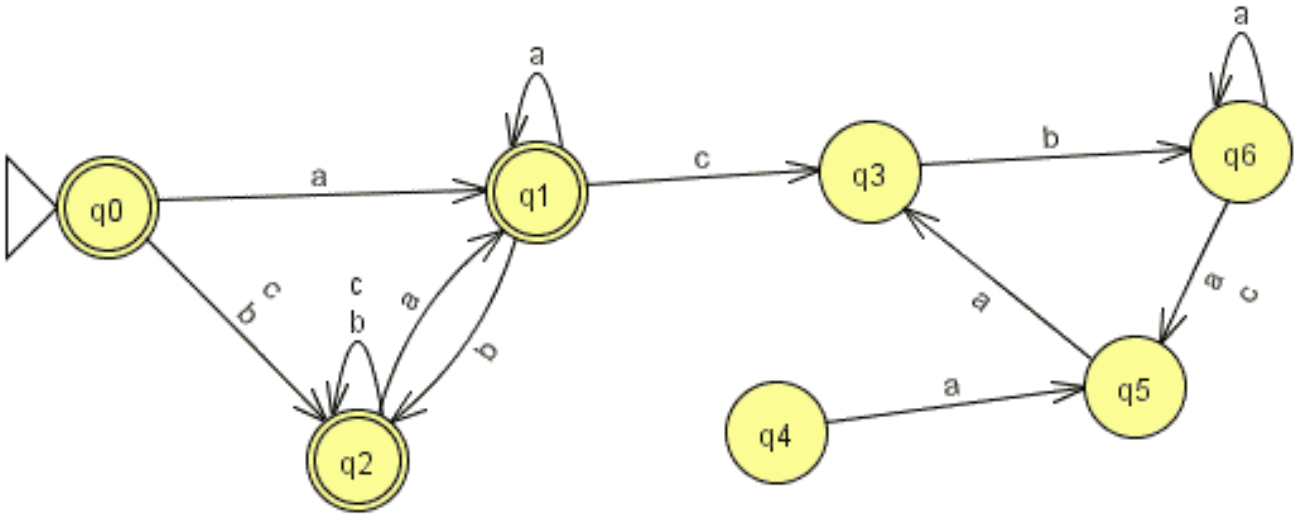
- e est **utile$_I$** s'il est accessible ;
- e est **utile$_F$** s'il est co-accessible ;
- e est **utile** si : utile$_I$(e) ∧ utile$_F$(e) ≡ Accessible(e) ∧ co-accessible(e). Dans le cas contraire, l'état est dit **inutile**.

Exercices et tests :

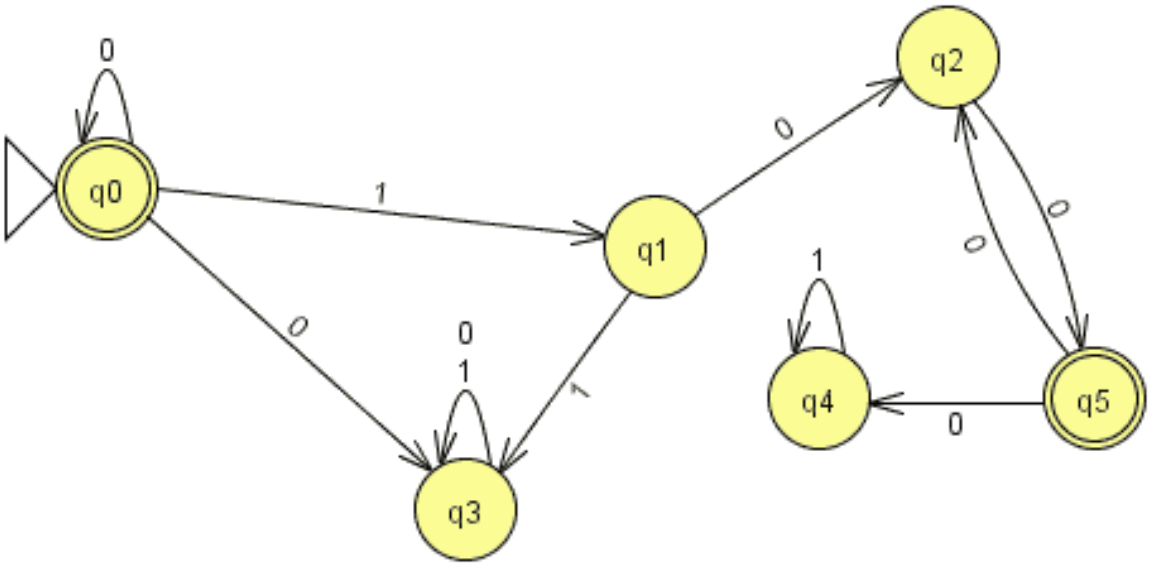
Exercice 6.1. Donner les propriétés des états pour les automates suivants (images issues de JFLAG) :

1. 

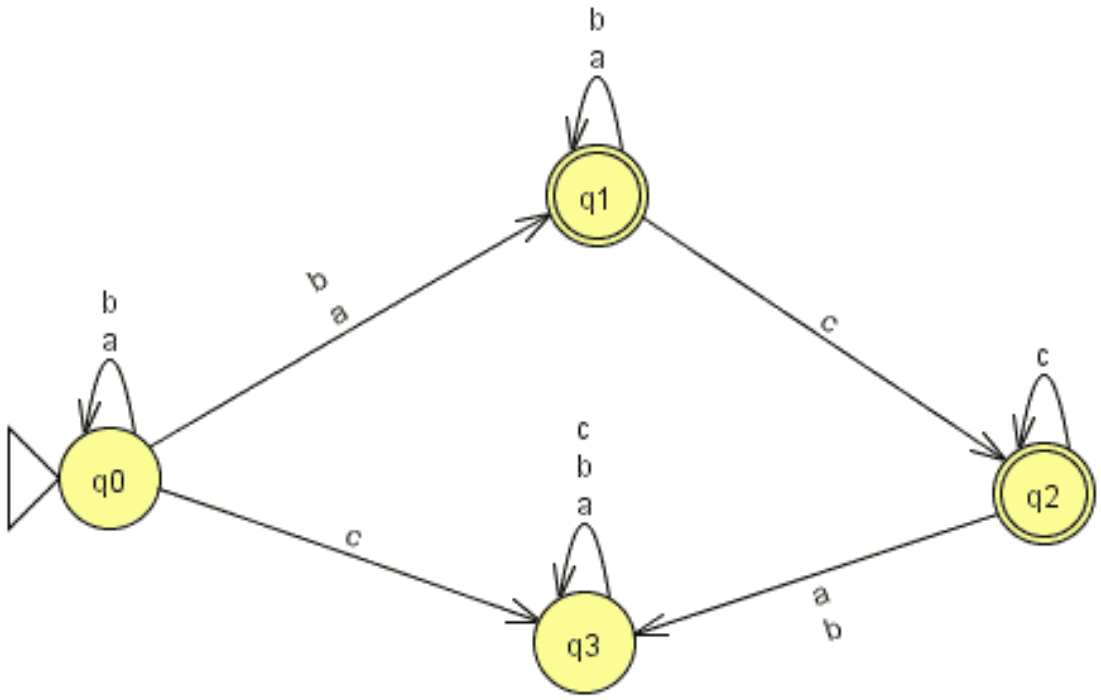
sur l'alphabet {a,b}

2. 

sur l'alphabet {a,b,c,d}

3. 

sur l'alphabet {0,1,2,3,4,5,6,7,8,9}



4.
sur l'alphabet {a,b,c,d}

7. Propriétés des AFNs

7.1. Dimension d'un automate

Outre le fait qu'un automate soit ou non déterministe, il est possible de mettre en évidence d'autres caractéristiques. La dimension d'un automate fini correspond simplement au nombre d'états qu'il possède. Cette dimension peut être minimale lorsqu'il n'est plus possible de trouver un automate équivalent de dimension strictement inférieure.

Dimension d'un AFN
AFN minimal

La **dimension** d'un AFN T=(A, Q, I, F, •), notée |T| est définie par :
|T | = |Q|. Autrement dit, la dimension d'un automate est le nombre d'états de cet automate.

Un **automate de dimension minimale** est un automate T tel que ∀ T', T' ≡ T alors |T'| ≥ |T|.

7.2. Propriétés liées aux caractéristiques des états

Un automate fini complet est un automate pour lequel, pour toute configuration, il existe une transition vers un état de cet automate. La fonction d'appartenance est totalement définie

(pour tout couple (symbole courant, état courant)).

AFN complet

Un **AFN complet** T=(A, Q, I, F, •) est un AFN vérifiant : Complet(T) ≡ (•(a,q) ≠ 0, ∀ q ∈ Q, ∀ a ∈ A) avec 0 l'état d'erreur.

Remarque : si Complet(T) alors {∪ LG(q), ∀ q ∈ Q} = A*.

Pour rendre un automate complet (la complétion), il suffit d'ajouter un état puit "p" et d'y diriger toutes les transitions manquantes. Si T={A, Q, I, F, •} alors Completion(T) = {A, Q ∪ {p}, I, F, •'} avec •' = • ∪ {(q,a,p) : ∀ q' ∈ Q, (q,a,q') ∉ •} ∪ {(p,a,p) : ∀ a}. Le langage n'est pas modifié car on ajoute un état qui n'est pas final et des transitions initialement absentes vers cet état.

Un automate fini est utile si tout état de cet automate possède un langage gauche et un langage droit, c'est-à-dire si tout état de l'automate peut être utilisé pour reconnaître au moins un mot du langage.

AFN utile
AFN émondé

Soit un AFN T=(A, Q, I, F, •) :

- $Utile_I$ est défini par : $Utile_I(T) \equiv utile_I(q), \forall q \in Q$;
- $Utile_F$ est défini par :$Utile_F(T) \equiv utile_F(q), \forall q \in Q$;

Un **AFN émondé** (ou utile) est un AFN dont tous les états sont utiles. Donc : Emondé(T) ≡ $Utile_I(T) \wedge Utile_F(T) \equiv utile(q), \forall q \in Q$.

Il est possible d'en déduire une relation entre langage reconnaissable et automate émondé.

Théorème de l'émondage

Pour tout langage reconnaissable L sur un alphabet A (ie L ∈ Rec (A*)) il existe un automate émondé qui le reconnait.

Démonstration :

La démonstration se fait par construction de l'automate émondé à partir d'un AFN T=(A, Q, I, F, •) qui reconnaît le langage (algorithme de l'émondage) :

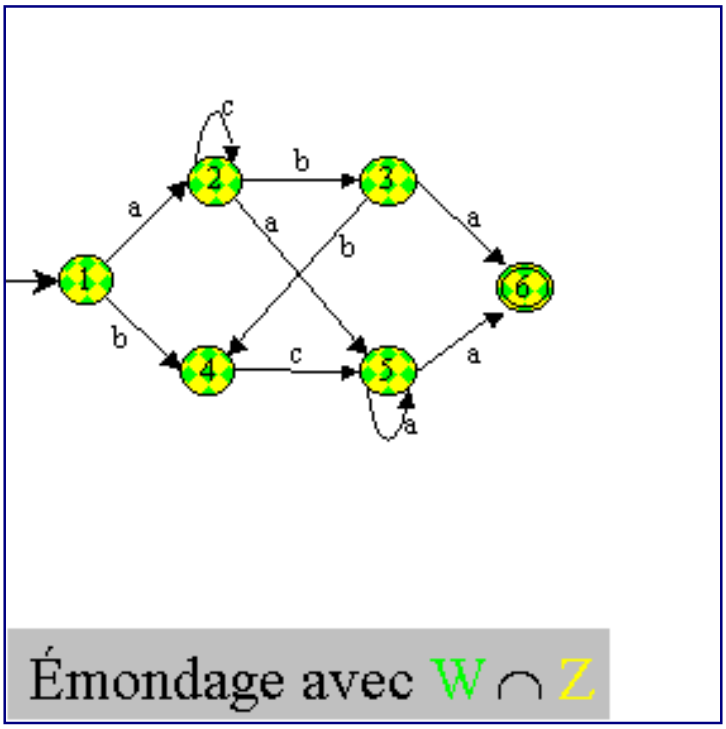
1. Détermination de l'ensemble w des états accessibles

a. Soit w_i l'ensemble des états accessibles à l'instant i.
   Au départ, les seuls états accessibles sont les états initiaux, donc $w_0 = I$
b. Tout état q est accessible s'il existe une transition $\bullet(a,p)=q$ avec p lui-même accessible. Donc,
   $w_{i+1} = w_i \cup \{q \mid q \in Q, \exists\ p \in w_i \text{ et } \exists\ a \in A \text{ tels que } \bullet(a,p)=q\}$
c. C'est terminé quand $w_{i+1} = w_i$ donc $w = w_i$

2. Détermination de l'ensemble z des états co-accessibles
   a. Soit z_i l'ensemble des états co-accessibles à l'instant i. Au départ, seuls les états finaux sont co-accessibles, donc $z_0 = F$
   b. Tout état q est co-accessible s'il existe une transition $\bullet(a,q)=p$ avec p lui-même co-accessible. Donc,
      $z_{i+1} = z_i \cup \{q \mid q \in Q, \exists\ p \in z_i \text{ et } \exists\ a \in A \text{ tels que } \bullet(a,q)=p\}$
   c. C'est terminé quand $z_{i+1} = z_i$ donc $z = z_i$
3. Construction de l'automate émondé T' :
   $T'=(A, Q' = w \cap z, I' = I \cap w \cap z, F' = F \cap w \cap z, \bullet')$ avec $\bullet' = \{(q,a,q') \mid (q,a,q') \in \bullet, q, q' \in Q'\}$

L'automate obtenu est équivalent à l'automate d'origine. Pour cela, il est possible d'avancer quelques éléments de preuve. Soit $q \in Q$, $i \in I$, $f \in F$ et $x,y \in A^*$ alors $\forall\ w=xy \in L(T)$: $(xy,i) \text{-}^*\!\rightarrow (y,q) \text{-}^*\!\rightarrow (\varepsilon,f)$
Donc les états qui participent à une telle suite d'actions sont accessibles et co-accessibles. Tout état ne participant pas à de telles suites n'est pas utile.
L'algorithme précédent permet de localiser les états participants à des suites d'actions.

Sur un exemple, cela donne les étapes suivantes :



Remarque : il est possible d'améliorer un peu cet algorithme en initialisant z_0 non pas à F mais à F ∩ w. De plus, on ne met dans z que les états étant déjà dans w. Du coup, au final, Q'=z.

7.3. Propriétés sur la structure de l'automate

Un autre ensemble de propriétés concerne le nombre des états initiaux et finaux ainsi que les transitions qui les concernent. Nous avons alors les définitions suivantes :

AFN unitaire
AFN standard
AFN normalisé
AFN homogène

Un AFN T=(A, Q, I, F, •) est dit **unitaire** s'il ne possède qu'un seul état initial, c'est-à-dire que I={i} avec i ∈ Q.

Un AFN T=(A, Q, I, F, •) est dit **standard** s'il est unitaire (I={i}) et si {•(a,q)=i | x ∈ A, q ∈ Q, I={i}} = ∅. Autrement dit, aucune transition arrive sur le seul état initial.

Un AFN T=(A, Q, I, F, •) est dit **normalisé** s'il est standard et si F={f] avec f ∈ F et {•(a,f)=q | x ∈ A, q ∈ Q, I={i}} = ∅. Autrement dit, s'il est standard et ne possède qu'un seul état final et qu'aucune transition n'a pour origine cet état final.

Un AFN T=(A, Q, I, F, •) est dit **homogène** si toutes les transitions qui arrivent sur un état sont sur le même symbole de l'alphabet.

Les propriétés "standard" et "normalisé" ne sont pas limitatives en ce qui concerne l'automate

reconnu. En effet, tout langage reconnaissable peut l'être par un automate standard et même normalisé.

Théorème de l'automate standard

Pour tout langage reconnaissable L sur un alphabet A (ie $L \in Rec(A^*)$) il existe un automate standard qui le reconnait.

Démonstration :

La démonstration se fait par construction de l'automate standard à partir d'un AFN $T=(A, Q, I, F, \bullet)$ qui reconnaît le langage (algorithme de la standardisation) :

- Soit T' l'automate standard équivalent à T,
  $T'=\{A, Q \cup \{d\}, \{d\}, F', \bullet'\}$
  - $d \notin Q$
  - $F' = F \cup \{d\}$ si $I \cap F \neq \emptyset$
    $F' = F$ sinon
  - $\bullet' = \bullet \cup \{(d,a,q) \mid (i,a,q) \in \bullet, i \in I\}$

Montrons que l'automate standard T' ainsi obtenu reconnaît bien le même langage que T. Autrement dit, montrons que $L(T) = L(Standard(T)) = L(T')$.

Montrons que L(T') est équivalent à L(T).

Si $\varepsilon \in L(T)$ alors $\exists i \in I$ tel que $i \in F$.
Or $I \cap F \neq \emptyset \Leftrightarrow d \in F'$ (par construction de T').
Donc $\varepsilon \in L(T')$.

Soit $w=ax$, $a \in A$, $x \in A^*$. Montrons que $w \in L(T) \Leftrightarrow w \in L(T')$.

Avant cela, définissons deux ensembles particuliers :

1. Soit Act(T) (resp. Act(T')) l'ensemble des actions possibles sur T (resp. T').
   $(a,p,q) \in \mu \Leftrightarrow \{(ax,p) \rightarrow (x,q)\} \in Act(T)$, $x \in A^*$, $a \in A$, $p,q \in Q$
2. Soit S(T) (resp. S(T')) l'ensemble des suites d'actions possibles sur T (resp. T').
   $Act(T) \subseteq S(T)$
   Si $s=\{(xy,q) \text{-}^*\rightarrow (y,q')\} \in S(T)$ et que $\{(axy,i) \rightarrow (xy,q)\} \in Act(T)$, $x,y \in A^*$, $a \in A$ alors $\{(axy,i) \rightarrow (xy,q) \text{-}^*\rightarrow (y,q')\} \in S(T)$.

Or $\mu' = \mu \cup \{(d,a,q) \mid (i,a,q) \in \bullet, i \in I\}$ donc $Act(T) \subset Act(T')$

Donc $S(T) \subset S(T')$

Si $w=ax \in L(T)$ alors $\exists i \in I, q \in Q$ et $f \in F$ tels que :
$s=\{(ax,i) \rightarrow (x,q) \text{-}^*\rightarrow (\varepsilon,f)\}$, et $s \in S(T)$

Donc $s \in S(T')$

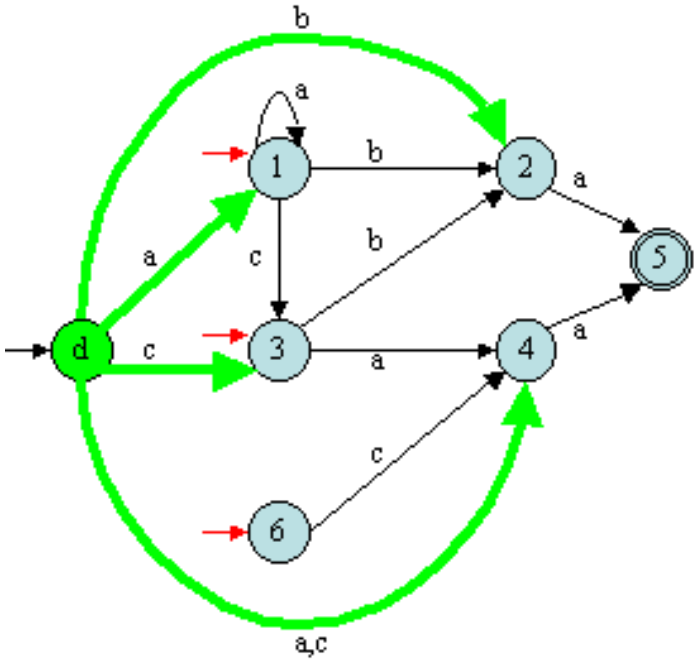
Or $(d,a,q) \in \mu' \Leftrightarrow \exists i \in I, (i,a,q) \in \mu$ (ou μ')

Donc, avec $i \in I$, $\{(ax,i) \rightarrow (x,q)\} \in Act(T') \Leftrightarrow \{(ax,d) \rightarrow (x,q)\} \in Act(T')$

Donc $s=\{(ax,i) \rightarrow (x,q) \text{-}^*\rightarrow (\varepsilon,f)\} \in S(T') \Leftrightarrow \{(ax,d) \rightarrow (x,q) \text{-}^*\rightarrow (\varepsilon,f)\} \in S(T')$

Donc $w \in L(T) \Leftrightarrow w \in L(T')$: CQFD

La figure suivante montre un exemple de standardisation d'un automate fini. L'état et les transitions ajoutés par l'algorithme sont en vert "fluo". Ce qui est en rouge disparaît par l'algorithme.



Remarque : dans cet exemple, après la standardisation, l'état 6 n'est plus un état final. Comme aucune transition n'arrive sur celui-ci, il devient alors inaccessible. Aussi, il est préférable de lancer un émondage après une standardisation pour éliminer ces états inutiles.

Théorème de l'automate normalisé

Pour tout langage reconnaissable L sur un alphabet A (ie L ∈ Rec (A*)) il existe un automate normalisé qui le reconnaît.

Démonstration :

La démonstration se fait par construction de l'automate normalisé à partir d'un AFN T=(A, Q, I, F, •) qui reconnaît le langage : similaire à la démonstration du théorème de l'automate standard.

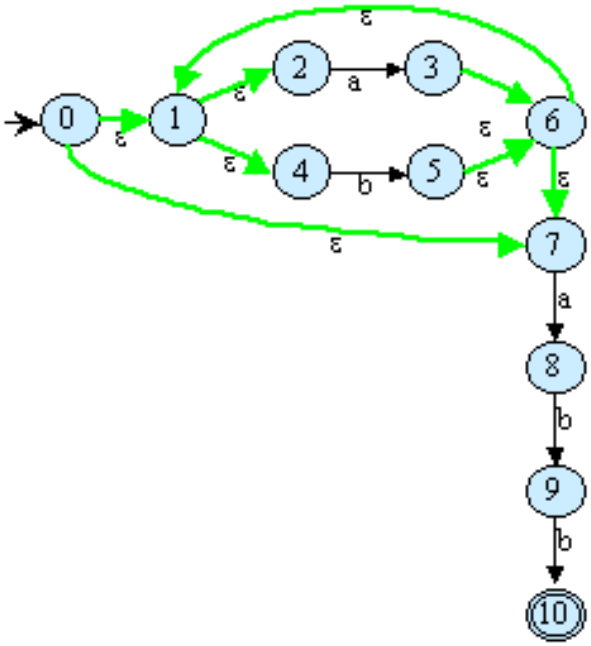
7.4. Cas des automates avec des ε-transitions

Parmi les transitions possibles dans un AFN, il en existe une particulièrement spécifique appelée l'ε-transition (epsilon-transition). De telles transitions sont caractéristiques des automates non déterministes.

ε-transition

Une **ε-transition** est une transition qui n'utilise aucune entrée. C'est une transition <<spontanée>>, c'est-à-dire que l'automate <<décide>> simplement de changer d'état sans lire de symbole.

La figure suivante montre un automate comportant des ε-transitions (en vert). Il reconnait le langage sur l'alphabet {a,b} dont les mots terminent par le suffixe "abb".



Ces transitions très particulières sont principalement introduites lors de la construction automatique d'un automate à partir d'une expression rationnelle (vue plus loin). Cependant, de part leur nature, **elles sont à éviter**. Nous verrons comment les supprimer en présentant la déterminisation d'un AFN. L'AFN sans ε-transition est appelé ε-libre.

AFN ε-libre

Un AFN T=(A, Q, I, F, •) est dit **ε-libre** s'il ne possède pas de ε-transition, c'est-à-dire si : •(ε,a)=0, ∀ q ∈ Q.

Remarque : Même si un AFN T=(A, Q, I, F, •) est ε-libre, il est possible que ε ∈ L(T) quand I ∈ F.

7.5. Automates déterministes (AFD)

Lorsqu'il n'existe au plus qu'une seule action possible au maximum pour une configuration donnée, l'automate fini est alors dit déterministe.

Automate fini déterministe AFD

Un AFN T=(A, Q, I, F, •) est dit **déterministe** (**AFD**) si :

- il est ε-libre
- il est unitaire
- si ∀ q ∈ Q, ∀ a ∈ A : |•(a,q)| ≤ 1.
  Autrement dit, si •(a,q)=q_1 et •(a,q)=q_2 si et seulement si q_1=q_2.

Notation : L'expression $|\bullet(a,q)|$ avec $q \in Q$, $a \in A$ indique le nombre de transitions existant dans l'automate sur la configuration (aw,q), $w \in A^*$.

Les AFDs sont intéressants car, lors d'une analyse, ils ne posent pas de problèmes de choix : pour un état et un symbole, il n'existe au plus qu'une seule transition possible. L'acceptation avec de tels automates est alors plus facile et souvent plus performante. Nous proposerons plus loin un algorithme pour rendre déterministe un automate fini quelconque.

Remarque : Parfois, on trouve la notion d'automate faiblement déterministe. Un AFN $T=(A, Q, I, F, \bullet)$ est dit faiblement déterministe (AFd) si : $\forall q_0 \in Q$, $\forall q_1 \in Q$, $q_0 \neq q_1$, $LG(q_0) \cap LD(q_1) = \varnothing$. On peut montrer que si T est un AFD alors T est un AFd.

7.6. AFD minimal

Un automate fini déterministe de dimension minimale est un cas particulier d'AFD pour lequel il n'existe pas d'AFD équivalent possédant moins d'états. Par contre, il est possible qu'il y en ait avec autant d'états (l'AFD de dimension minimale n'est pas unique).

Par contre, un automate fini minimal est unique. Pour définir un automate minimal, il est possible de s'appuyer sur les notions d'états indistinguables et d'automates réduits.

Il est possible de comparer le rôle de deux états dans un automate par rapport aux mots reconnus : la notion d'états distinguables ou indistinguables.

Etats distinguables
Etats k-indistinguables

Soit un AFD $T=(A, Q, I, F, \bullet)$, on dit que la chaîne d'entrée $w \in A^*$ distingue l'état e de l'état f ($e \neq f$) dans T si :

- partant de l'état e et le faisant fonctionner avec w, on termine dans un état d'acceptation ;
- partant de l'état f et le faisant fonctionner avec w, on termine dans un état de non-acceptation (ou vice versa).

Les états e et f sont alors dits **distinguables**. Autrement dit, w distingue e et f si $\exists q_1, q_2 \in Q$, $(w, e) —*/T\rightarrow (\varepsilon, q_1)$ et $(w, f) —*/T\rightarrow (\varepsilon, q_2)$ avec exactement 1 des états q_1 ou $q_2 \in F$.

Deux états e et f d'un AFD T sont **k-indistinguables**, noté $e \equiv_k f$ si : $\forall w \in A^*$, $|w| \leq k$ w ne distingue pas e et f.

Deux états e et f d'un AFD T sont **indistinguables**, si $\forall k \geq 0$, $e \equiv_k f$.

Remarque : e et f sont distinguables s'ils vérifient la condition suivante : ($w \in LD(e)$ et $w \notin LD(f)$) ou ($w \in LD(f)$ et $w \notin LD(e)$)

Un automate réduit est un automate qui ne possède pas de couples d'états indistinguables.

Automate réduit

Un AFD T=(A, Q, I, F, •) est dit **réduit** si tous les états de Q sont accessibles et si tous les d'états de Q pris deux à deux sont distinguables.

Un automate minimal est alors un automate réduit dont on ne peut trouver un automate équivalent avec un nombre d'états strictement inférieur au sien (de dimension minimale). Cet automate est unique à la numérotation des états près.

Automate minimal

Un AFD T=(A, Q, I, F, •) est dit **minimal** s'il est réduit et si : $Min(T) \equiv \forall T' \in AFD, L(T) = L(T') : |T| \leq |T'|$.

Un AFD T=(A, Q, I, F, •) est dit **minimal complet** si : $Min_C(T) \equiv \forall T' \in AFD, Complet(T'), L(T) = L(T') : |T| \leq |T'|$.

Nous verrons plus loin un algorithme pour minimiser un automate fini.

7.7. Quelques langages et leurs automates

Beaucoup d'opérations sur les langages reconnaissables ont un algorithme associé sur les automates qui les reconnaissent. Nous allons en étudier trois ici, d'autres viendrons dans une prochaine section.

Théorème des facteurs

Pour tout langage reconnaissable L sur un alphabet A (ie $L \in Rec(A^*)$), Pref(L) (le langage des préfixes des mots L), Suff(L) (le langage des suffixes des mots de L) et Fact(L) (le langage des facteurs dans L) sont aussi reconnaissables.

Démonstration :

La démonstration se fait par construction des automates à partir d'un AFN T=(A, Q, I, F, •) qui reconnait le langage L :

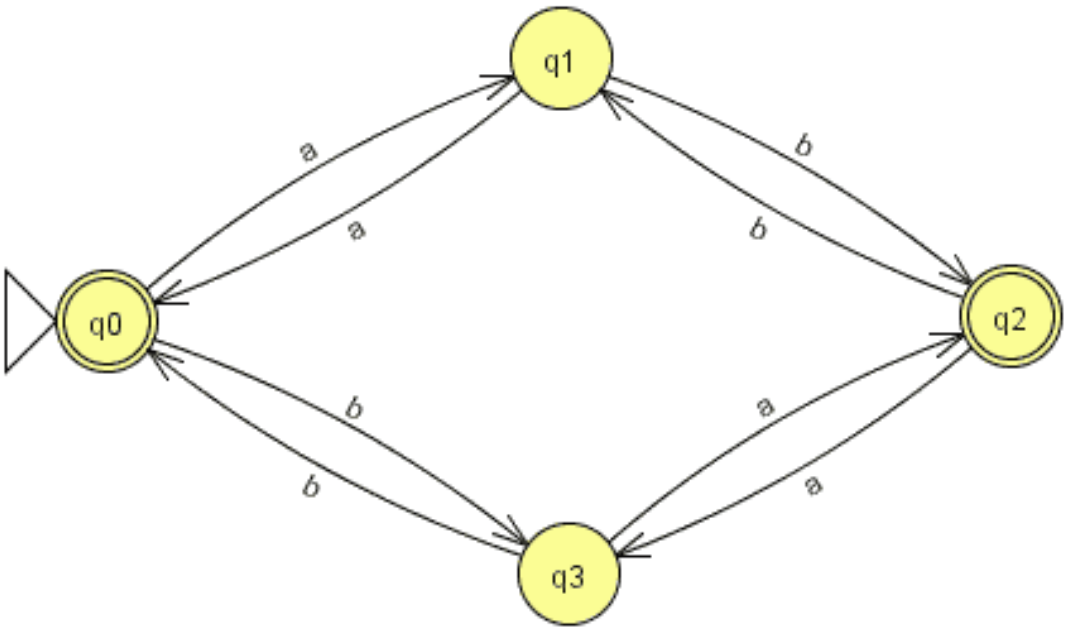
- T_{Pref} = {A, Q, I, Q, •} si $utile_F(T)$ (ie, tous les états sont co-accessibles)
- T_{Suff} = {A, Q, Q, F, •} si $utile_I(T)$ (ie, tous les états sont accessibles)

- T_{Fact} = {A, Q, Q, Q, •} si émondé(T) (ie, tous les états sont accessibles et co-accessibles)

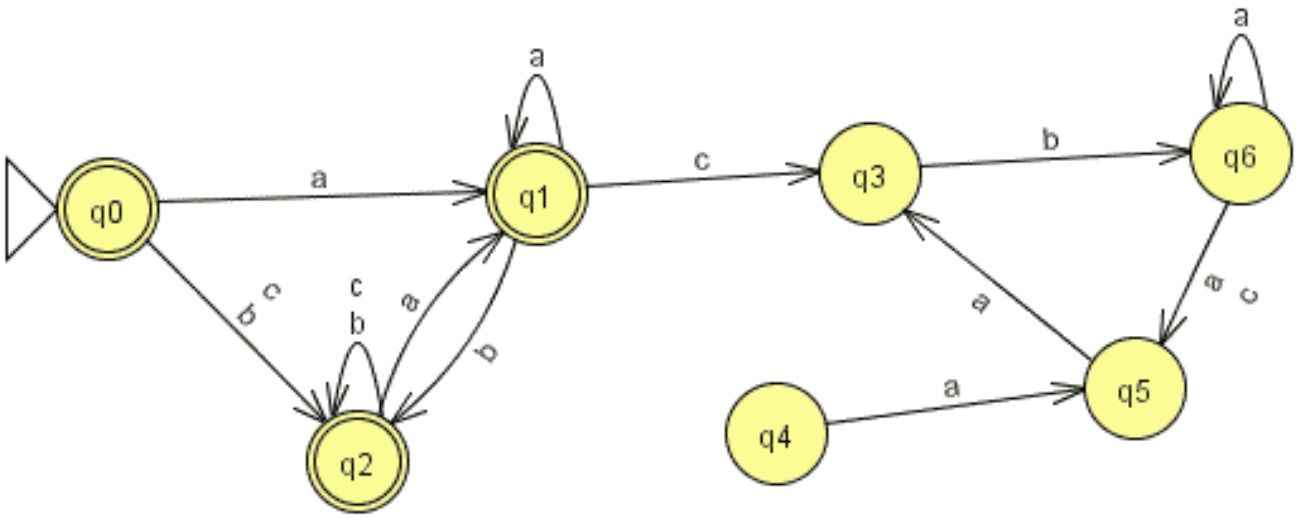
Exercices et tests :

Exercice 7.1. Montrer pourquoi un automate fini normalisé n'est jamais complet.

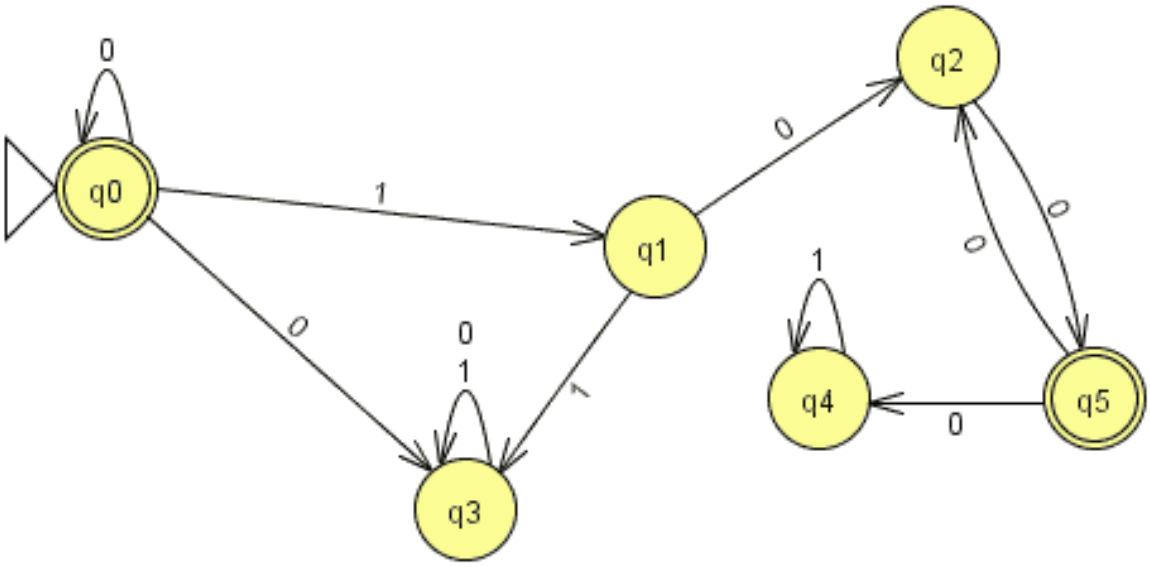
Exercice 7.2. Donner les propriétés (la minimalité n'est pas demandée) des automates suivants (images issues de JFLAG ; avec ce logiciel les ε-transitions sont notée λ) :



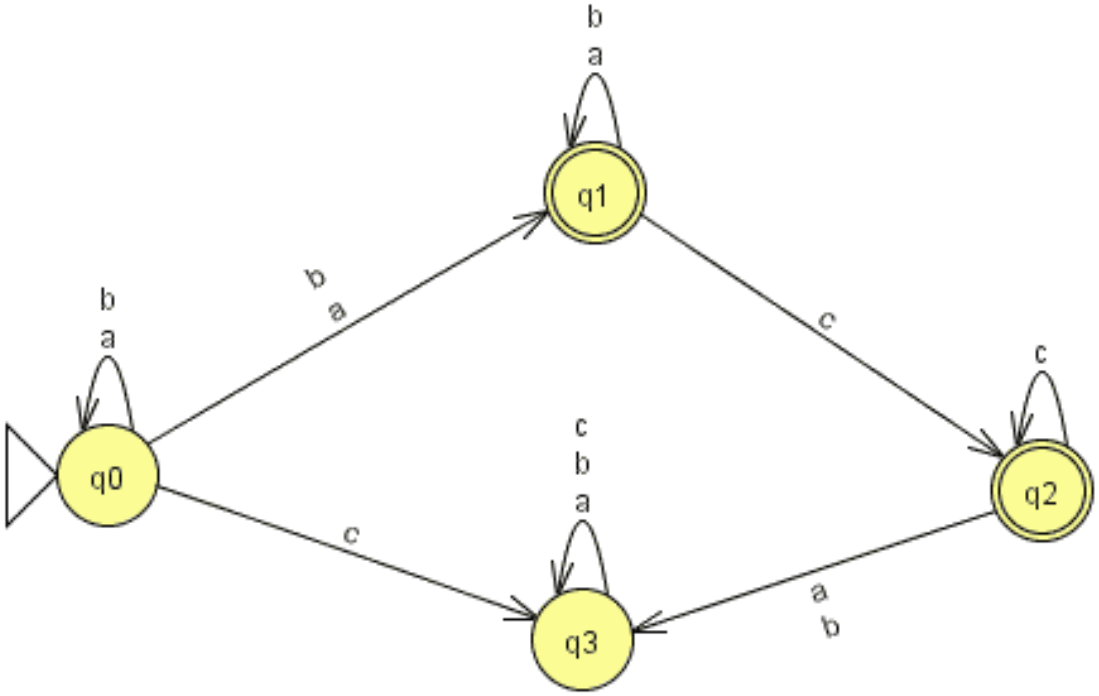
1. sur l'alphabet {a,b}

2. 

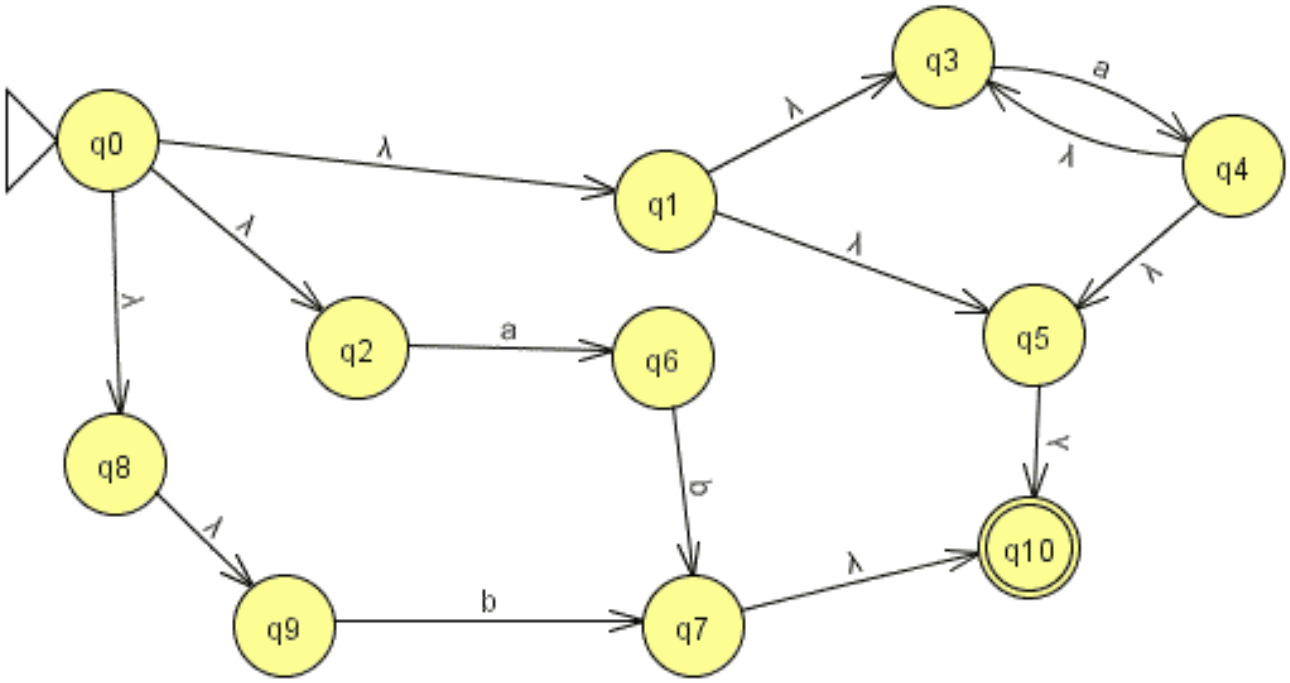
sur l'alphabet {a,b,c,d}

3. 

sur l'alphabet {0,1,2,3,4,5,6,7,8,9}

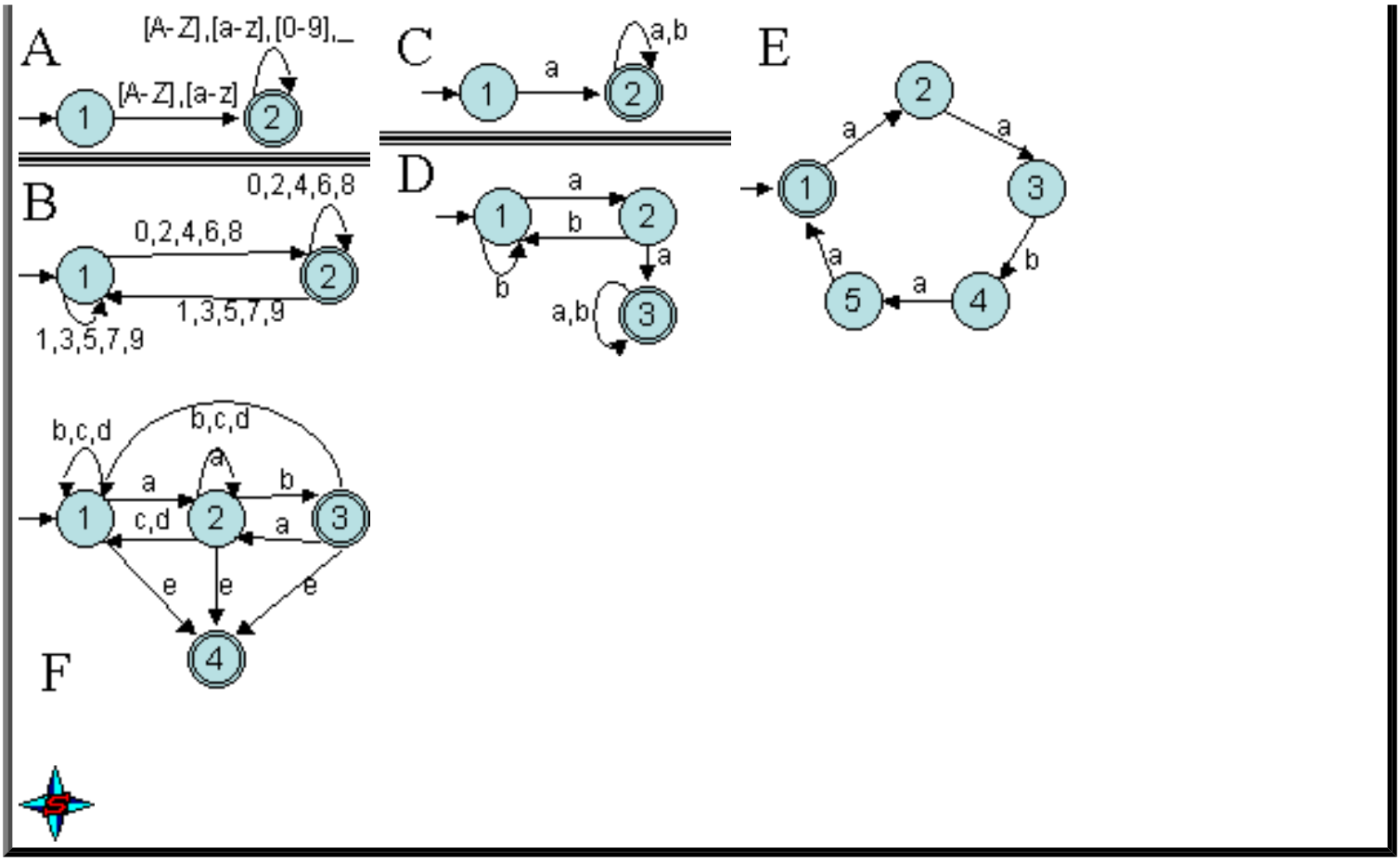


4. sur l'alphabet {a,b,c,d}



5. sur l'alphabet {a,b}

Exercice 7.3. Les automates suivants sont-ils déterministes :



8. Déterminisation d'un AFN

8.1. Introduction

Après avoir introduit les automates finis non déterministes (AFN) et déterministes (AFD), nous allons tout d'abord présenter un algorithme permettant de passer d'un AFN à un AFD. Il en exite plusieurs, plus ou moins performants. Nous ne présenterons ici qu'un seul algorithme qui nous semble suffisamment simple pour avoir une idée de la méthode abordée.

8.2. AFN et AFN ε-libre

A tout AFN comportant des ε-transitions, il est possible de faire correspondre un AFN équivalent, c'est-à-dire reconnaissant le même langage, sans ε-transition.

Théorème de l'ε-fermeture	A tout automate T=(A, Q, I, F, •) comportant des ε-transitions existe un automate T'=(A, Q, I, F', ε-fermeture•) équivalent, sans ε-transition, avec F' = F ∪ {I} s'il existe une chaîne de ε-transitions entre I et un état q ∈ F et F' = F sinon.

La fonction ε-fermeture•(a,q) détermine tous les états dans lesquels peut passer l'automate

dans l'état q après (1) zéro ou plus ε-transitions, (2) une transition sur a puis (3) zéro ou plus ε-transitions. L'algorithme permettant de déterminer l'ε-fermeture pour un ensemble d'états sera présenté plus loin.

8.3. Passage d'un AFN vers un AFD

Il n'est pas difficile de comprendre que la complexité d'un algorithme de reconnaissance basé directement sur un AFN est beaucoup plus importante (beaucoup d'états, états inutiles et surtout non-déterminisme) que celle basée sur un AFD.

Or, il est possible de montrer que, pour tout AFN, il existe un AFD équivalent, c'est-à-dire reconnaissant le même langage. Il est donc intéressant de chercher à transformer un AFN en AFD. Pour cela, nous allons présenter une méthode simple appelée la méthode par construction des sous-ensembles.

L'algorithme de transformation d'un AFN en AFD n'est possible que parce qu'on peut montrer qu'à un AFN donné il existe un AFD équivalent.

Théorème d'équivalence entre AFN et AFD

Soit T=(A, Q, I, F, •) un AFN qui reconnaît le langage L=L(T) alors il existe un AFD T'=(A, Q', I', F', •') tel que L'=L(T')=L.

Dém. (constructive) : on construit T' de la façon suivante :

- $Q' \subseteq P(Q)$, c'est-à-dire que les états de T' sont des ensembles d'états de T ;
- $I' = \{I\}$;
- $F' = \{Q' \subseteq Q : Q' \cap F \neq \emptyset\}$;
- $\forall S \subset Q$ et $\forall a \in A$, $\bullet'(a,S)=S'$ où $S'=\{p \mid \exists q \in S : p \in \bullet(a,q)\}$.

On peut montrer que $(w,S) \text{—}i/T'\rightarrow (\varepsilon, S')$ si et seulement si $S'=\{p \mid \exists q \in S : (w,q) \text{—}i/T\rightarrow (\varepsilon, p)\}$. Cas particulier : $(w, \{q_0\}) \text{—}i/T'\rightarrow (\varepsilon, S')$ pour $S' \in F'$ si et seulement si $(w, q_0) \text{—}i/T\rightarrow (\varepsilon, p)$ pour $p \in F$. En effet :

- Vrai pour i=1 : $(a,S) \text{—}1/T'\rightarrow (\varepsilon, S')$ donc $\forall p \in S'$, $\exists q \in S$, $p \in \bullet(a,q)$ d'où $\forall p \in S$ $(a,q) \text{—}1/T\rightarrow (\varepsilon, p)$ donc $S' = \{p \mid \exists q \in S : (w,q) \text{—}1/T\rightarrow (\varepsilon, p)\}$;
- Supposons vrai pour i=n, montrons pour i=n+1 :
  $(aw,S) \text{—}n+1/T'\rightarrow (\varepsilon, S') \Leftrightarrow (aw,S) \text{—}1/T'\rightarrow_1 (w, S'') \text{—}n/T'\rightarrow_2 (\varepsilon, S')$.
  $\text{—}1/T'\rightarrow_1 \Leftrightarrow \bullet(a, S) = S''$ et $S'' = \{q'' \mid \exists q \in S : q'' \in \bullet(a,q)\}$.
  $\text{—}n/T'\rightarrow_2 \Leftrightarrow S' = \{p \mid \exists q'' \in S'' : (w,q'') \text{—}n/T\rightarrow (\varepsilon, p)\}$.

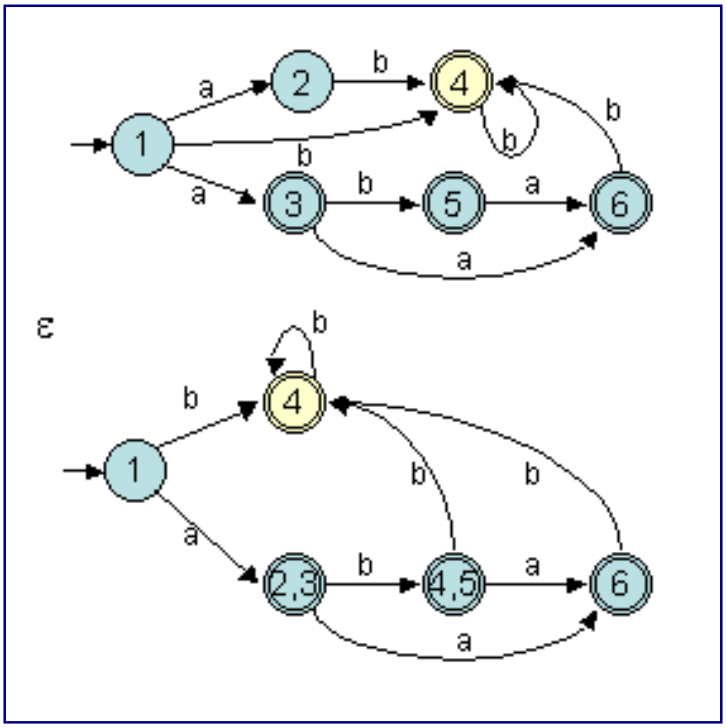
En fusionnant S' et S'' S'={p | ∃ q ∈ S : q'' ∈ •(a,q), (w,q'') —n/T→ (ε, p)}.
Or {p | ∃ q ∈ S : q'' ∈ •(a,q)} ⇒ (aw,q) —1/T→ (ε, p)
donc S' = {p | ∃ q ∈ S : (aw,q) —n+1/T→ (ε, p)}.

8.4. Construction d'un AFD à partir d'un AFN avec ε-transitions.

Nous présentons maintenant un algorithme qui construit à partir d'un AFN un AFD qui reconnaît le même langage. Cet algorithme, appelé souvent **par construction des sous-ensembles**, est utile pour faire simuler un AFN par un programme.

L'idée générale sur laquelle se base la transformation d'un AFN en AFD est que chaque état de l'AFD correspond à un ensemble d'états de l'AFN. Un état courant de l'AFD correspond à l'ensemble des états possibles de l'AFN qui sont atteints avec la même chaîne d'entrée. L'AFD "simule" le parcours "en parallèle" des états pour un mot donné.

L'exemple suivant illustre la reconnaissance du mot "abab" sur deux automates équivalents l'un (celui du haut) non déterministe et l'autre (celui du bas) déterministe et obtenu par la méthode de construction des sous-ensembles. Le nom des états indiquent les états dans lesquels on arrive avec le parcours en parallèle du premier automate.



La construction d'un AFD à partir d'un AFN se base sur les définitions de l'ε-fermeture, permettant de supprimer les ε-transitions et d'une fonction M permettant de déterminer les états accessibles depuis un ensemble d'états par les transitions selon un symbole donné de l'alphabet.

ε-fermeture d'un état

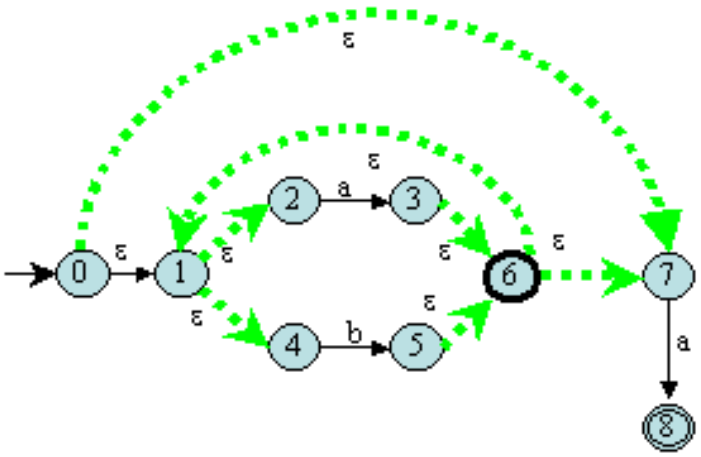
L'ε-fermeture d'un état e ∈ Q d'un AFN T=(A, Q, I, F, •), notée ε-fermeture(e), est l'ensemble des états de T accessibles depuis l'état e par 0, 1... n ε-transitions seulement.

ε-fermeture d'un ensemble d'états

L'ε-fermeture d'un ensemble d'états E ⊂ Q d'un AFN T=(A, Q, I, F, •), notée ε-fermeture(E), est l'ensemble des états de T accessibles depuis chacun des états e ∈ E par des ε-transitions seulement.

Donc :
ε-fermeture(T,E) = E ∪ {∪ ε-fermeture(T,{q_j}) tels que ∃ q_i ∈ E, •(ε,q_i)=q_j}

Par exemple, considérons l'automate T suivant :



Calculons ε-fermeture(T,{6}).

•(ε, 6) = 1 et •(ε, 6) = 7 donc :
ε-fermeture(T,{6}) = {6} ∪ ε-fermeture(T,{1}) ∪ ε-fermeture(T,{7}).

•(ε, 1) = 2 et •(ε, 1) = 4 donc :
ε-fermeture(T,{1}) = {1} ∪ ε-fermeture(T,{2}) ∪ ε-fermeture(T,{4})

•(ε, 2) = ∅ donc ε-fermeture(T,{2}) = {2}

•(ε, 4) = ∅ donc ε-fermeture(T,{4}) = {4}

Donc ε-fermeture(T,{1}) = {1, 2, 4}

•(ε, 7) = ∅ donc ε-fermeture(T,{7}) = {7}

Donc ε-fermeture(T,{6}) = {1, 2, 4, 6, 7}

Le calcul de l'ε-fermeture d'un ensemble E est un processus typique de recherche dans un graphe d'un ensemble donné de noeuds. Dans ce cas, les états de E forment un ensemble donné de noeuds et le graphe est constitué uniquement des ε-transitions de l'AFN. Pour calculer l'ε-fermeture, on utilise une pile pour conserver les états dont les transitions n'ont pas encore été examinées.

M

La fonction M(a,E) est l'ensemble des états E' de T vers lesquels il existe une transition sur le symbole d'entrée a à partir des états e ∈ E.

Donc : M(a,E) = $\{q_i\} \subseteq Q$ tels que ∃ •(a,p)=q_i avec a ∈ A et p ∈ E.

Si l'on prend l'automate T précédent, M(a,{2,7}) = {3,8}.

L'algorithme construit la fonction de transition •' de l'AFD T'=(A, Q', I', F', •'). Chaque état de l'AFD est un ensemble d'états de l'AFN T=(A, Q, I, F, •) et on construit •' de telle manière que **l'AFD simulera "en parallèle" tous les déplacements possibles que l'AFN peut effectuer sur une chaîne d'entrée donnée**. On utilise les opérations définies ci-dessus pour garder trace des ensembles d'états de l'AFN.

On construit Q', I' et •' de la manière suivante. Chaque état de Q' correspond à un ensemble d'états de T dans lesquels l'automate pourrait se trouver après avoir lu une suite de symboles en entrée en incluant les ε-transitions possibles avant ou après la lecture des symboles. L'état de départ I' est l'ε-fermeture de I. Un état de T' est un état d'acceptation si c'est un ensemble d'états de T qui contient au moins un état d'acceptation.

8.5. Algorithme de construction d'un AFD à partir d'un AFN

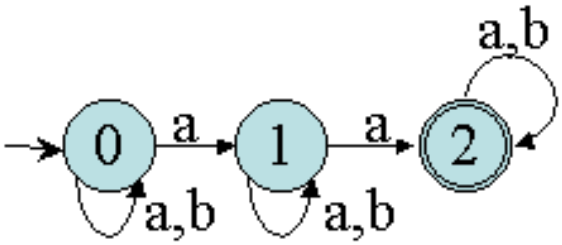
Compte-tenu de ce qui a été présenté à la section précédente, l'algorithme de construction

d'un AFD T' = {A, Q', I', F', •'} à partir d'un AFN quelconque T = {A, Q, I, F, •} est le suivant :

- I' = ε-fermeture(T,I) ; Ajouter I' dans Q' ;
- Pour chaque "état" E (ensemble d'états de T) non-marqué dans Q' :
  - Le marquer
  - Déterminer pour chaque symbole a de A
    - E' = M(a,E)
    - E'' = ε-fermeture(T,E')
    - Si E'' n'existe pas dans Q' alors :
      - L'ajouter dans Q'
      - S'il contient un état final de T alors l'ajouter dans F'
    - Créer une transition •'(a,E)

8.6. Exemple 1

Soit l'automate suivant :



La construction de l'AFD équivalent commence par la détermination de l'état initial (c'est un AFD, donc il n'y a qu'un seul état initial). Celui-ci est construit à partir de l'ε-fermeture de tous les états initiaux. Donc ici, I'={0}. A partir de là, on construit la table des transitions.

Phase 1 :

•	a	b
A = {0} = I'		

Puis, on regarde vers quels états l'automate se trouve pour chacun des symboles de l'alphabet

à partir de I'. Normalement, il faut faire l'ε-fermeture sur les états obtenus mais dans cet automate il n'y a aucune ε-transition. On les ajoute à la table s'ils n'y sont pas déjà.

Phase 2 :

•	a	b
A = {0} = I'	{0,1}	{0}
B = {0,1}		

Puis on fait la même chose pour chacun des états pas encore traités.

Phase 3 :

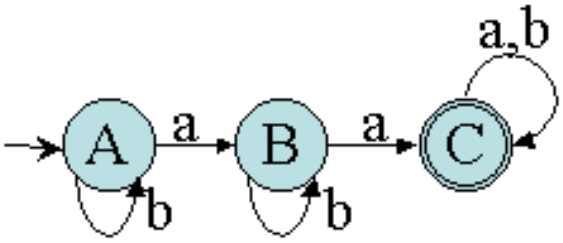
•	a	b
A = {0} = I'	{0,1}	{0}
B = {0,1}	{0,1,2}	{0,1}
C = {0,1,2}		

C contient l'état 2 qui est final dans l'automate de départ. Donc C est un état final dans l'AFD obtenu.

Phase 4 :

•	a	b
A = {0} = I'	{0,1}	{0}
B = {0,1}	{0,1,2}	{0,1}
C = {0,1,2} ∈ F	{0,1,2}	{0,1,2}

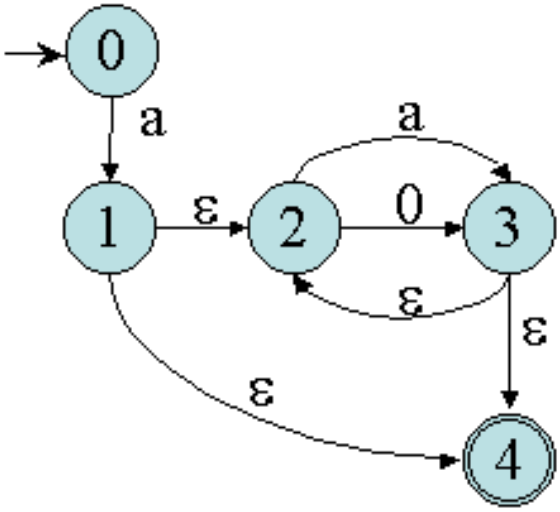
Finalement, le tableau est rempli, l'automate est terminé. On obtient l'automate suivant :



Remarque : Cette méthode ne comporte pas de difficultés particulières. La seul difficulté est que le nombre d'ensembles d'états peut être assez grand ainsi que la dimension de ces ensembles. Par conséquent, il n'est pas rare de faire des erreurs d'étourderie (la où la machine ne risque pas d'en faire !).

8.6. Exemple 2

Prenons comme nouvel exemple un automate comportant des ε-transitions :



L'algorithme passe par les phases suivantes :

Phase 1 :

•	a	0
A = {0} = I '		

Phase 2 :

•	a	b
A = {0} = I '	{**1**,2,4}	∅
{1,2,4} = B ∈ F		
∅ = C		

L'état en vert dans l'ensemble {**1**,2,4} est l'état dans lequel on arrive directement depuis l'état 0. Les autres sont ceux obtenus après ε-fermeture.

Phase 3 :

•	a	b
A = {0} = I '	{**1**,2,4}	∅
{1,2,4} = B ∈ F	{3,2,4}	{3,2,4}
∅ = C		
{2,3,4} = D ∈ F		

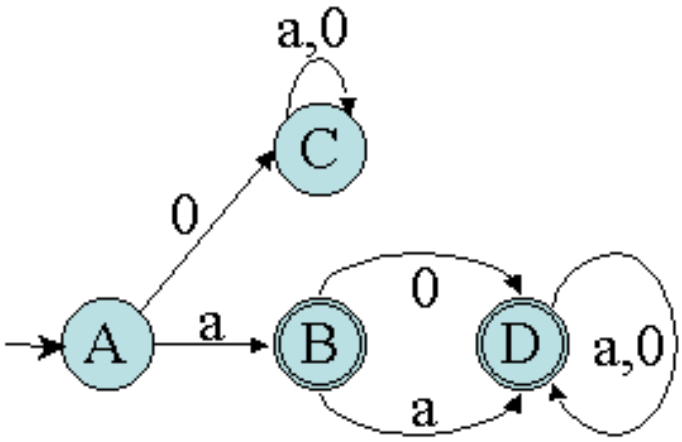
Phase 4 :

•	a	b
A = {0} = I '	{**1**,2,4}	∅
{1,2,4} = B ∈ F	{3,2,4}	{3,2,4}
∅ = C	∅	∅
{2,3,4} = D ∈ F		

Phase 5 :

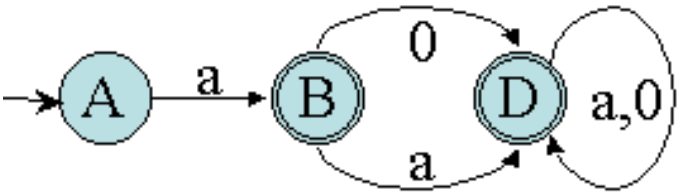
•	a	b
A = {0} = I '	{**1**,2,4}	∅
{1,2,4} = B ∈ F	{3,2,4}	{3,2,4}
∅ = C	∅	∅
{2,3,4} = D ∈ F	{3,2,4}	{3,2,4}

Le tableau est terminer donc l'automate aussi. L'automate obtenu est le suivant :



A noter l'état C qui est un état puits. Il est possible de ne pas l'ajouter et de ne pas remplir les cases qui correspondent. Ceci donne alors :

•	a	b
A = {0} = I '	{**1**,2,4}	
{1,2,4} = B ∈ F	{3,2,4}	{3,2,4}
{2,3,4} = C ∈ F	{3,2,4}	{3,2,4}



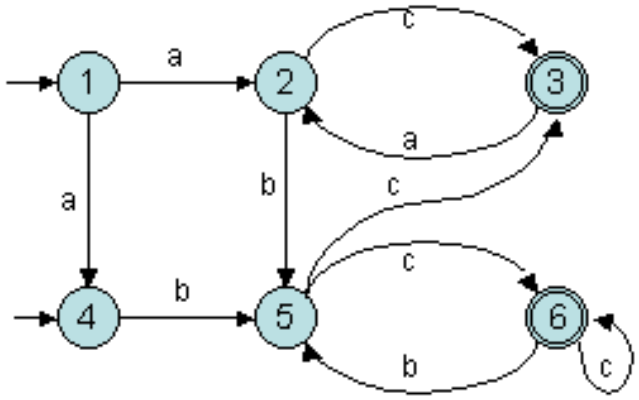
8.7. Simulation directe d'un AFN

Il est possible d'effectuer une reconnaissance de chaîne donnée directement à partir de l'AFN sans passer par une déterminisation.

Cet algorithme construit dynamiquement les sous-ensembles. Il calcule une transition depuis l'ensemble courant d'états vers le prochain ensemble d'états en deux étapes. D'abord, il détermine l'ensemble de tous les états accessibles depuis l'ensemble courant par une transition sur le caractère courant. Ensuite, il calcule l'ε-fermeture, c'est-à-dire tous les états qui peuvent être atteints par zéro ou plusieurs ε-transitions.

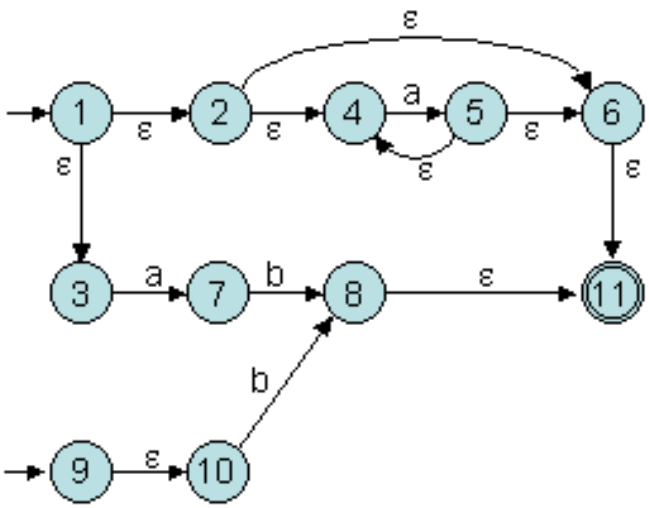
Exercices et tests :

Exercice 8.1. Soit l'automate suivant sur l'alphabet {a,b,c} :



Donner l'automate déterministe équivalent par la méthode de construction des sous-ensembles (donner le détail de la méthode).

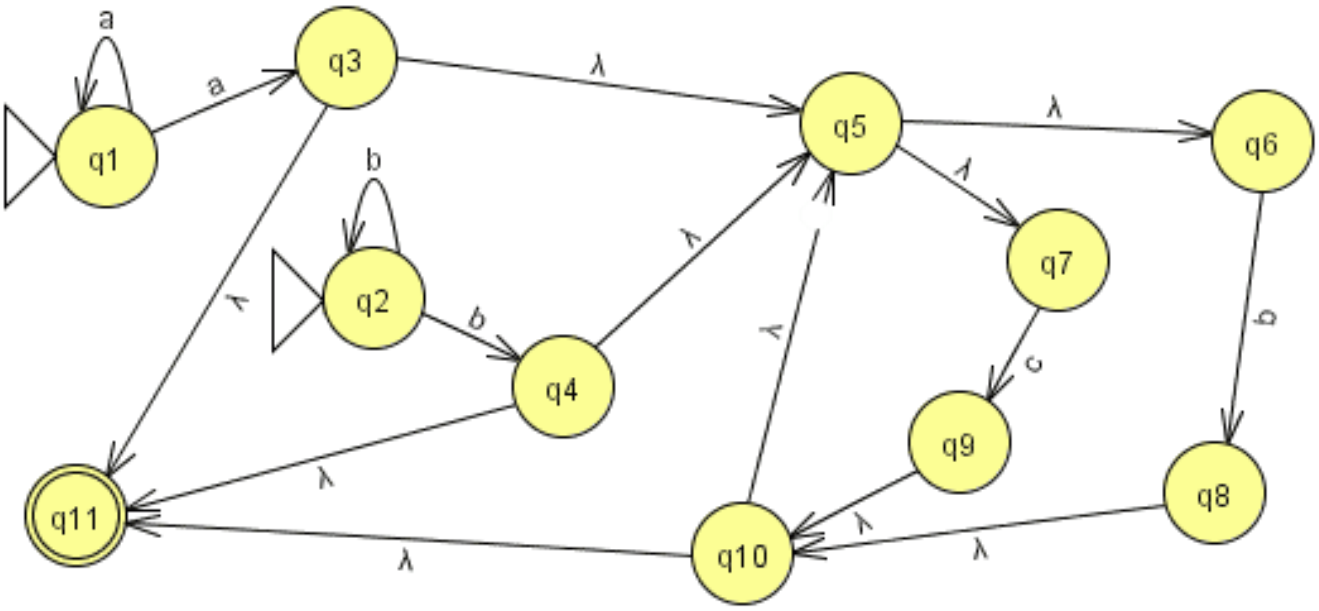
Exercice 8.2. Soit l'automate suivant sur l'alphabet {a,b} :



Donner l'automate déterministe équivalent par la méthode de construction des sous-

ensembles (donner le détail de la méthode).

Exercice 8.3. Soit l'automate suivant sur l'alphabet {a,b,c} (images issues de JFLAG ; avec ce logiciel les ε-transitions sont notée λ) :



Donner l'automate déterministe équivalent par la méthode de construction des sous-ensembles (donner le détail de la méthode).

9. Minimisation d'un AFD

9.1. Introduction

Après avoir introduit les automates finis non déterministes (**AFN**) et déterministes (**AFD**) et présenté un algorithme permettant de passer d'un AFN à un AFD, nous allons étudier une méthode permettant de simplifier, **minimiser**, un AFD.

9.2. Minimisation d'un AFN : Algorithme de Moore

Certains AFDs possèdent un nombre d'états et de transitions important. États et transitions ne sont pas toujours indispensables. Typiquement, cette situation a lieu lorsque l'automate est construit à l'aide d'algorithmes automatiques comme dans le cas de la construction

automatique d'un AFN et de la transformation d'un AFN en AFD. Afin d'améliorer la reconnaissance des chaînes et d'optimiser la place mémoire utilisée, il est intéressant d'essayer de minimiser un AFD. Une première approche de cette minimisation peut consister à étudier les caractéristiques des états et de supprimer ceux qui sont inutiles. Cependant, il est possible d'aller plus loin en garantissant que l'AFD obtenu est un AFD minimal. Ceci est l'objectif de la méthode que l'on présente ici.

Il existe de nombreuses stratégies pour minimiser un AFD [Wat93b]. Dans cette référence, on trouvera aussi la formalisation et la caractérisation des classes d'équivalence. L'algorithme de Moore présenté ici est un algorithme, issu de [ASU91], basé sur les classes d'équivalence d'états.

L'algorithme de minimisation du nombre d'états d'un AFD fonctionne en déterminant tous les groupes d'états qui peuvent être distingués par une chaîne d'entrée. Chaque groupe d'états indistinguables est alors fusionné en un état unique. L'algorithme travaille en mémorisant et en raffinant une partition de l'ensemble des états. Chaque groupe d'états à l'intérieur de la partition correspond aux états qui n'ont pas encore été distingués les uns des autres. Toute paire d'états extraits de différents groupes a été prouvée "distinguable" par une chaîne.

Initialement, la partition consiste à former deux groupes : les états d'acceptation et les autres. L'étape fondamentale prend un groupe d'états et un symbole puis étudie les transitions de ces états sur ce symbole. Si ces transitions conduisent à des états qui tombent dans au moins deux groupes différents de la partition courante, alors on doit diviser ce groupe. La division est effectuée avec pour objectif que les transitions depuis chaque sous-groupe soient confinées à un seul groupe de la partition courante. Ce processus de division est répété jusqu'à ce qu'aucun groupe n'ait plus besoin d'être divisé.

Les états du nouvel automate sont donnés par un représentant de chaque groupe. Les états finaux sont les représentants des groupes possédant un état final de l'AFD de départ. L'état de départ est le représentant du groupe possédant l'état de départ de l'AFD initial. Les transitions sont construites avec les transitions de chaque état représentant de groupe vers un état représentant d'un autre groupe si cet état engendre une transition vers un élément du groupe.

Sur les états créés, il faut supprimer les états stériles et les états non-accessibles depuis l'état initial.

Théorème de l'AFD minimal réduit

L'automate T' obtenu par cet algorithme est minimal.

Démonstration

Supposons que $\exists$ T''=(A, Q'', I'', F'', •''), |T''| < |T'|, L(T'')=L(T').

$\forall$ q $\in$ Q', Accessible(q).

$\exists$ w,x $\in$ A*, (w,I'') —*/T''→ (ε, q) et (x,I'') —*/T''→ (ε, q).

Mais w et x conduisent T' à des états différents.

Il s'en suit que ces deux chaînes conduisent T à des états différents p et q qui sont distingables, c'est-à-dire $\exists$ y $\in$ A*, exactement une des chaînes xy et wy $\in$ L(T).

Mais, ces chaînes doivent conduire T'' au même état, c'est-à-dire l'état s : (y, q) —*/T''→ (ε, s).

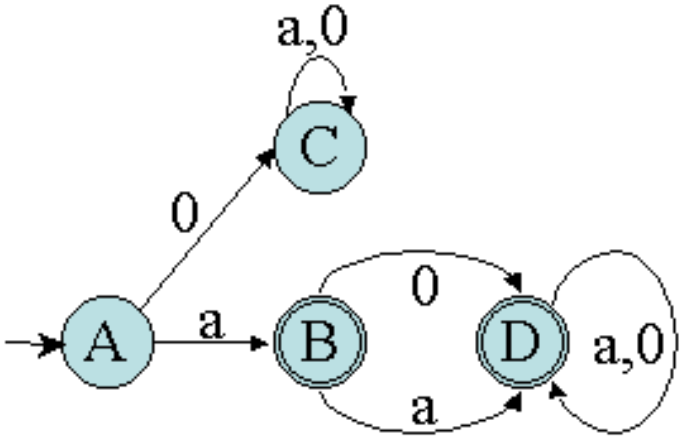
Donc (wy,I'') —*/T''→ (y, q) —*/T''→ (ε, s) et (xy,I'') —*/T''→ (y, q) —*/T''→ (ε, s).

Il n'est donc pas possible qu'une seule chaîne wy et xy $\in$ L(T'') donc à L(T).

Donc T'' n'existe pas.

Remarque : cette méthode de minimisation (comme pour la méthode de déterminisation) est applicable par la manipulation d'une table de transitions. La première ligne donne pour chacun des états son appartenance à un des deux ensembles de départ. Ensuite, pour chaque état et chaque symbole on détermine vers quel ensemble nous mène la transition correspondante. Une fois tous les couples étudiés, on divise les ensembles de départ en ensembles dont les états se comportent de la même manière. On recommence jusqu'à ce qu'il n'y ait plus de division d'ensemble. L'exemple suivant illustre cette manière de faire.

9.3. Exemple 1



Si on applique cet algorithme à l'automate déterministe de l'exemple 2 de la section précédente (automate ci-dessus), la phase d'initialisation permet d'obtenir :

	A	B	C	D
ε (initialisation)	I	I I	I	I I
a				
0				
Bilan 1				

En effet, A et C ne sont pas des état finaux. Ils sont donc dans le même ensemble (noté I). De même, B et C sont finaux. Ils sont donc dans le même ensemble (noté I I).

Ensuite, pour chaque couple (état, symbole), on regarde vers quel ensemble nous mène la transition de l'automate (si elle existe, sinon on ne met rien).

	A	B	C	D
ε (initialisation)	I	I I	I	I I
a	I I	I I	I	I I
0	I	I I	I	I I
Bilan 1				

Puis on effectue la séparation des ensembles. Deux états sont dans un même ensemble s'ils étaient déjà dans le même ensemble et si les transitions mènent dans les mêmes ensembles. Dans cet exemple, B et D se comportent exactement de la même manière, donc ils restent ensemble. Par contre, A et C qui faisaient partie du même ensemble se comportent différemment sur le symbole "a". Par conséquent, l'ensemble noté I se divise en deux. Nous obtenons donc :

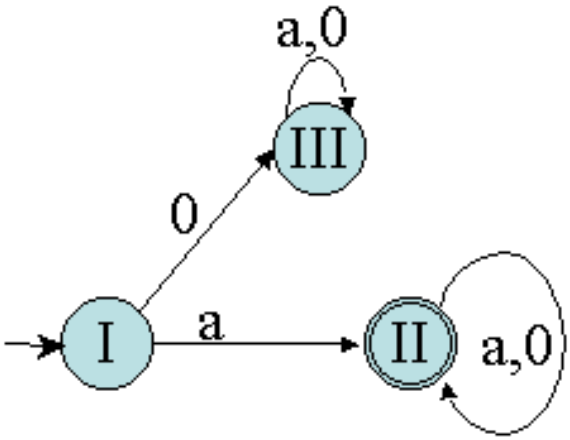
	A	B	C	D
ε (initialisation)	I	I I	I	I I
a	I I	I I	I	I I
0	I	I I	I	I I
Bilan 1	I	I I	I I I	I I

La situation courante (la ligne Bilan 1) est différente que la situation de départ. Donc il faut recommencer !

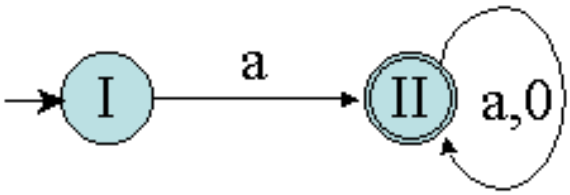
	A	B	C	D
ε (initialisation)	I	I I	I	I I
a	I I	I I	I	I I
0	I	I I	I	I I
Bilan 1	I	I I	I I I	I I
a	I I	I I	I I I	I I
0	I I I	I I	I I I	I I
Bilan 2	I	I I	I I I	I I

La ligne "Bilan 2" est identique à la ligne "Bilan 1". Par conséquent, dans chacun des ensembles restants, les états ne sont pas distinguables (ils se comportent exactement de la même manière et sont donc "redondants").

Nous pouvons donc construire le nouvel automate en associant un état à chaque ensemble. Les transitions sont indiquées par le tableau. L'état initial est celui contenant l'état initial de l'automate de départ : ici I contient l'état A, état initial. Les états finaux sont les états dont l'ensemble correspondant contient au moins un état final : ici, I I contient B et D qui sont finaux.

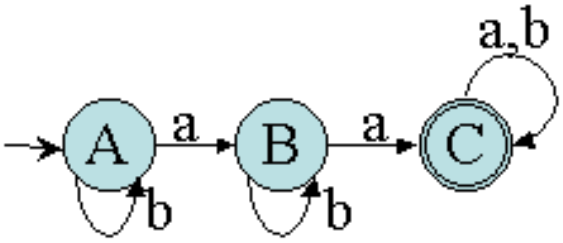


Cependant, l'algorithme n'est pas encore terminé ! Pour l'instant, nous avons "fusionné" les états indistinguables. Il faut maintenant supprimer tous les états inutiles restant. Dans notre exemple, l'état III est un état stérile, donc inutile. Il faut donc le supprimer. D'où l'automate déterministe minimal de la figure suivante.



9.4. Exemple 2

Considérons maintenant l'exemple A de la section précédente :



Appliquons la même méthode de minimisation :

	A	B	C
ε (initialisation)	I	I	I I
a	I	I I	I I
b	I	I	I I
Bilan 1	I	I I	I I I
a			
b			
Bilan 2			

A ce stade, le bilan 1 est différent de l'état initial. Cependant, il y a autant d'ensembles que d'états (ou seulement un état par ensemble). Il ne peut donc pas y avoir d'autre division d'ensemble. Par conséquent, nous pouvons arrêter là le traitement. Notons que nous obtenons exactement le même automate que celui de départ (à la numérotation près). Par conséquent, nous avions un automate déjà minimal !

9.5. Pré-traitement

Pour des automates complexes, cet algorithme peut s'avérer coûteux. Aussi, il est préférable d'effectuer un pré-traitement permettant de réduire simplement le nombre d'états de

l'automate.

Une première idée est d'appliquer l'algorithme d'émondage pour supprimer les états de l'AFD qui sont inutiles.

Une seconde idée consiste à supprimer les états qui ont les mêmes transitions pour n'en garder qu'une seule. Par exemple, prenons la table de transition suivante :

•	a	b
v		x
x	z	t
u	y	
y	z	t
t	x	y
z		

Les transitions partant de x et de y sont les mêmes. Par conséquent, x et y ne sont pas distinguables. Donc, il est possible d'en supprimer un : par exemple y. Toutes les transitions menant à y sont alors dirigées vers x. Nous obtenons alors la table suivante :

•	a	b
v		x
x	z	t
u	x	
t	x	x
z		

9.6. Est-il minimal ?

Pour vérifier si un automate déterministe est minimal, il suffit d'appliquer l'algorithme de minimisation. Ensuite, si les deux automates sont identiques (à la numérotation des états près) alors l'automate d'origine est minimal.

Il existe aussi une heuristique permettant de vérifier la "minimalité" d'un automate en se basant sur le langage qu'il reconnaît. Si ce langage est de la forme u* alors le nombre d'état

est |u|. Dans le cas contraire, le nombre d'état sera |x| où x est le plus petit mot du langage en dehors d'ε.

9.7. Deux automates équivalents

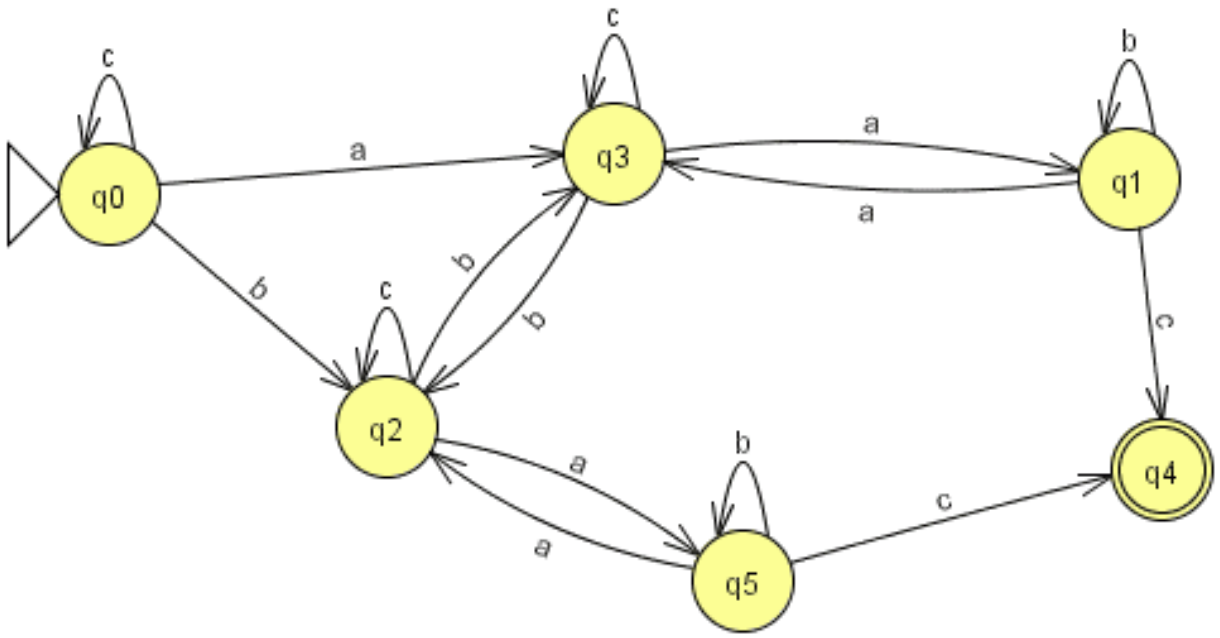
Deux automates finis sont équivalents s'ils reconnaissent le même langage. Cependant, il n'est pas toujours facile et même possible de se baser sur le langage pour les comparer (surtout s'il est infini !). Par contre, pour un langage donné, il n'existe qu'un unique AFD minimal (à la numérotation des états près). Par conséquent, pour comparer deux automates, il suffit, pour chacun d'eux, de calculer l'automate déterministe minimal équivalent et de comparer ces deux automates.

Pour comparer deux automates déterministe minimaux, il suffit de parcourir en parallèle les deux automates et de renuméroter les états rencontrés en fur et à mesure. A la fin du parcours, les deux tables de transitions doivent être identiques.

Remarque : il existe certainement d'autres techniques issues de la théorie des graphes à propos de la comparaison de graphes.

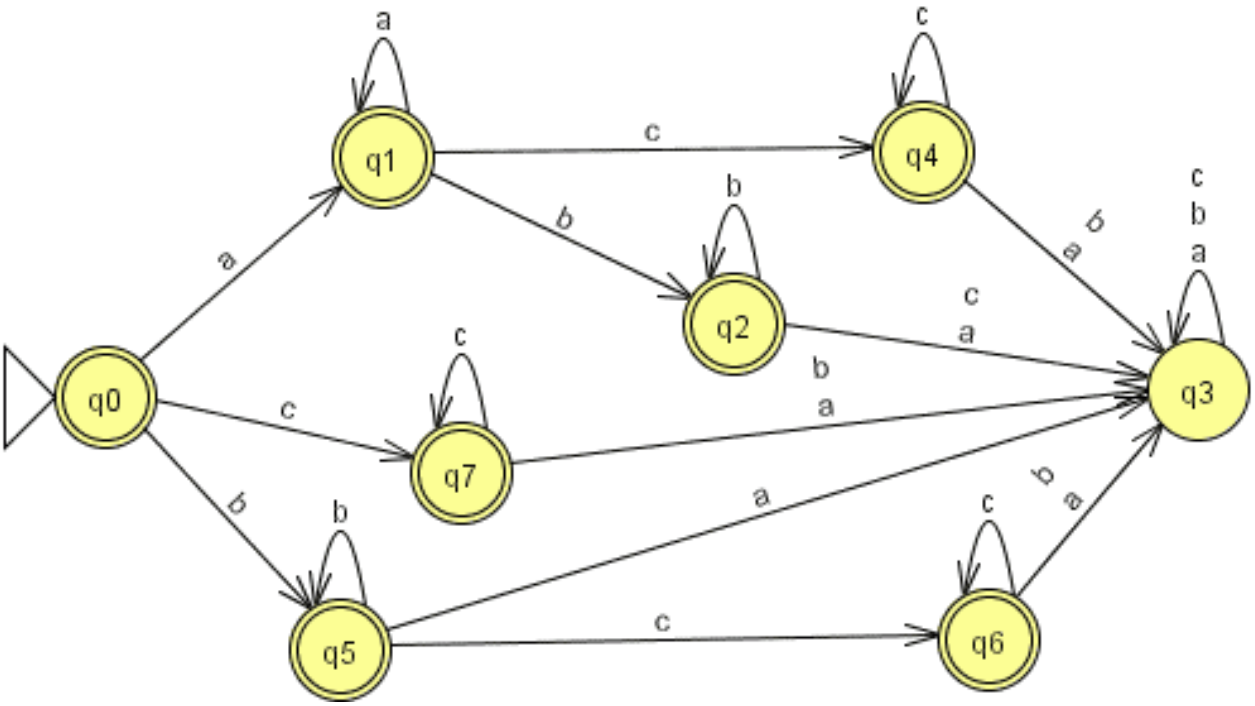
Exercices et tests :

Exercice 9.1. Soit l'automate déterministe suivant sur l'alphabet {a,b,c} (image issue de JFLAG) :



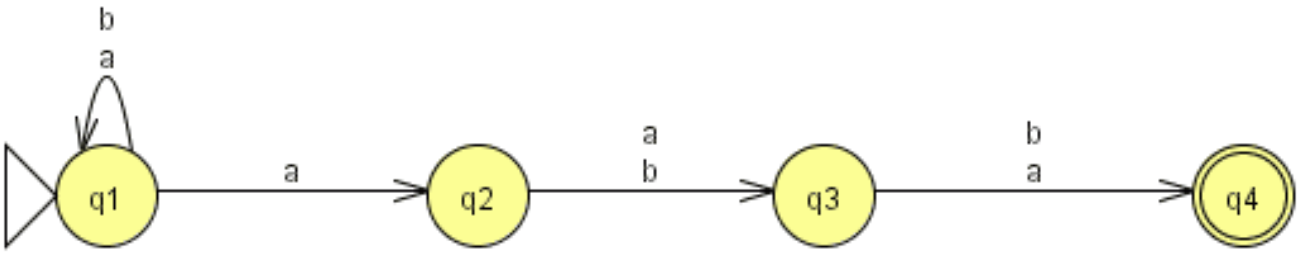
Donner l'automate déterministe minimal équivalent par la méthode de Moore (donner le détail de la méthode).

Exercice 9.2. Soit l'automate suivant sur l'alphabet {a,b,c} (image issue de JFLAG) :



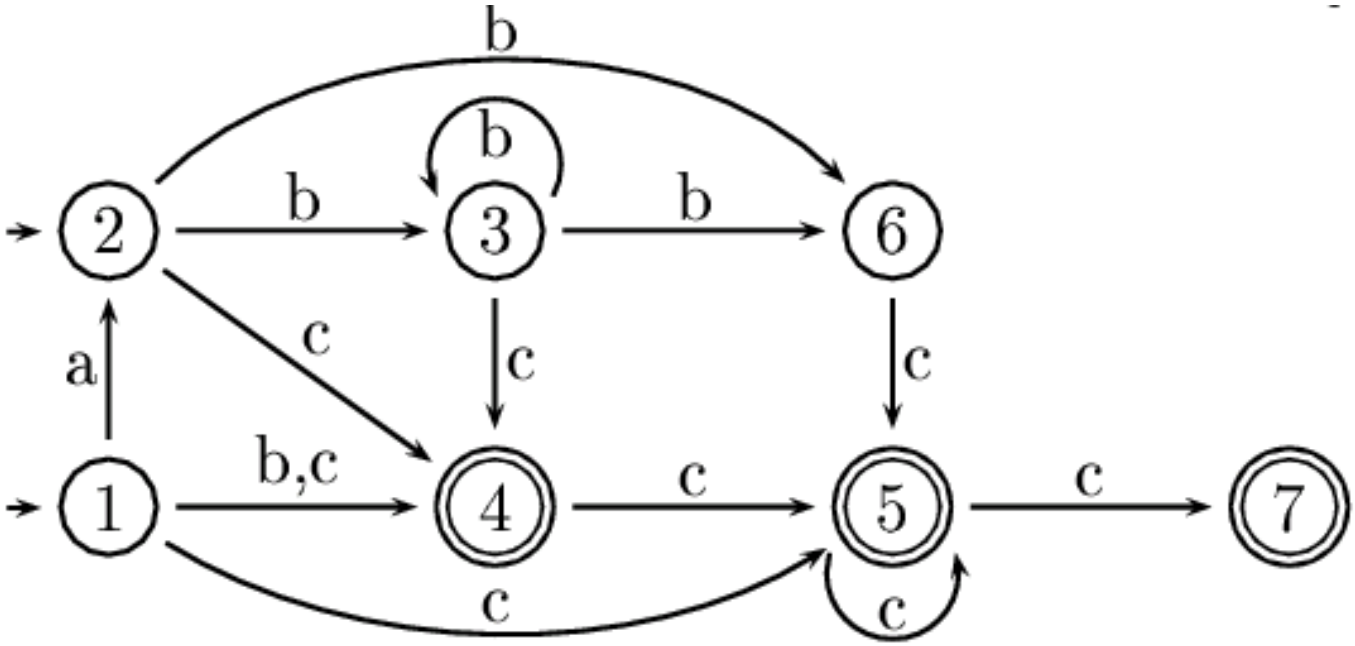
Donner l'automate déterministe minimal équivalent par la méthode de Moore (donner le détail de la méthode).

Exercice 9.3. Soit l'automate suivant sur l'alphabet {a,b} (image issue de JFLAG) :



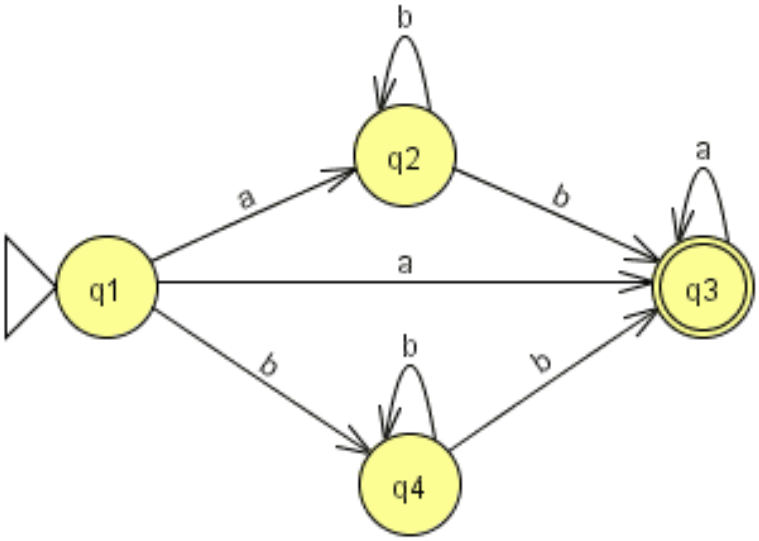
Donner l'automate déterministe minimal équivalent par la méthode de construction des sous-ensembles et la méthode de Moore (donner le détail des méthodes).

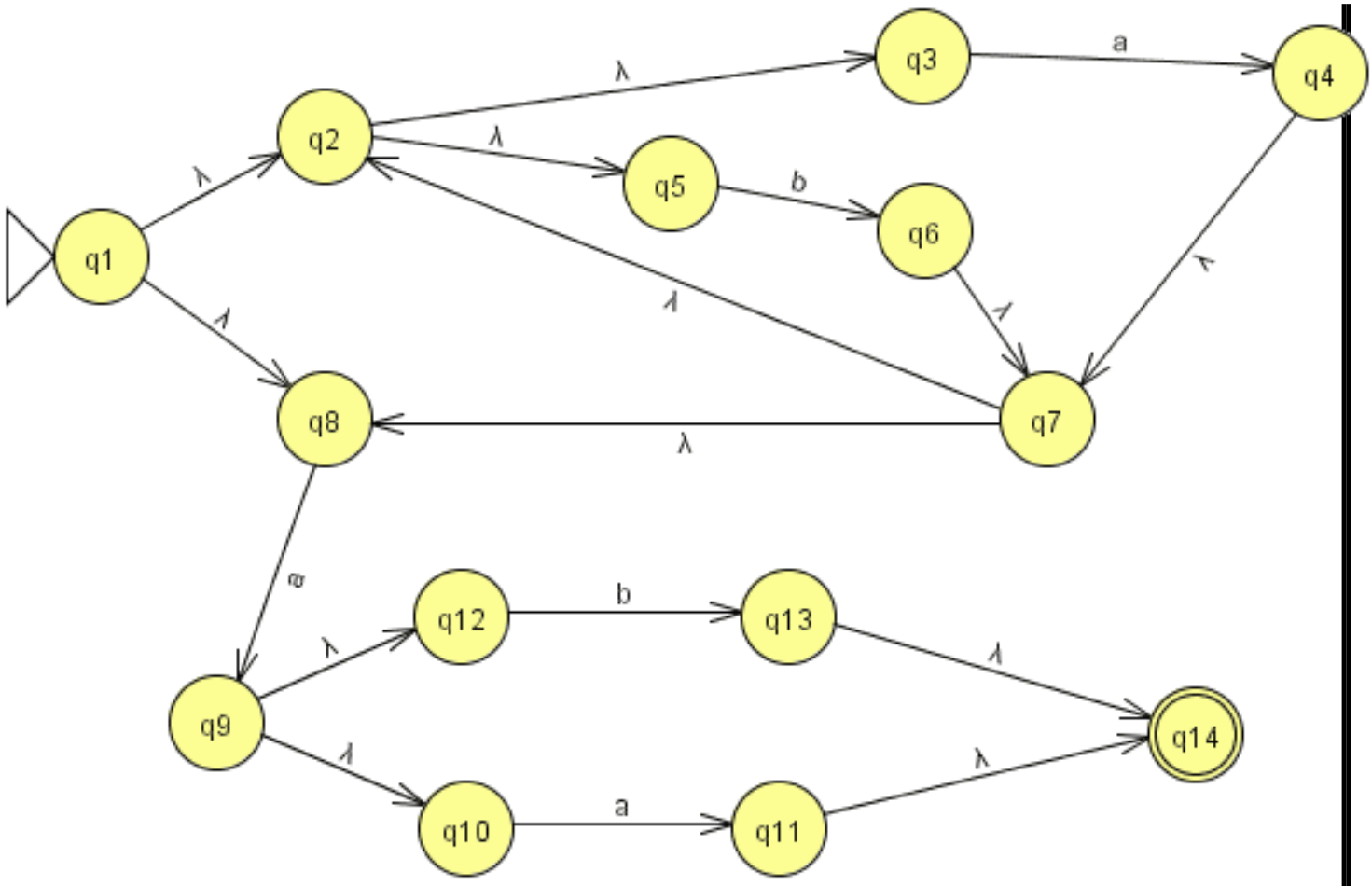
Exercice 9.4. Soit l'automate suivant sur l'alphabet {a,b,c} :



Donner l'automate déterministe minimal équivalent par la méthode de construction des sous-ensembles et la méthode de Moore (donner le détail des méthodes).

Exercice 9.5. Soit les automates suivant sur l'alphabet {a,b} (images issues de JFLAG):





Donner les automates déterministes minimaux équivalents respectifs par la méthode de construction des sous-ensembles et la méthode de Moore (donner le détail des méthodes)

10. Combinaisons d'automates finis

10.1. Introduction

Dans les sections précédentes, nous avons présenté des algorithmes pour optimiser des automates (déterminisation, minimisation). Dans cette section, nous allons aborder l'application d'opérateurs sur les automates : le produit de deux automates, la mise à l'étoile d'un automate et l'union de deux automates.

10.2. Produit de deux automates

Le produit de deux langages permet de construire un langage dont les mots résultent de la concaténation d'un mot du premier langage avec un mot du second langage. Dans le cas des langages reconnaissables, cette opération permet d'obtenir un langage reconnaissable.

Théorème du produit cartésien

Soient L_1 et L_2 deux langages reconnaissables alors le produit cartésien de L_1 par L_2, noté $L_1 \times L_2$, est reconnaissable.

Autrement dit, si $L_1 \in Rec(A^*)$ et $L_2 \in Rec(A^*)$ alors $L_1 \times L_2 \in Rec(A^*)$.

Démonstration :

Si les langages L_1 et L_2 sont reconnaissables alors il existe des automates finis les reconnaissant, notés respectivement $T_1=\{A_1, Q_1, I_1, F_1, \bullet_1\}$ et $T_2=\{A_2, Q_2, I_2, F_2, \bullet_2\}$. Si $L_1 \times L_2$ est reconnaissable alors il existe aussi un automate fini $T=\{A, Q, I, F, \bullet\}$. Il est possible de construire ce dernier. Prenons comme hypothèse que $Q_1 \cap Q_2 = \varnothing$ (quitte à renommer l'un ou l'autre).

- $A = A_1 \cup A_2$
- $Q = Q_1 \cup Q_2 \backslash \{i_2\}$
- $F = F_2$ si $\forall i \in I_2, i \notin F_2$
  $F = (F_1 \cup F_2) \backslash I_2$ sinon
- $\bullet = \bullet_1 \cup \bullet_2' \cup \bullet_2''$
  avec $\bullet_2' = \{(q,a,q') : (q \notin I_2)$ et $(q,a,q') \in \bullet_2\}$ (toutes les transitions de $\bullet_2$ sauf celles qui partent d'un état initial de T_2) et
  $\bullet_2'' = \{(q_1,a,q') : q_1 \in F_1, (i,a,q') \in \bullet_2, i \in I_2\}$ (toutes les transitions qui partent d'un état initial de T_2 partent des états finaux de T_1)

Il est possible de prouver que T reconnait bien le langage $L_1 \times L_2$ en utilisant les suites d'actions (ou de configurations). Autrement dit, il faut **prouver que $L(T) = L(T_1) \times L(T_2)$**

Démo produit

Pour cela, il suffit de montrer les deux propositions suivantes : $L(T_1) \times L(T_2) \subseteq L(T)$ et $L(T) \subseteq L(T_1) \times L(T_2)$.

Pour simplifier un peu, supposons que T_2 est standard donc $I_2=\{i_2\}$ (il est toujours possible de le construire)

I. Montrons que $L(T_1) \times L(T_2) \subseteq L(T)$.

Soit $w \in L(T_1) \times$, $w = w_1w_2$ avec $w_1 \in L(T_1)$ et $w_2 \in L(T_2)$.

I.a. Cas $\varepsilon \in L(T_2)$.

Donc $w = w_1$
$w_1 \in L(T_1) \Leftrightarrow \exists i_1 \in I_1$ tel que (w_1,i_1) -*/$T_1\rightarrow$ (ε, f_1), $f_1 \in F_1$.
Or si $\varepsilon \in L(T_2)$ alors $F = (F_1 \cup F_2)\backslash\{i_2\}$
Donc $f_1 \in F$ et $\forall q \in Q_1$, $q \in Q$
Donc (w_1,i_1) -*/$T\rightarrow$ (ε, f_1) donc $w=w_1 \in L(T)$. CQFD

I.b. Cas $\varepsilon \notin L(T_2)$.

Donc $w = w_1w_2$ avec $w_2=ax$, $a \in A$, $x \in A^*$
$w_1 \in L(T_1) \Leftrightarrow \exists i_1 \in I_1$ tel que (w_1,i_1) -*/$T_1\rightarrow$ (ε, f_1), $f_1 \in F_1$.
$w_2 \in L(T_2) \Leftrightarrow (w_2,i_2)$ -*/$T_2\rightarrow$ (ε, f_2), $f_2 \in F_2$.
$w_2=ax$, donc (ax,i_2)-$T_2\rightarrow$ (x,p) -*/$T_2\rightarrow$ (ε, f_2), $p \in Q_2$
$\forall q \in Q_1$, $q \in Q$ donc (w_1,i_1) -*/$T\rightarrow$ (ε, f_1) d'où (w_1w_2,i_1) -*/$T\rightarrow$ (w_2, f_1) (1)
$\forall q \in Q_2\backslash\{i_2\}$, $q \in Q$ donc (x,p) -*/$T\rightarrow$ (ε, f_2) (2)
$\forall f_1 \in F_1$ si $\mu_2(a,i_2)=p$ alors $\mu(a,f_1)=p$ (par construction de T)
Or (ax,i_2)-$T_2\rightarrow$ (x,p) donc $\forall f_1 \in F_1$ (ax,f_1) -$T\rightarrow$ (x,p) (3)
Avec (1), (2) et (3) on déduit que (w_1ax,i_1) -*/$T\rightarrow$ (ax, f_1) $T\rightarrow$ (x,p) -*/$T\rightarrow$ (ε, f_2)
Donc (w_1w_2,i_1) -*/$T\rightarrow$ (ε, f_2) d'où (w,i_1) -*/$T\rightarrow$ (ε, f_2)
Donc $w \in L(T)$. CQFD

II. Montrons que $L(T) \subseteq L(T_1) \times L(T_2)$.

II.a. $\varepsilon \in L(T)$

$\varepsilon \in L(T) \Leftrightarrow \exists i \in I$ tel que $i \in F$.
Or $i \in I = I_1$
$i \in F \Leftrightarrow I \cap F \neq \emptyset \Leftrightarrow I_1 \cap F \neq \emptyset$
Or $I_1 \cap F \neq \emptyset \Leftrightarrow F = (F_1 \cup F_2)\backslash\{i_2\}$ et $I_1 \cap F_1 \neq \emptyset$ (car comme $Q_1 \cap Q_2 = \emptyset$ alors $I_1 \cap F_2 = \emptyset$)
Donc $i \in F \Leftrightarrow \varepsilon \in L(T_2)$ et $\varepsilon \in L(T_1) \Leftrightarrow \varepsilon \in L(T_1) \times L(T_2)$. CQFD

II.b. $\varepsilon \notin L(T)$.

$w=ax \in L(T)$, $a \in A$, $x \in A^*$
$\exists i \in I$ tel que (w,i) -*/$T\rightarrow$ (ε, f), $f \in F$.
donc $(ax,i) \rightarrow (x,p)$ -*/$T\rightarrow$ (ε, f)

Si $f \in F_1$ alors $F = (F_1 \cup F_2)\backslash\{i_2\}$ donc $w=ax\varepsilon$ avec $ax \in L(T_1)$ (en effet, $f \in F_1$!) et $\varepsilon \in L(T_2)$ (en effet, $F = (F_1 \cup F_2)\backslash\{i_2\}$!)
donc $w \in L(T_1) \times L(T_2)$: CQFD

Si $f \in F_2$
Rappel : $(ax,i) \rightarrow (x,p)$ -*/T→ (ε, f)
Or : $\bullet = \bullet_1 \cup \bullet_2' \cup \bullet_2''$ Donc μ comprend les transitions de T_1 (donc sur Q_1), celles de T_2 (donc sur Q_2) sauf celles concernant i_2 et un ensemble de transitions allant d'un état de Q_1 vers un état de Q_2. Donc toute suite d'action dont la dernière configuration est dans Q_2 possède (1) des actions dans Q_1, (2) une action permettant de passer de Q_1 à Q_2 et (3) des actions dans Q_2.
Donc (x,i) -*/T→ (x',p) -1/T→ (x'',p') -*/T→ (ε, f) ou encore
(x,i) -*/T_1→ (x',p) -1/T→ (x'',p') -*/T_2→ (ε, f) avec i et p dans Q_1 (et même $i \in I_1$ et $p \in F_1$), p' et f dans Q_2 (et même $f \in F_2$). Donc, il existe une transition $\mu(a,p)=p'$ (issue de μ_2'') dans μ.
Soient, $x=yx'$, $x'=ax''$ ($y,x',x'' \in A^*$ et $a \in A$) alors :
(yax'',i) -*/T_1→ (ax'',p) -1/T→ (x'',p') -*/T_2→ (ε, f)

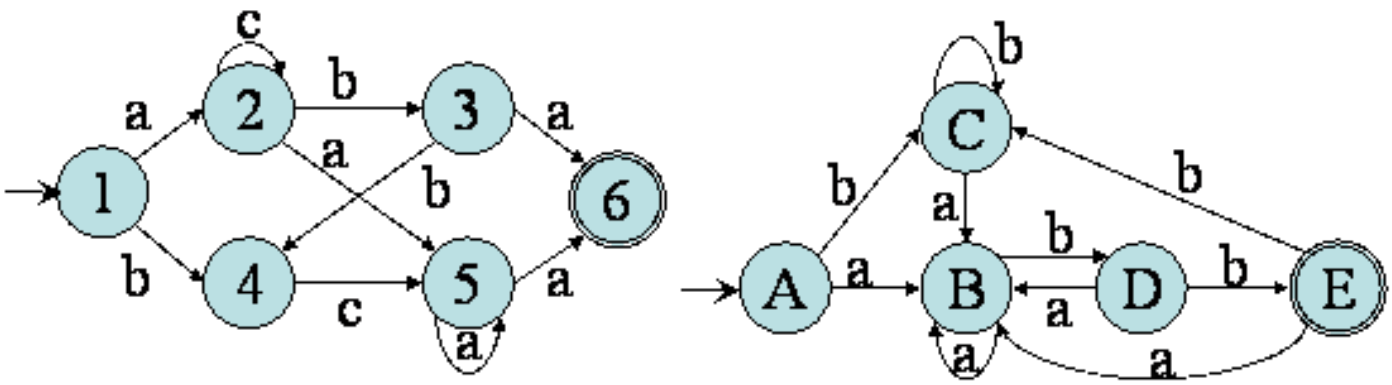
Donc $y \in L(T_1)$ (car $i \in I_1$ et $p \in F_1$).

(ax'',p) -1/T→ (x'',p') si et seulement si (ax'',i_2) -1/T→ (x'',p') (par construction de μ_2'')
Donc $ax'' \in L(T_2)$ (car $f \in F_2$)

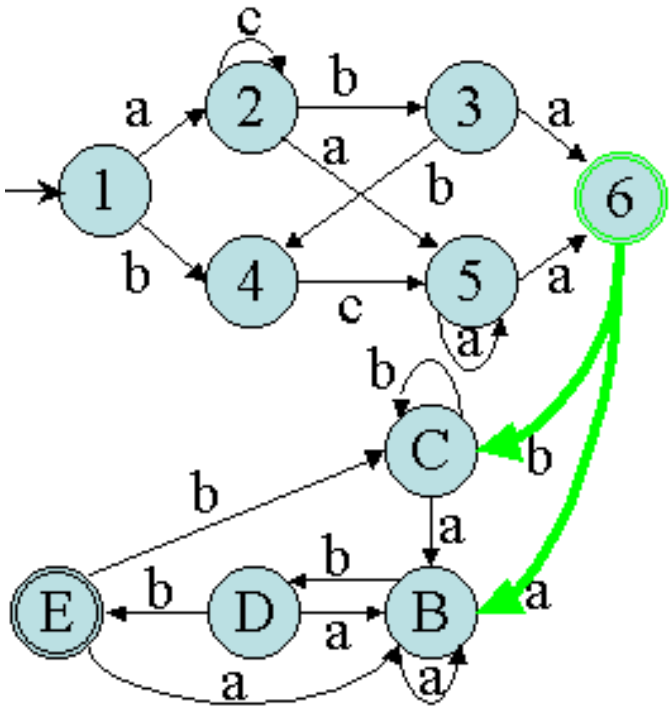
Donc pour tout mot $x=yax'' \in L(T)$ alors $y \in L(T_1)$ et $ax'' \in L(T_2)$ donc $x \in L(T_1) \times L(T_2)$.
CQFD

Cette démonstration nous permet de donner un méthode pour construire l'automate résultant du produit cartésien de deux langages reconnaissables. Par abus de langage, nous parlerons du produit (cartésien) de deux automates et ce sera noté $T_1 \times T_2$.

Pour illustrer cet algorithme, prenons les deux automates suivants :



On obtient dans ce cas :



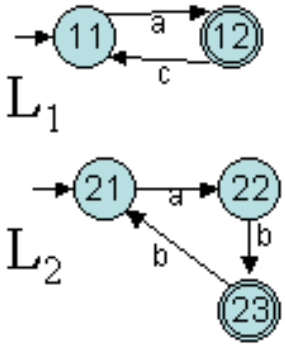
Remarques :

- Si A avait été final alors 6 serait resté un état final dans T.
- Un émondage est utile pour éventuellement supprimer les états initiaux de T_2 qui sont devenus inutiles.

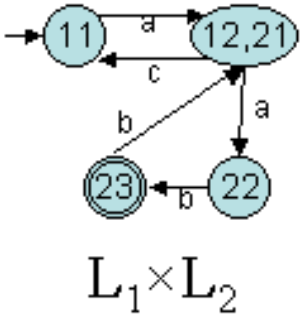
Dans le cas où T_2 est standard, une autre approche du produit (similaire) est une "fusion" entre

les états finaux de premier automate et l'état initial du second. L'état initial de T_2 est ensuite supprimé par émondage.

 Remarque : dans ce cas, T_2 doit absolument être standard donc $I_2=\{i_2\}$ (il est toujours possible de le construire), car il ne faut pas de transitions "retournant" sur l'état initial de T_2. En effet, de la manière où T est construit, il serait possible de revenir sur T_1 et donc de produire des mots de T_1 ! Prenons par exemple les langages L_1 et L_2 reconnus respectivement par les deux automates dans la figure suivante :



Le produit par fusion de ces deux automates $L = L_1 \times L_2$, donne :



L contient des mots comme "aab" qui est bien la concaténation de "a" du premier langage et "ab" dans le second langage. Par contre, L contient aussi "aabbcaab" qui ne peut pas être décomposé en un préfice dans L_1 et le reste dans L_2 !

10.3. Mise à l'étoile d'un automate

La mise à l'étoile d'un langage L, notée L*, est l'ensemble des produits cartésiens d'un langage avec lui-même à l'infini. Dans le cas des langages reconnaissables, cette opération permet d'obtenir un langage reconnaissable.

Théorème de la mise à l'étoile

Soient L un langage reconnaissable alors la mise à l'étoile de L, notée L*, est reconnaissable.

Autrement dit, si $L \in Rec(A^*)$ alors $L^* \in Rec(A^*)$.

Démonstration :

Si le langage L est reconnaissable alors il existe un automate fini le reconnaissant, noté T={A, Q, I, F, •}. Si L* est reconnaissable alors il existe aussi un automate fini T'={A, Q', I', F', •'}. Il est possible de construire ce dernier. Prenons comme hypothèse que T est standard donc I={i} (il est toujours possible de le construire).

- Q' = Q
- I' = I = {i}
- F' = F ∪ I (i devient final - s'il ne l'est pas déjà - car $\varepsilon \in L^*$)
- •' = • ∪ {(q,a,q') : q ∈ F, (i,a,q') ∈ •} (toutes les transitions qui partent de i partent des états finaux de T)

Il est aussi possible de prouver que T reconnaît bien le langage L* en utilisant les suites d'actions (ou de configurations). Autrement dit, on peut montrer que L(T) = L*.

Démo étoile

Cette démonstration peut être une démonstration par induction sur la longueur des mots. Pour cela, il suffit de s'appuyer sur une définition possible de L* par induction. Cette définition est la suivante :

- $\varepsilon \in L^*$ (cas de base 1)
- $\forall m \in L, m \in L^*$ (cas de base 2)
- Soit $m \in L^*$ et $n \in L^*$ alors $mn \in L^*$ (cas d'induction)

Cas de base

$\varepsilon \in L(T)^*$ et $\varepsilon \in L(T^*)$ (car $i \in F')^*$

$\forall m \in L(T), m \in L(T)^*$ et $m \in L(T^*)$ (car $\mu \subset \mu'$, I = I', $F \subset F'$)

Cas général

Soient $m_1 \in L(T)^*$, $m_1 \in L(T^*)$ et $m_2 \in L(T)^*$, $m_2 \in L(T^*)$ avec $m_2 \neq \varepsilon$
(si $m_2 = \varepsilon$ alors $m_1m_2 = m_1 \in L(T^*)$ et $\in L(T^*)$)

Par définition, $m_1m_2 \in L(T)^*$

$m_1 \in L(T^*) \Leftrightarrow (m_1,i)$ -*→ (&epsilon,f_1)
$m_2 \in L(T^*) \Leftrightarrow (m_2,i)$ -*→ (&epsilon,f_2)

$m_2 \neq \varepsilon \Leftrightarrow m_2 = am_3$ ($a \in A$, $m_3 \in A^*$)

Donc $(am_3,i) \rightarrow (m_3,p)$ -*→ (&epsilon,f_2)

Or si $(i,a,p) \in \mu$ alors $(i,a,p) \in \mu'$ et $\forall f \in F$, $(f,a,p) \in \mu'$

Donc $\forall f \in F$, $(am_3,f) \rightarrow (m_3,p)$ -*→ (&epsilon,f_2) est une suite d'actions valide dans T*.

Donc $(am_3,f_1) \rightarrow (m_3,p)$ -*→ (&epsilon,f_2) l'est aussi.

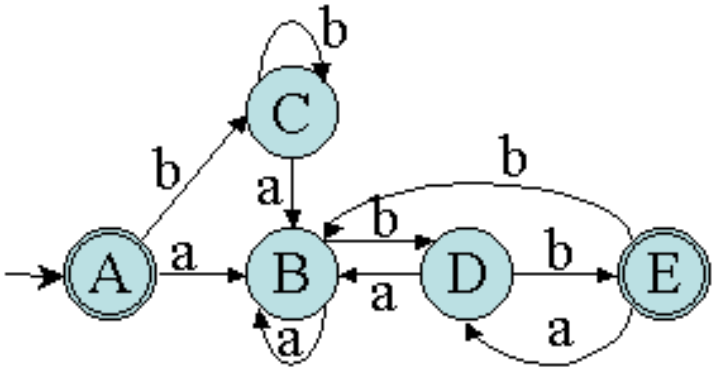
Or (m_1,i) -*→ (&epsilon,f_1) donc (m_1am_3,i) -*→ $(am_3,f_1) \rightarrow (m_3,p)$ -*→ (&epsilon,f_2) est valide.

Par conséquent, $m_1am_3 = m_1m_2 \in L(T^*)$

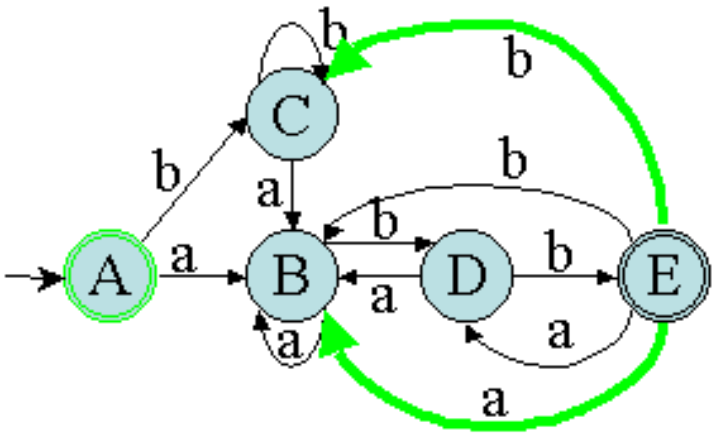
CQFD

Cette démonstration nous permet de donner un méthode pour construire l'automate résultant de la mise à l'étoile d'un langage reconnaissable. Par abus de langage, nous parlerons de la mise à l'étoile d'un automate et ce sera noté T*.

Pour illustrer cet algorithme, prenons l'automate suivant :

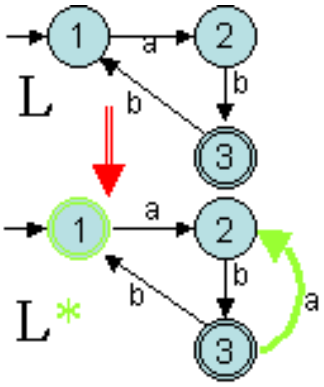


On obtient dans ce cas :



Remarque : T doit être absolument standard, car il ne faut pas de transitions retournant sur l'état initial de T. En effet, de la manière où T* est construit, il serait possible de terminer sur i qui n'était pas final dans T !

Prenons par exemple le langage L dans la figure suivante :



L reconnait des mots comme "ab", "abbab", "abbabbab"...L'automate n'est pas standard (à cause de la transition (3,b,1)). L* reconnaît bien évidemment ε mais aussi "abb" qui ne peut pas être constitué de concaténations de mots de L ! En effet, les mots les plus courts le L* sont "ε", "ab", "abab", "abbab"...

10.4. Union de deux automates

L'union de deux langages permet de construire un langage dont les mots sont issus soit d'un mot du premier langage soit d'un mot du second langage. Dans le cas des langages reconnaissables, cette opération permet d'obtenir un langage reconnaissable.

Théorème de l'union

Soient L_1 et L_2 deux langages reconnaissables alors l'union de L_1 et L_2, noté $L_1 \cup L_2$, est reconnaissable.

Autrement dit, si $L_1 \in Rec(A^*)$ et $L_2 \in Rec(A^*)$ alors $L_1 \cup L_2 \in Rec(A^*)$.

Démonstration :

Si les langages L_1 et L_2 sont reconnaissables alors il existe des automates finis les reconnaissant, notés respectivement $T_1=\{A_1, Q_1, I_1, F_1, \bullet_1\}$ et $T_2=\{A_2, Q_2, I_2, F_2, \bullet_2\}$. Si $L_1 \cup L_2$ est reconnaissable alors il existe aussi un automate fini $T=\{A, Q, I, F, \bullet\}$. Il est possible de construire ce dernier. Prenons comme hypothèse que $Q_1 \cap Q_2 = \varnothing$ (quitte à renommer l'un ou l'autre).

- $A = A_1 \cup A_2$
- $Q = Q_1 \cup Q_2$
- $I = I_1 \cup I_2$
- $F = F_1 \cup F_2$
- $\bullet = \bullet_1 \cup \bullet_2$

Il est assez facile de prouver que T reconnaît bien le langage $L_1 \cup L_2$ en utilisant les suites d'actions (ou de configurations) puisque l'on simule un parcours en parallèle des deux automates.

Cette démonstration nous permet de donner un méthode pour construire l'automate résultant de l'union de deux langages reconnaissables. Par abus de langage, nous parlerons de l'union de deux automates et ce sera noté $T_1 \cup T_2$.


Cette méthode de construction est très simple mais possède un inconvénient considérable : même si les deux automates de départ sont déterministes, le résultat n'est forcément pas déterministe. En effet, l'automate union possède toujours au moins deux états initiaux ($I = I_1 \cup I_2$) !

Il existe une autre méthode pour construire l'automate union pour qu'à partir de deux automates déterministe, cet automate union soit, lui aussi, déterministe. Soit $T_1=\{A_1, Q_1, I_1, F_1, \bullet_1\}$ et $T_2=\{A_2, Q_2, I_2, F_2, \bullet_2\}$ deux automates déterministes et complets. $T = T_1 \cup T_2 = \{A_1 \cup A_2, Q_1 \times Q_2, \{(i_1,i_2)\}, (F_1 \times Q_2) \cup (Q_1 \times F_2), \bullet\}$ avec $\bullet = \{((p_1,p_2),a,(q_1,q_2)) : (p_1,a,q_1) \in \bullet_1$ et $(p_2,a,q_2) \in \bullet_2\}$. Autrement dit, on construit un automate dont les états sont tous les couples possibles d'états entre T_1 et T_2 et les états finaux sont ceux ayant un état final de T_1 ou de T_2 . L'état initial est le couple des états initiaux des deux automates. Il n'y en a qu'un puisqu'ils sont déterministes.

Cette dernière méthode propose bien un automate déterministe et complet mais celui ci possède un très grand nombre d'états ($|Q_1| \times |Q_2|$) dont un grand nombre est inutile (pas accessibles et/ou co-accessibles).

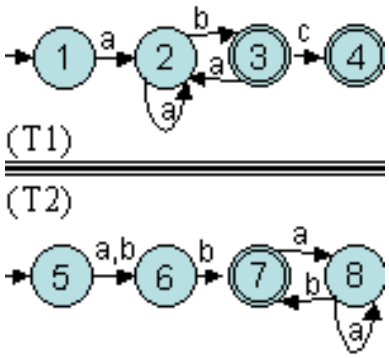
Il est cependant possible d'améliorer cet algorithme en ne construisant que des états accessibles. Pour cela, il suffit d'adapter la méthode utilisée pour rendre déterministe un AFN, c'est-à-dire l'algorithme par construction des sous-ensembles. Considérons T_1 et T_2 deux automates déterministes émondés (c'est toujours possible). L'état de départ est l'état composé du couple des états initiaux. Ensuite, pour chaque état créé (p_1,p_2), et chaque symbole a de l'alphabet :

- créer la transition $((p_1,p_2),a,(q_1,q_2))$ si $(p_1,a,q_1) \in \bullet_1$ et $(p_1,a,q_2) \in \bullet_2$ (ajouter l'état (q_1,q_2) s'il n'existe pas déjà et l'ajouter dans F si $q_1 \in F_1$ OU si $q_2 \in F_2$)
- créer la transition $((p_1,p_2),a,(q_1,\varnothing))$ si $(p_1,a,q_1) \in \bullet_1$ et $(p_1,a,q_2) \notin \bullet_2$ (ajouter l'état $(q_1,\varnothing)$ s'il n'existe pas déjà et l'ajouter dans F si $q_1 \in F_1$)
- créer la transition $((p_1,p_2),a,(\varnothing,q_2))$ si $(p_1,a,q_1) \notin \bullet_1$ et $(p_1,a,q_2) \in \bullet_2$ (ajouter l'état $(\varnothing,q_2)$ s'il n'existe pas déjà et l'ajouter dans F si $q_2 \in F_2$)

- si aucun des deux ne possède une transition sur a à partir de (p_1,p_2) il n'est pas nécessaire de construire un état ∅ qui sera supprimé au prochain émondage.

Cet algorithme construit un automate déterministe dont les états sont accessibles mais pas forcément co-accessibles. Il est nécessaire de le minimiser ensuite.

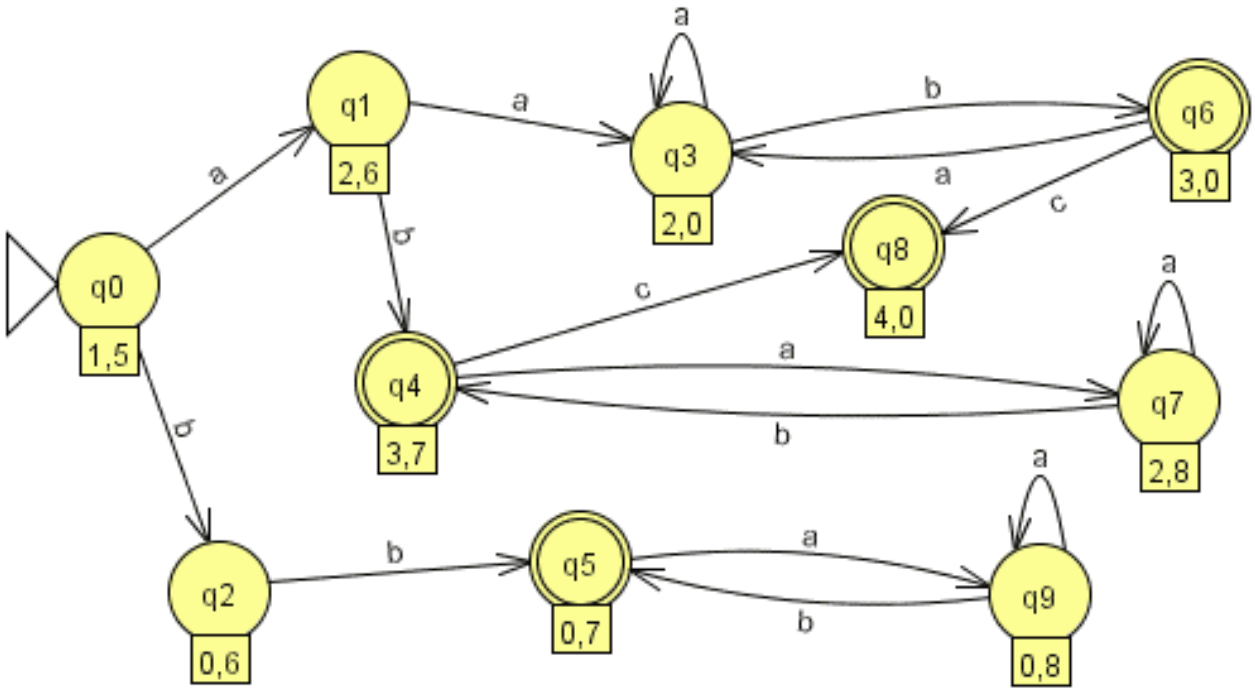
Pour illustrer cet algorithme, prenons les deux automates suivants :



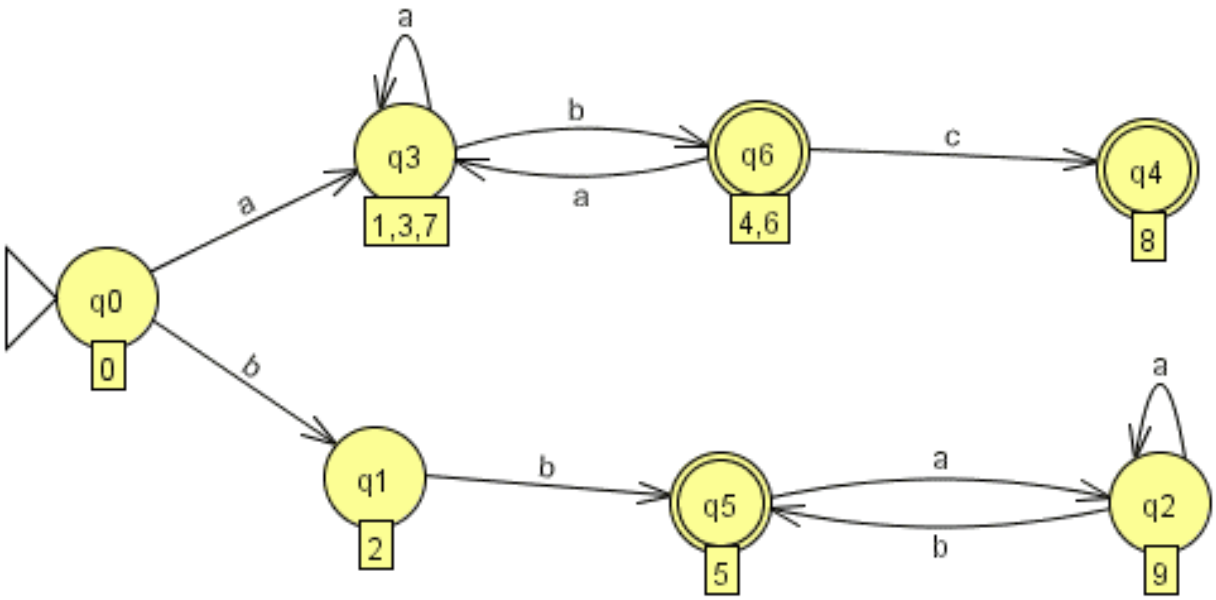
L'algorithme de construction permet d'obtenir le tableau suivant :

	a	b	c
1,5 ∈ I	2,6	0,6	0,0
2,6	2,0	3,7	0,0
0,6	0,0	0,7	0,0
2,0	2,0	3,0	0,0
3,7 ∈ F	2,8	0,0	4,0
0,7 ∈ F	0,8	0,0	0,0
3,0 ∈ F	2,0	0,0	4,0
2,8	2,8	3,7	0,0
4,0 ∈ F	0,0	0,0	0,0
0,8	0,8	0,7	0,0

On obtient dans ce cas (image issue de JFLAG) :



Après minimisation (comme quoi c'est nécessaire !), on obtient :



Une conséquence de ce théorème de l'union est le théorème suivant :

Théorème du langage fini

Tout langage de dimension finie est reconnaissable.

Avant de démontrer ce théorème, démontrons d'abord le lemme suivant :

Lemme du langage réduit à un mot

Tout langage réduit à un mot est reconnaissable.

Démonstration du lemme :

L'automate permettant de reconnaître un unique mot $x = a_1a_2...a_n$ se construit avec n+1 états noté de 1 à n+1, 1 est l'état initial, n+1 l'état final et les transitions $\forall i \in [1..n]$, $\bullet(a_i,n)=n+1$.

Démonstration du théorème :

Un langage fini possède un nombre fini de mots, soit m ce nombre. Ce langage peut donc être construit comme étant l'union de m langages réduits à un seul mot. Or, par le lemme précédent, chacun de ces langages est reconnaissable et, de plus, l'union de langages reconnaissables est reconnaissable. CQFD

Exercices et tests :

Exercice 10.1. Démontrer le théorème suivant :

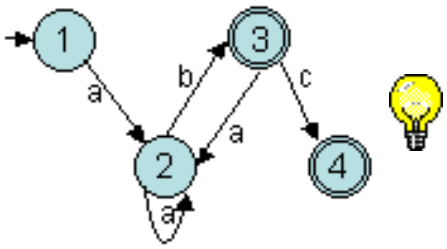
Théorème de la complémentation

Soient L un langage reconnaissable alors le complémentaire de L, noté Comp(L), est reconnaissable.

Autrement dit, si $L \in Rec(A^*)$ alors $Comp(L) \in Rec(A^*)$.

En déduire une méthode pour construire l'automate "complémentaire" d'un automate.

Appliquer la méthode sur l'automate suivant :



Exercice 10.2. Démontrer le théorème suivant :

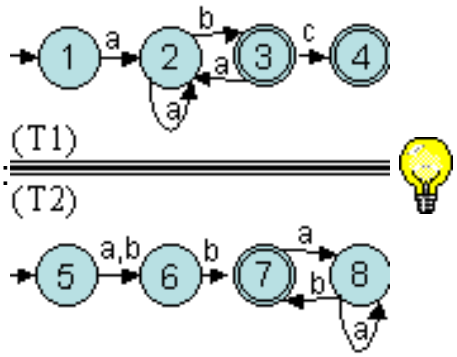
Théorème de l'intersection

Soient L_1 et L_2 deux langages reconnaissables alors l'intersection de L_1 et L_2, noté $L_1 \cap L_2$, est reconnaissable.

Autrement dit, si $L_1 \in Rec(A^*)$ et $L_2 \in Rec(A^*)$ alors $L_1 \cap L_2 \in Rec(A^*)$.

En déduire une méthode pour construire l'automate issu de "l'intersection" de deux automates. Faites en sorte qu'il soit déterministe si les deux automates de départ le sont.

Appliquer la méthode sur les automates suivants :



Exercice 10.3. Soit L en langage et $a_1a_2...a_n$ un mot de L alors le transposé de L, noté L^{-1} contiendra le mot $a_n...a_2a_1$. Démontrer le théorème suivant :

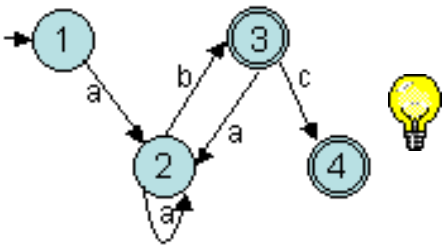
Théorème du transposé

Soient L un langage reconnaissable alors le transposé de L, noté L^{-1}, est reconnaissable.

Autrement dit, si $L \in Rec(A^*)$ alors $L^{-1} \in Rec(A^*)$.

En déduire une méthode pour construire l'automate "transposé" d'un automate.

Appliquer la méthode sur l'automate suivant :



Exercice 10.4. Démontrer le théorème suivant :

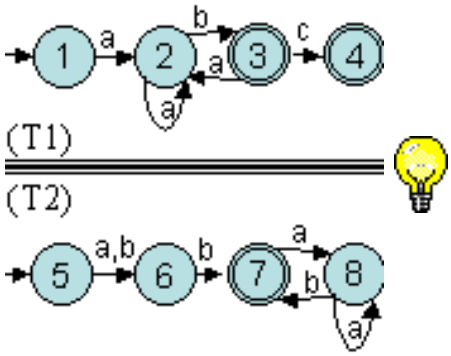
Théorème de l'exclusion

Soient L_1 et L_2 deux langages reconnaissables alors la différence de L_1 et L_2, noté $L_1 - L_2$ ou $L_1 \backslash L_2$, est reconnaissable.

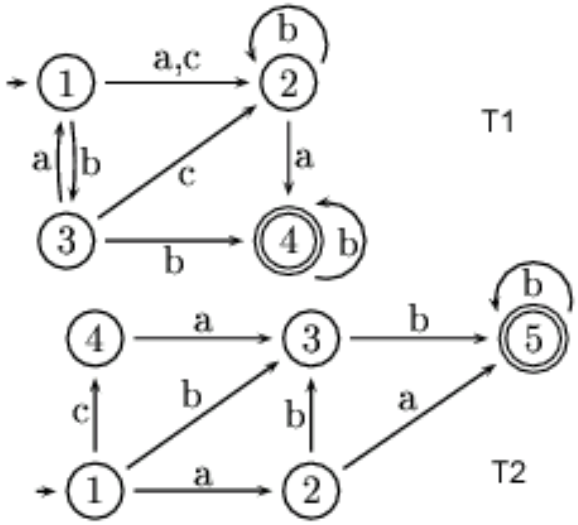
Autrement dit, si $L_1 \in Rec(A^*)$ et $L_2 \in Rec(A^*)$ alors $L_1 \backslash L_2 \in Rec(A^*)$.

En déduire une méthode pour construire l'automate issu de "la différence " de deux automates. Faites en sorte qu'il soit déterministe si les deux automates de départ le sont

Appliquer la méthode sur les automates suivants :



Exercice 10.5. Soit les automates T1 sur {a,b,c} et T2 sur {a,b,c,d} suivants :



Construire les automates ε-libres suivants (à chaque fois, précisez l'alphabet) :

1. Comp(T1)
2. T1*
3. T1 ∩ T2
4. T1 ∪ T2
5. T1 × T2
6. T1\T2

Langages rationnels et reconnaissables

1. Introduction

Dans les chapitres précédents, nous avons étudié les langages rationnels, les langages décrit par des grammaires régulières et les langages reconnaissables.

L'objectif de ce chapitre est de montrer que tous ces langages correspondent à la même classe de langage. Nous présenterons d'abord le théorème de Kleene qui prouve cette équivalence. Ensuite, nous présenterons comment passer d'une expression rationnelle à un automate fini (déterministe minimal). Puis, nous présenterons le processus inverse consistant à déterminer l'expression rationnelle décrivant le langage reconnu par un automate fini. Enfin, nous présenterons le lien entre grammaires rationnelles, automates finis et expressions rationnelles.

2. Le théorème de Kleene

2.1. Présentation

Dans [Kleene56], Stephen Cole Kleene (1909-1994) démontre une sorte de théorème de représentation qui établit l'**équivalence entre deux modes de définition du même objet**. Ces objets sont ce que Kleene appelle **les événements réguliers**, aujourd'hui souvent appelés **langages** plutôt qu'événements et qualifiés de **rationnels** plutôt que de **réguliers**. Le terme d'événement renvoie bien sûr à la terminologie du calcul des probabilités mais aussi au point de vue de départ de cette théorie qui s'adresse à la **description** de phénomènes plutôt qu'à la **spécification** de propriétés formelles, comme de nos jours.

2.2. Le théorème

Les événements réguliers sont ceux que l'on peut décrire à partir d'événements de base en utilisant les opérateurs de :

- l'union ensembliste (équivalent au 'ou' logique) ;
- la concaténation ;
- l'itération (notée par une étoile '*').

Théorème de KLEENE

Les événements, que l'on peut décrire à partir d'événements de base en utilisant les trois opérateurs d'union, de concaténation et d'itération, sont exactement ceux que l'on peut spécifier à l'aide d'un automate fini.

Dém.

Il est possible de construire un AFN reconnaissant un langage rationnel par induction à partir des symboles et des trois opérateurs de base. En effet, si l'on reprend d'une part le lemme du langage réduit à un mot et d'autre part les théorèmes du produit cartésien (×), de la mise à l'étoile (*) et de l'union (∪), à toute étape de la construction d'un langage rationnel correspond la construction d'un automate fini reconnaissant le même langage. Plus formellement, cela donne :

- $\emptyset \in Rat(A^*) \Leftrightarrow \emptyset = L(\{A, \{1\}, \{1\}, \emptyset, \emptyset\})$ (par convention)
- $\{\varepsilon\} \in Rat(A^*) \Leftrightarrow \{\varepsilon\} = L(\{A, \{1\}, \{1\}, \{1\}, \emptyset\})$
- $\forall a \in A, \{a\} \in Rat(A^*) \Leftrightarrow \{a\} = L(\{A, \{1,2\}, \{1\}, \{2\}, \{\bullet(a,1)=2\}\})$
- Soit I_1 décrivant $L_1 \in Rat(A^*)$, $L_1 = L(T_1) = L(\{A, Q_1, I_1, F_1, \bullet_1\})$ et I_2 décrivant $L_2 \in Rat(A^*)$, $L_2 = L(T_2) = L(\{A, Q_2, I_2, F_2, \bullet_2\})$:
  - $I_1|I_2$ décrit $L_1 \cup L_2 \Leftrightarrow L_1 \cup L_2 = L(T_1) \cup L(T_2) = L(T_\cup)$

- $l_1 l_2$ décrit $L_1 \times L_2 \Leftrightarrow L_1 \times L_2 = L(T_1) \times L(T_2) = L(T_x)$
- l_1^* décrit $L_1^* \Leftrightarrow L_1^* = L(T_1)^* = L(T_1^*)$

Théorème de KLEENE (conséquence1)

Chaque ensemble rationnel est reconnu par un AFD à nombre d'états minimum qui est unique à un renommage des états près et réciproquement. Autrement dit, un langage est accepté par un AFD si et seulement si c'est un langage rationnel.

Autrement dit :
$L \in Rat(A^*) \Leftrightarrow L \in Rec(A^*)$

Pour effectuer le passage d'une expression régulière à un automate fini déterministe minimal, une méthode possible consiste à construire d'abord un AFN reconnaissant le même langage, de le transformer en un AFD équivalent puis, finalement, de le réduire. Pour les deux dernières phases, nous avons déjà présenté une méthode adaptée. Intéressons-nous désormais à la phase de construction de l'AFN à partir d'une expression régulière. Le processus inverse est aussi possible. Nous verrons cela plus loin.

Il n'est pas toujours aisé de prouver qu'un langage (souvent décrit pas une expression) est rationnel. Nous avons déjà présenté quelques idées. Le théorème de Kleene nous permet de proposer une autre stratégie.

Pour montrer qu'un langage est rationnel, il suffit de construire un automate fini reconnaissant ce langage. Parfois, la construction de cet automate est plus aisée que la construction d'une expression rationnelle canonique.

Théorème de KLEENE (conséquence2)

Le complémentaire d'un langage rationnel est un langage rationnel

L'intersection de deux langages rationnels est un langage rationnel

Démonstrations

Pour la première partie, il suffit de se rappeler que le complémentaire d'un langage reconnaissable est un langage reconnaissable. Si on ajoute à cela le théorème de Kleene CQFD

Pour la second partie, il suffit de se rappeler que l'intersection de deux langages reconnaissables est un langage reconnaissable. Si on ajoute à cela le théorème de Kleene CQFD

2.3. Automate fini et grammaire

Grâce au théorème de Kleene, il est possible de déduire qu'à tout automate fini il est possible d'associer une grammaire linéaire droite.

Théorème de l'équivalence AFN et grammaire linéaire droite

Soit T un AFN tel que $L(T)=L$ alors il existe une grammaire linéaire droite LD telle que $L=L(LD)$.

Dém. (par construction)

Une dérivation de la grammaire correspond à un mouvement de l'automate. Soit T=(A, Q, I, F, •), on pose G=(A, Q, I, S) où

S est défini par :

- si •(a, q)=r alors q → ar ∈ S ;
- si p ∈ F alors p → ε ∈ P.

3. Construction d'un AFN à partir d'une expression rationnelle

Du fait de l'équivalence entre langages reconaissables et langages rationnels, des méthodes permettant de passer d'une expression rationnelle à un AFN ont été mises au point.

Remarque : un intérêt de cette conversion est la possibilité de mettre en place des traitements de validation de suites de symboles ou d'évènements dans des langages ne gérant pas les expressions rationnelles ou lorsque les traitements associés ne sont pas facilement descriptibles ou pas très performants dans ces langages.

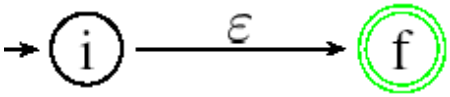
3.1. Méthode de Thompson

Il existe de nombreuses stratégies pour construire de façon automatique un automate fini à partir d'une expression rationnelle [Watson93a] (cette référence présente de manière formelle différents algorithmes dont celui de Thompson). L'algorithme présenté ici est l'algorithme de construction de Thompson [Thompson68]. Plusieurs variantes sont possibles. Celle présentée ici est simple et surtout facile à implanter (ce qui n'est pas le cas de toutes les méthodes).

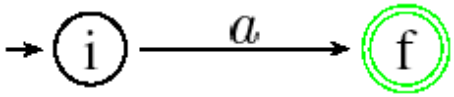
L'algorithme est dirigé par la syntaxe, c'est-à-dire qu'il utilise la structure syntaxique de l'expression rationnelle pour guider le processus de construction. Il est récursif sur l'arbre syntaxique de l'expression rationnelle. A partir d'une expression r sur un alphabet A, on cherche à construire un AFN T(r) qui reconnaît L(r) (le langage reconnu par r).

Si l'expression est réduite à un symbole s alors on construit l'automate :

- {A, {i,f}, {i}, {f}, {•(ε,i)=f}} pour ε. Cet automate reconnaît alors le langage {ε}.

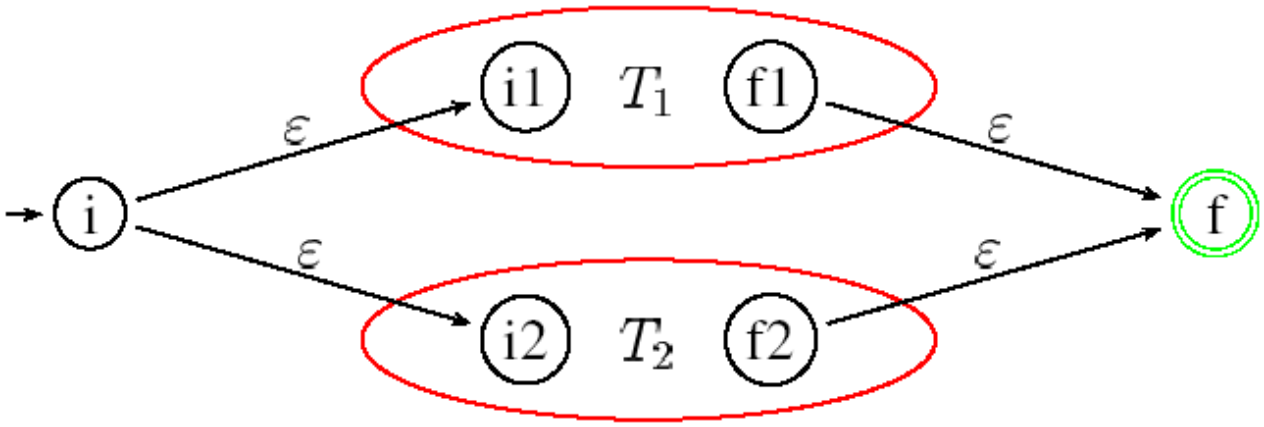


- {A, {i,f}, {i}, {f}, {•(a,i)=f}} pour tout symbole a ∈ A. Cet automate reconnaît alors le langage {s}.

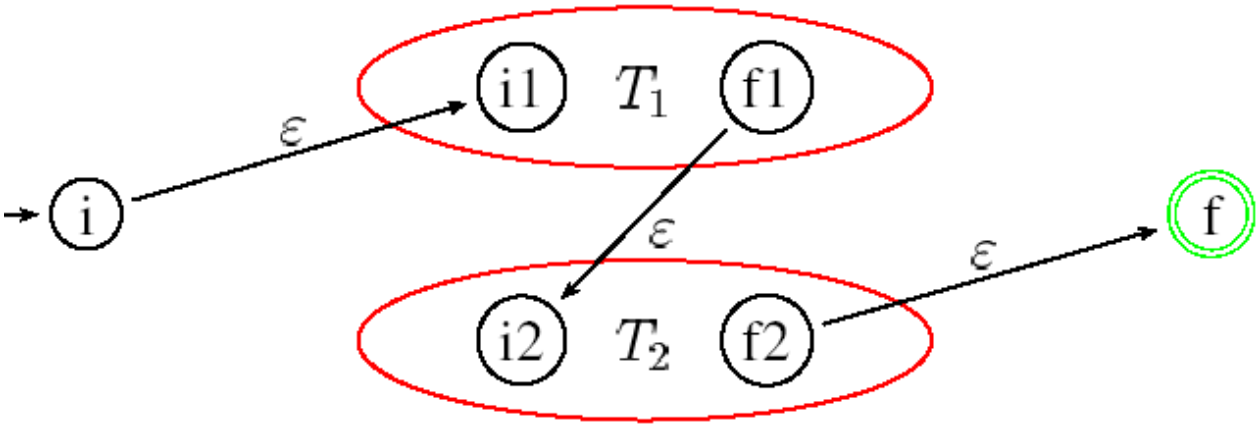


Si l'expression n'est pas un symbole, il faut alors la décomposer selon l'opérateur utilisé puis construire les automates $T(r_1)=T_1=\{A, Q_1, I_1=\{i_1\}, F_1=\{f_1\}, \bullet_1\}$ et $T(r_2)=T_2=\{A, Q_2, I_2=\{i_2\}, F_2=\{f_2\}, \bullet_2\}$ associés aux opérandes r_1 et r_2. Enfin, selon le type d'expression :

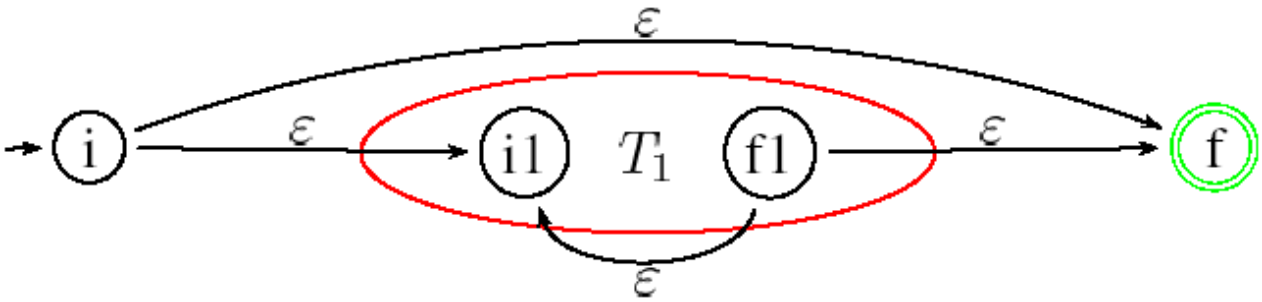
- $r_1|r_2$: on construit l'automate $T(r_1|r_2) = \{A, Q_1 \cup Q_2 \cup \{i,f\}, \{i\}, \{f\}, \bullet_1 \cup \bullet_2 \cup \{\bullet(\varepsilon,i)=i_1; \bullet(\varepsilon,i)=i_2; \bullet(\varepsilon,f_1)=f; \bullet(\varepsilon,f_2)=f\}$ où i est un nouvel état de départ et f un nouvel état d'acceptation. Il y a une ε-transition depuis i vers les états de départ de $T(r_1)$ et $T(r_2)$ et depuis les états finaux de $T(r_1)$ et $T(r_2)$ vers f. Les états de départ et d'acceptation de $T(r_1)$ et $T(r_2)$ sont des états de transition pour $T(r_1|r_2)$. Tout chemin depuis i vers f doit traverser soit $T(r_1)$ soit $T(r_2)$ exclusivement donc l'automate composé reconnaît $L(r_1) \cup L(r_2)$.



- r_1r_2 : on construit l'automate $T(r_1r_2) = \{A, Q_1 \cup Q_2 \cup \{i,f\}, \{i\}, \{f\}, \bullet_1 \cup \bullet_2 \cup \{\bullet(\varepsilon,i)=i_1; \bullet(\varepsilon,f_1)=i_2; \bullet(\varepsilon,f_2)=f\}\} = \{A, Q_1 \cup Q_2, \{i_1\}, \{f_2\}, \bullet_1 \cup \bullet_2 \cup \{\bullet(\varepsilon,f_1)=i_2\}\}$ (il est aussi possible de fusionner f_1 et i_2)
où l'état de départ de $T(r_1)$ devient celui de $T(r_1r_2)$ et l'état d'arrivée de $T(r_2)$ devient celui de $T(r_1r_2)$. L'état d'acceptation de $T(r_1)$ et celui de départ de $T(r_2)$ (états de transition) peuvent être fusionnés. Un chemin depuis i_1 vers f_2 doit d'abord traverser $T(r_1)$ puis $T(r_2)$ sans retour en arrière possible. Donc, $T(r_1r_2)$ reconnaît le langage $L(r_1)$ $L(r_2)$.



- r_1^* : on construit l'automate $T(r_1^*) = \{A, Q_1 \cup \{i,f\}, \{i\}, \{f\}, \bullet_1 \cup \{\bullet(\varepsilon,i)=i_1; \bullet(\varepsilon,i)=f; \bullet(\varepsilon,f_1)=i_1; \bullet(\varepsilon,f_1)=f\}\}$
où i est un nouvel état de départ et f un nouvel état d'acceptation. On peut aller soit directement de i à f en suivant une ε-transition qui représente le fait que $\varepsilon \in L(r_1^*)$ soit aller de i à f en traversant $T(r_1)$ une ou plusieurs fois. Cet automate reconnaît le langage $L(r_1^*)$.



- (r_1) : on construit l'automate $T((r_1)) = T(r_1)$.

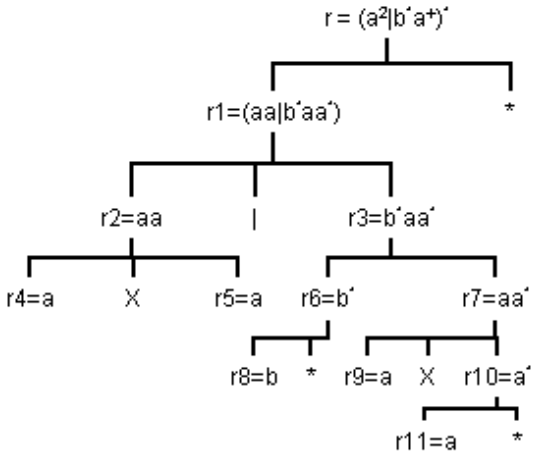
Chaque fois que l'on construit un nouvel état, on lui donne un nom distinct pour qu'il n'y ait pas deux états de même nom. Même si un symbole apparaît plusieurs fois dans l'expression rationnelle, on crée pour chaque symbole un AFN séparé avec ses propres états. Chaque étape de la construction produit un AFN qui reconnaît le langage correct.

L'AFN ainsi produit possède les propriétés suivantes :

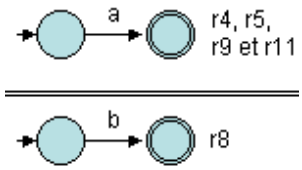
- il possède au plus deux fois plus d'états qu'il n'y a de symboles et d'opérateurs dans l'expression (i.e. $|Q| \leq 2(n+m)$ où n est le nombre de symboles et m le nombre d'opérateurs) ;
- il a exactement un état de départ et un état d'acceptation. On peut même aller plus loin, il est normalisé.
- Chaque état de l'automate a soit une transition sortante sur un symbole de A soit au plus deux ε-transitions sortantes.

- Il n'y a pas de boucles d'ε-transitions. Il n'est pas possible de revenir sur un état en suivant uniquement des ε-transitions.

Appliquons de cet algorithme sur l'expression régulière $(a^2 \mid b^*a^+)^*$. Tout d'abord, il faut construire l'arbre syntaxique associé à cette expression (attention, plusieurs arbres sont souvent possibles) :

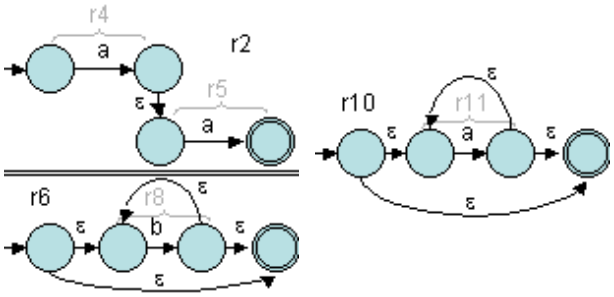


Ensuite, il suffit d'appliquer les règles de Thompson en partant des feuilles. Donc, il faut d'abord construire r4, r5, r8, r9 et r11.

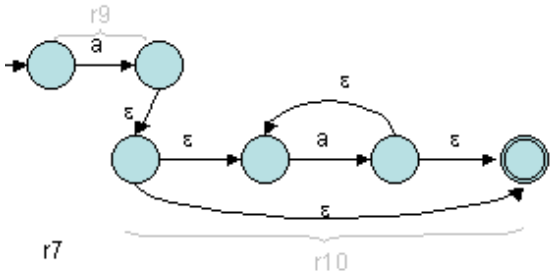


Remarque : il n'est pas toujours nécessaire d'étiqueter (numéroter) les états. Dans ce type d'algorithme, il n'est pas utile de les repérer (pas nécessaire de les identifier).

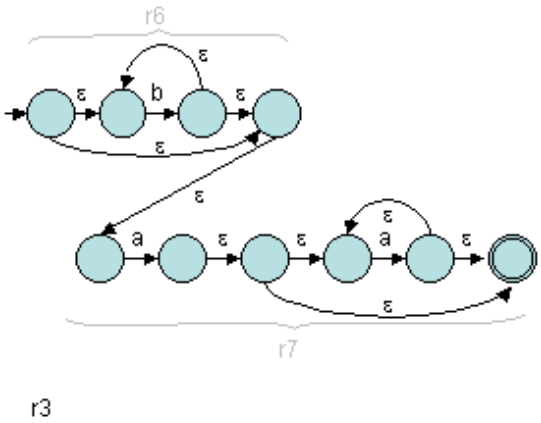
Ensuite, il faut construire r2, r6 et r10 :



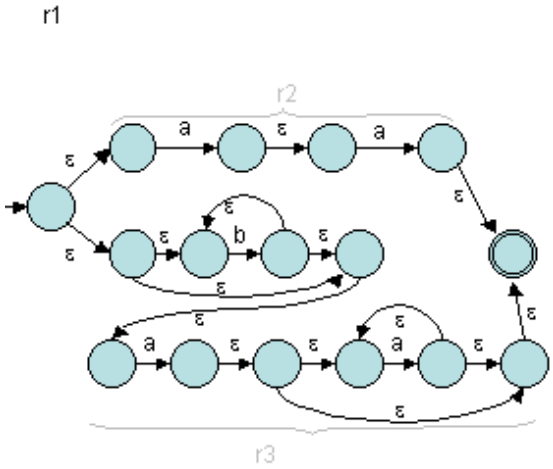
Puis, on construit r7 :



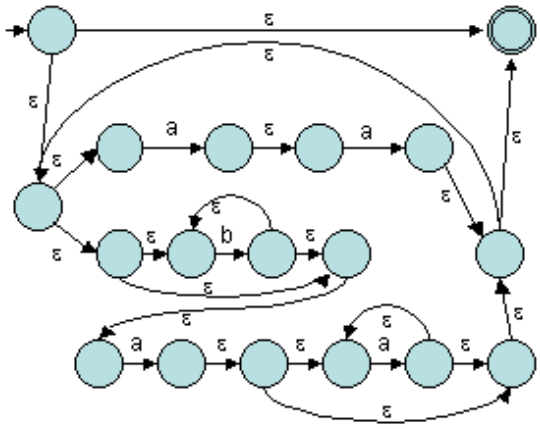
Puis, c'est le tour de r3 :



Ensuite, c'est r1 :

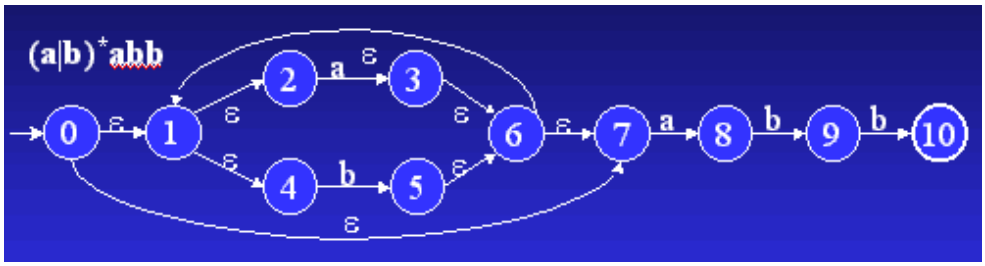


Pour terminer, r permet de construire l'automate suivant :



Ouf !

Sur l'expression (a|b)*abb, Thompson permet d'obtenir l'automate suivant :



Ici, le produit est effectué par fusion de l'état final du premier avec l'état initial du second.

Remarque : il est possible de construire directement un AFD à partir d'une expression régulière dite "étendue", c'est-à-dire à laquelle on ajoute en fin le caractère #. Nous n'aborderons pas cette méthode ici (voir [ASU91], pp.159-165).

3.2. Méthode de Glushkov

Bien évidemment, l'algorithme de Thompson n'est pas le seul possible. Il en existe de nombreux dont celui de Glushkov. Cette algorithme est intéressant, car il construit un automate ε-libre et homogène.

Le principe de cette méthode est le suivant : si tous les symboles de l'expression sont différents alors il est possible d'associer un état à chaque symbole et une transition est présente entre deux états si les deux symboles associés se suivent dans un mot du langage.

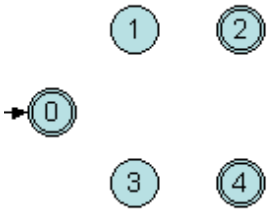
Plus précisément, cette méthode se divise en 4 étapes :

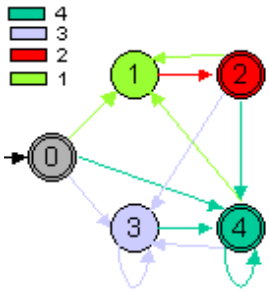
1. numéroter de 1 à n les symboles de l'expression (un symbole présent plusieurs fois sera numéroté autant de fois) ;
2. créer un état par numéro, l'état initial 0 et rendre final tout état dont le symbole associé peut terminer un mot du langage (0 est final si ε est un mot de ce langage) ;
3. créer une transition •(j,i)=j s'il existe le facteur $x_i x_j$ dans un mot du langage et •(j,0)=j si le symbole associé à l'état j peut commencer un mot ;
4. remplacer les numéros des transitions par le symbole correspondant dans l'expression.

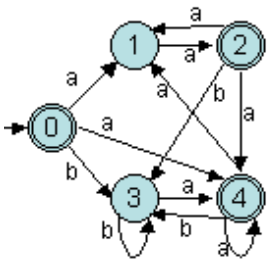
Pour l'étape 1, si un symbole représente une classe de symboles alors on peut se contenter de ce symbole mais en phase 4 il faudra créer autant de transitions qu'il y a de symboles de l'alphabet dans la classe.

Il faut noter aussi que l'expression ne doit pas contenir ε afin de ne pas générer un automate avec des ε-transitions.

Reprenons notre exemple habituel, l'expression $(a^2| b^*a^+)^*$. On passe alors par les étapes suivantes :

1. $(a_1a_2|b_3^*a_4^+)^*$
2. 

3. 

(pour alléger l'automate, une couleur est associée à chaque numéro)
4. 

De cet exemple, on déduit que l'automate produit à de grandes chances de ne pas être déterministe.

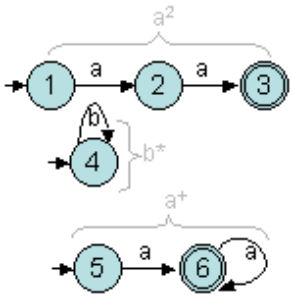
3.3. Autres méthodes

Du théorème de Kleene et de ses conséquences, nous en déduisons facilement une méthode manuelle pour construire un AFN à partir d'une expression rationnelle. Il suffit de décomposer l'expression comme une suite d'opérations (produit, union et étoile) sur des langages typiques ou réduits à un mot voire à un symbole. Puis on construit les automates finis correspondant à ces langages "élémentaires" (Cf. la démonstration du lemme du langage réduit à un mot). On applique ensuite les opérateurs sur ces automates (Cf. démonstrations des théorèmes du produit cartésien, de la mise à l'étoile et de l'union).

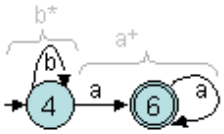
Remarque : la décomposition peut être arrêtée dès que l'on voit apparaître une expression dont l'automate est "classique" (a^*, a^+, $a?$, $a|b$...).

Par exemple, étudions le langage décrit par $(a^2| b^*a^+)^*$.

Les automates reconnaissant $r_1=a^2$, $r_2=b^*$ et $r_3=a^+$ sont faciles à obtenir :



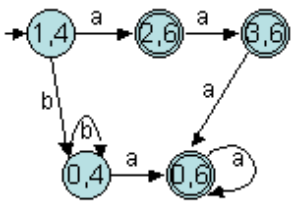
Ensuite, on construit l'automate produit reconnaissant $r_4=r_2r_3$:



Ensuite, nous obtenons r_0 par l'union de r_1 avec r_4 :

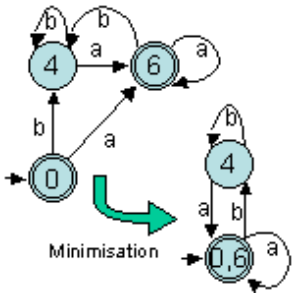
	a	b
1,4∈ I	2,6	0,4
2,6 ∈ F	3,6	0,0
0,4	0,6	0,4
3,6 ∈ F	0,6	0,0
0,6 ∈ F	0,6	0,0

D'où :



En minimisant (ou en réfléchissant dès le départ car $a^2 \subset b^*a^+$!) on obtient $r_0=r_4$!

Enfin, $r = r_4^*$ donne l'automate homogène suivant :



3.4. Suite d'expressions rationnelles

Nous venons de présenter des méthodes permettant de passer d'une expression rationnelle décrivant un langage à un automate fini (déterministe minimal) le reconnaissant. Cependant, il est souvent plus facile de définir un langage à l'aide d'un ensemble d'expressions plutôt qu'avec une seule et unique expression. Pour cela, il est possible de donner un ensemble de définitions rationnelles. Le langage décrit est alors l'union des langages définis par chacune des définitions.

Par exemple, supposons un langage L composé d'identificateurs et d'entiers, il y aura alors les deux définitions rationnelles (elles-mêmes basées sur deux classes de symboles) suivantes :

Ltr → A | ... | Z | a | ... | z
Ch → 0 | ... | 9
Id → **Ltr**(**Ltr** | **Ch**)*
Ent → **Ch** **Ch***

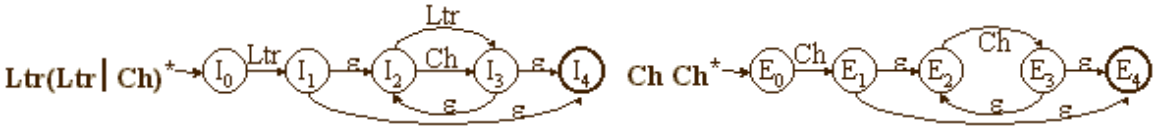
Nous allons maintenant présenter une méthode pour regrouper les différents automates associés à chacune des définitions.

Soit une suite de définitions rationnelles de la forme :

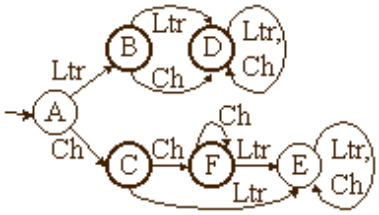
- $A_1 \rightarrow R_1$;
- $A_2 \rightarrow R_2$;
- ...
- $A_n \rightarrow R_n$.

La génération pour cet ensemble d'expressions rationnelles passera par les étapes suivantes (en figures l'application sur L) :

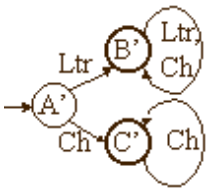
- La production des AFNs $M_i=(A_i, Q_i, I_i, F_i, \bullet_i)$ pour les expressions rationnelles R_i (algorithme de Thompson par exemple) avec des ensembles Q_i disjoints.



- La construction de l'AFN M=(A, Q, I, F, •), **union** des M_i. Cet AFN est donc obtenu par la mise en commun des différents AFNs.
- L'application de l'algorithme par construction des sous-ensembles sur M permettant de le rendre déterministe. On obtient M'=AFN2AFD(M).



- L'application de l'algorithme de minimisation sur M'.



Exercices et tests :

Exercice 3.1. Donner les automates finis reconnaissant les langages définis par les expressions rationnelles suivantes sur {a,b,c} selon les trois méthodes vues dans cette section (méthode intuitive, de Thompson et de Glushkov) :

- a+
- ab|c
- a(b|c)*

Exercice 3.2. Donner l'automate fini optimal reconnaissant le langage défini par l'expression rationnelle suivante sur {a, b} : (ab*)|(ab)*

4. Construction d'une expression rationnelle à partir d'un AFN

Le théorème de Kleene nous indique que les langages reconnaissables et les langages rationnels sont en réalité une seule et même classe de langages. La démonstration de ce théorème est une démonstration par induction. A la section précédente, nous avons aussi montré comment passer d'une expression rationnelle à un automate fini reconnaissant le même langage. En particulier, les deux premières méthodes (méthode intuitive et méthode de Thompson) sont proches de la démonstration par induction puisqu'il y a décomposition de l'expression et construction de l'automate en fonction des langages de base et des opérateurs utilisés. Ces méthodes permettent de prouver, qu'à une expression rationnelle, on peut faire correspondre un automate fini. Il est aussi possible de montrer la réciproque et d'en déduire des algorithmes effectuant cette opération.

Remarque : un intéret de cette transformation est de faciliter l'implémentation d'un traitement décrit par un automate fini. En effet, certains langages de programmation comme , , , ... possèdent des fonctions permettant de faire des traitements décrits par des expressions rationnelles (souvent appelées plutôt expressions régulières).

4.1. Méthode de McNaughton & Yamada

La méthode de McNaughton & Yamada permet de prouver qu'un langage reconnaissable est rationnel. Cette démonstration se base sur la notion de langage entre deux états.

Soit T={A,Q,I={i},F, μ} un automate fini standard. L(T) désigne le langage reconnu par T et l(p,q) désigne le langage entre les

états p et q. Alors, L(T) = $\cup_{q\in F}$ l(i,q).

L'objectif est de montrer que ∀, p,q ∈ Q, l(p,q) ∈ Rat(A*)
et donc aussi L(T).

Il faut commencer par ordonner totalement les états en les numérotant de 1 à n (avec |q|=n). Ceci ne pose pas de problèmes car la numérotation des états n'a aucune influence sur la reconnaissance du langage.

Soit $l^k(i,j)$ (∀ i,j∈ Q, ∀ k ∈ [0,n]) l'ensemble des chemins (mots) permettant de passer de i à j sans passer par les états s > k.

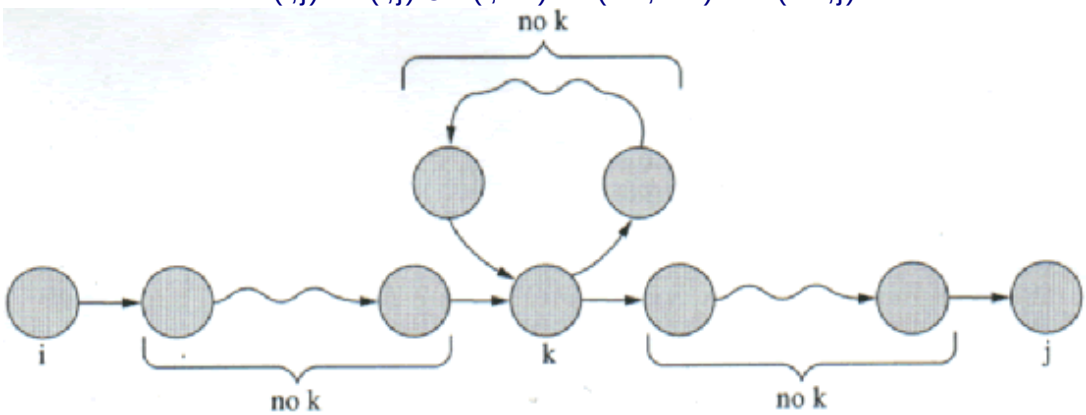
Alors

$$l(i,j) = \begin{cases} l^n(i,j) \text{ si } i \neq j \\ l^n(i,j) \cup \{\varepsilon\} \text{ si } i = j \end{cases}$$

Montrons que $l^k(i,j) \in Rat(A^*)$. Il suffit de procéder par induction :

- $l^0(i,j)$ correspond aux transitions directes entre i et j (sans passer par quelque autre état). Donc $l^0(i,j) \subseteq A$ d'où $l^0(i,j) \in Rat(A^*)$
- Supposons maintenant que $l^k(i,j) \in Rat(A^*)$ ∀ i,j ∈ Q, k<n
  - Alors $l^{k+1}(i,j)$ correspond à $l^k(i,j)$ auquel on ajoute le langage entre p et q en passant par l'état numéroté k+1. Ce langage est le produit du langage permettant d'aller de i à k+1, éventuellement le langage allant de k+1 à k+1 et le langage permettant d'aller de k+1 à j. Autrement dit :

$$l^{k+1}(i,j) = l^k(i,j) \cup l^k(i,k+1) \times l^k(k+1, k+1)^* \times l^k(k+1,j)$$



- Or, $l^k(i,j) \in Rat(A^*)$, $l^k(i,k+1) \in Rat(A^*)$, $l^k(k+1, k+1)$ et $l^k(k+1,j) \in Rat(A^*)$ (par hypothèse d'induction)
  Par définition, l'union, la mise à l'étoile ou le produit de langages rationnels est un langage rationnel.
- Donc $l^{k+1}(i,j) \in Rat(A^*)$ ∀ i,j∈ Q

Par conséquent comme

$$l(i,j) = \begin{cases} l^n(i,j) \text{ si } i \neq j \\ l^n(i,j) \cup \{\varepsilon\} \text{ si } i = j \end{cases}$$

$l(i,j) \in Rat(A^*)$ ∀ i,j∈ Q

De plus, comme L(T) = $\cup_{q\in F}$ l(i,q) et que l'union de langages rationnels est un langage rationnel alors $L(T) \in Rat(A^*)$

CQFD !

De plus, $l(i,j) \in Rat(A^*)$ donc l(i,j) peut être décrit par une expression rationnelle r_{ij}

Donc L(T) = $\cup_{q\in F}$ l(i,q) est décrit par r_{iq1} | ... |r_{iqn} avec $q_i \in F$

Il est donc possible de construire une expression rationnelle à partir d'un automate fini. Il suffit de construire les matrices successives des l^i (p en lignes et q en colonnes).

L'algorithme fonctionne de la manière suivante : après avoir renuméroté les états, on regarde les langages en transition directe entre les différents états de l'automate (l^0) puis les langages que l'on obtient en passant, en plus, par l'état 1 (l^1) puis ceux passant, en plus, par l'état 2 (l^2)... Lorsqu'il n'y a pas d'autres états à ajouter, s'est terminé ! Il suffit alors de prendre les langages entre un état initial et un état final. Lorsque ces deux états sont identiques, on ajoute alors ε.

Remarque : comme nous le verrons dans l'exemple suivant, cette méthode est extrêmement lourde à manipuler "à la main" (on préférera la méthode suivante). Par contre, elle est facile à implémenter.

Exemple :

$$T = \{\{a,b\},\{1,2\},\{1\},\{1\},\{\mu(a,1)=2,\mu(a,2)=1,\mu(b,2)=2\}\}$$

$$l^0 = \begin{pmatrix} \emptyset & a \\ a & b \end{pmatrix}$$

$$l^1 = \begin{pmatrix} \emptyset|\emptyset\emptyset^*\emptyset & a|\emptyset\emptyset^*a \\ a|a\emptyset^*\emptyset & b|a\emptyset^*a \end{pmatrix} = \begin{pmatrix} \emptyset & a \\ a & b|aa \end{pmatrix}$$

$$l^2 = \begin{pmatrix} \emptyset|a(b|aa)^*a & a|a(b|aa)^*(b|aa) \\ a|(b|aa)(b|aa)^*a & (b|aa)|(b|aa)(b|aa)^*(b|aa) \end{pmatrix}$$

$$= \begin{pmatrix} a(b|aa)^*a & a(b|aa)^* \\ (b|aa)^*a & (b|aa)^*(b|aa) \end{pmatrix}$$

$$\text{Donc } L(T) = l(1,1) = l^2(1,1) \cup \{\varepsilon\} = \varepsilon|a(b|aa)^*a$$

Et l'automate ne comporte que deux états !

Donc le langage reconnu par T est décrit par r_1=ε | a(b|aa)*a

4.2. Résolution d'un système d'équations régulières

Une autre approche pour prouver qu'un langage reconnaissable est rationnel est de passer par la notion de langage droit d'un état. Là encore, nous obtenons l'expression rationnelle du langage.

Soit T={A,Q,I,F, μ} un automate fini ε-libre. L(T) = $\cup_{i\in I}$ $LD_T(i)$.

Si $x \in LD_T(q)$ alors :

- $x = \varepsilon$ et $q \in F$
- $x = ay$, $a \in A$ et $y \in A^*$, $\mu(a,q) = p$ et $y \in LD_T(p)$

Donc $\forall p \in Q$, $LD_T(p) = \cup_{q \in Q} E_{p,q} LD_T(q) + \delta_p$

avec $E_{p,q}=\{a : a \in A, \mu(a,p)=q\} \in Rat(A^*)$ car fini (A est fini).

et $\delta_p=\varepsilon$ si $p \in F$, $\varnothing$ sinon

$\delta_p \in Rat(A^*)$ (par définition - cas de base - des langages rationnels)

En appliquant cette équation pour chacun des états de l'automate, nous obtenons alors un système de n équations rationnelles ($n=|Q|$) à n inconnues où les inconnues sont les $LD_T(p)$ (notée L_p) et les coefficients sont des expressions rationnelles. La résolution d'un tel système s'effectue en utilisant principalement le lemme d'Arden. Cette résolution permet d'obtenir pour chaque L_i une expression rationnelle décrivant $LD_T(i)$.

Le langage L(T) est alors : $L(T) = \cup_{i \in I} L_i$.

Reprenons l'exemple présenté pour la méthode précédente :

T = {{a,b}, {1,2}, {1}, {1}, {μ(a,1)=2, μ(a,2)=1, μ(b,2)=2}}.

La construction du système d'équations associé donne :

$L_1 = aL_2 + 1$ (car l'état 1 est final)
$L_2 = bL_2 + aL_1$

Qui se résoud de la manière suivante :

$L_2 = b^*aL_1$ (grâce au lemme d'Arden)
$L_1 = ab^*aL_1 + 1 = (ab^*a)^*$ (grâce au lemme d'Arden)

L'expression rationnelle décrivant le langage est L_1 car l'état 1 est le seul état initial. Donc L(T) est décrit par $r_2=(ab^*a)^*$.

Notons que pour un même automate (T), McNaughton&Yamada ($r_1 = \varepsilon \mid a(b|aa)^*a$) et la résolution de système d'équations ($r_2 = (ab^*a)^*$) ne donnent pas le même résultat mais ces deux expressions rationnelles décrivent bien le même langage (preuve possible).

Remarque : avant de calculer le système d'équation associé à un automate, il est conseillé de l'émonder. En effet, tout état en moins réduit le système d'une équation et d'une inconnue sans pour autant changer le langage. Dans le cas où l'automate n'est pas émondé, il faut par exemple faire attention aux états stériles.

Prenons l'exemple suivant :

T = {{a,b}, {1,2,3,4}, {1}, {1}, {μ(a,1)=2, μ(b,1)=4, μ(b,2)=1, μ(a,2)=3, μ(b,3)=3}}.

La construction du système d'équations associé donne :

$L_1 = aL_2 + bL_4 + 1$
$L_2 = bL_1 + aL_3$

$L_3 = bL_3 + \varnothing$
$L_4 = \varnothing$

d'où $L_3 = b^*\varnothing = \varnothing$

et donc : $L_2 = bL_1$ et $L_1 = aL_2 + 1$

$L_1 = abL_1 + 1 = (ab)^*$

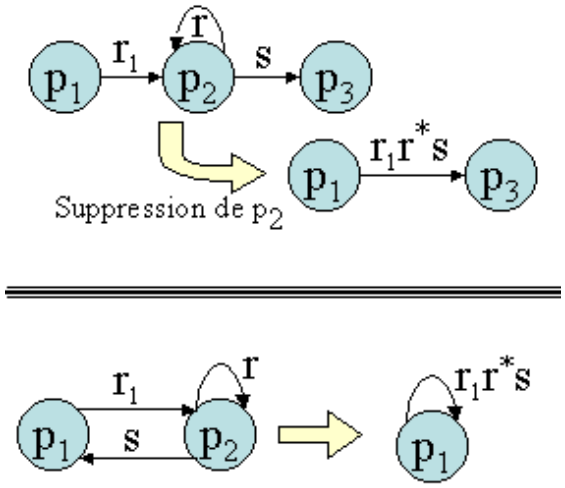
4.3. Réduction d'automates

Pour terminer, nous allons présenter une méthode plus "graphique" de transformation d'un AFN en une expression rationnelle. Cette méthode utilise un graphe (appelé parfois "automate étendu") identique à l'AFN au départ mais qui tout au long du traitement pourra comporter des expressions rationnelles comme étiquette sur les arcs (ex-transitions).

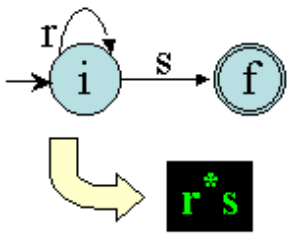
L'automate d'origine sera supposé unitaire mais pas forcément déterministe ou ε-libre.

Pour T={A,Q,{i},F, μ}, la méthode est la suivante :

- Ajouter un état final f connecté aux anciens états finaux par des ε-transitions (T'={A,Q∪{f},{i},{f}, μ'} avec μ' = μ ∪ {(p, ε, f) : si p ∈ F}) ;
- Effacer tous les autres états sauf l'état initial sans changer de langage en appliquant les règles suivantes :

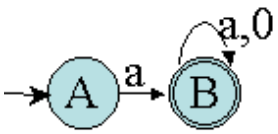


- Pour conclure, lorsqu'il ne reste plus que i et f alors on applique la règle suivante :

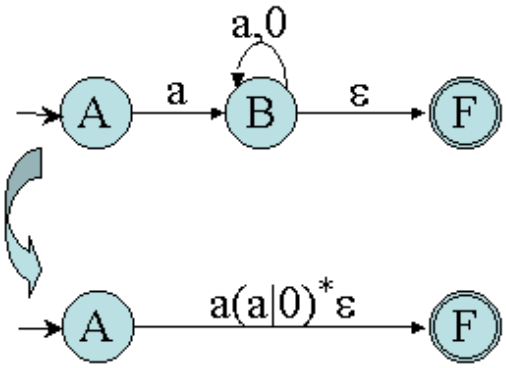


Lorsque plusieurs transitions entrent et sortent d'un état, alors pour le supprimer, il faut considérer toutes les combinaisons à trois états comprenant une transition entrante et une transition sortante.

Par exemple, soit l'automate suivant :

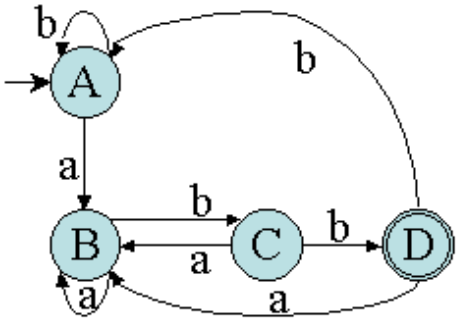


Nous obtenons alors les transformations suivantes :

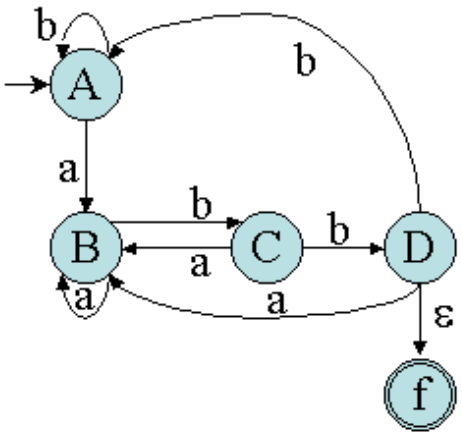


Nous pouvons en conclure que l'automate reconnaît le langage décrit par l'expression : a(a|0)*.

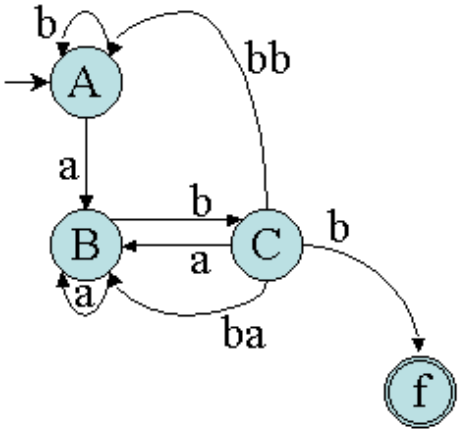
Prenons un autre exemple :

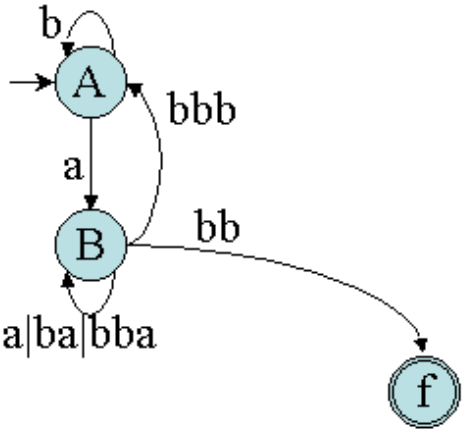


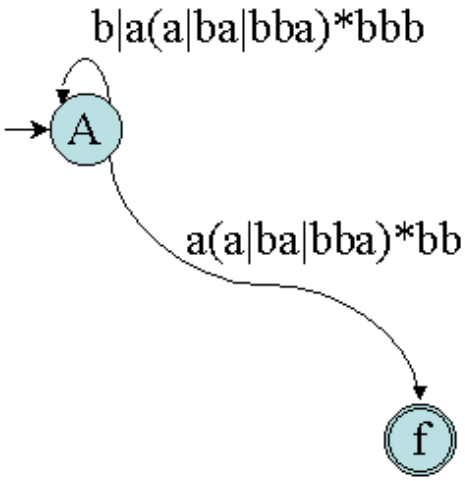
Nous obtenons alors en ajoutant un état final :



Puis nous obtenons les simplifications suivantes :





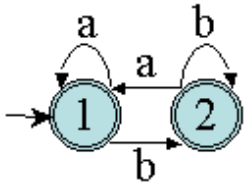


D'où l'expression suivante : (b|a(a|ba|bba)*bbb)*a(a|ba|bba)*bb

Noter que l'expression qui a été utilisée pour construire l'automate de départ était : (a|b)*abb !

Exercices et tests :

Exercice 4.1. Soit l'automate suivant sur l'alphabet {a,b} :



1. Donner l'expression rationnelle décrivant cet automate en utilisant la méthode de McNaughton&Yamada.
2. Même chose en utilisant la méthode par résolution d'un système d'équations rationnelles.
3. Même chose en utilisant la méthode par réduction d'automates.
4. (facultative) Si les expressions sont différentes, montrer qu'elles décrivent le même langage.

Exercice 4.2. Déterminer le langage reconnu par l'automate T={{a,b}, {1,2,3}, {1}, {1}, {μ(a,1)=1, μ(b,1)=2, μ(a,2)=3, μ(b,2)=1, μ(a,3)=2, μ(b,3)=3}} par la méthode de résolution de systèmes d'équations rationnelles.

Exercice 4.3. Soit l'automate fini suivant : T={{a,b,c}, {1,2,3,4}, {1,3}, {2,4}, {μ(b,1)=2, μ(b,1)=3, μ(a,2)=2, μ(a,2)=4, μ(c,3)=4, μ(c,4)=4}}.

1. Donner le système d'équations rationnelles issu de T
2. Résoudre ce système et en déduire l'expression rationnelle reconnaissant L(T)
3. Rendre cet automate déterministe et minimal si nécessaire en utilisant les méthodes de construction des sous-ensemble et de Moore. Soit T' ce nouvel automate.
4. Donner le nouveau système d'équations rationnelles issu de T'.
5. Résoudre ce dernier système et en déduire l'expression rationnelle reconnaissant L(T').

5. Grammaires, AFN et expressions rationnelles

Nous avons déjà montré la correspondance entre une grammaire linéaire droite et un langage rationnel par l'intermédiaire d'un système d'équations ratrionnelles. Il nous reste à faire la correspondance entre un AFN et une grammaire linéaire (et c'est possible puisque tous deux décrivent des langages rationnels !)

5.1. AFN et grammaires linéaires droites

La méthode est assez proche de celle utilisée pour obtenir une expression rationnelle à partir d'un AFN par un système d'équations rationnelles. Ce n'est pas un hasard puisque pour obtenir l'expression associé à une grammaire, on passe aussi par un système d'équations.

Soit T un AFN unitaire : T = {A, Q, I={i}, F, μ}. Alors, il est possible de construire une grammaire linéaire droite G={V_T, V_N, S, R} :

- $V_T = A$
- $V_N = \{A_i : A_i \equiv q_i \in Q\} \cup \{S\}$
- $S \rightarrow A_j \in R \Leftrightarrow A_j \equiv i$
- $\mu(a,q_i)=q_j \in \mu \Leftrightarrow A_i \rightarrow aA_j \in R$
  $q_i \in F \Leftrightarrow A_i \rightarrow \varepsilon \in R$

5.2. Exemple

Reprenons l'exemple présenté dans la section précédente pour le passage de l'AFN à l'expression en utilisant le système d'équations :

T = {{a,b}, {1,2}, {1}, {1}, {μ(a,1)=2, μ(a,2)=1, μ(b,2)=2}}.

Si l'on applique la méthode vue ci-dessus, nous obtenons :

G = {{a,b}, {S, A_1, A_2}, S, R} avec R tel que :

- $S \rightarrow A_1$
- $A_1 \rightarrow aA_2 \mid \varepsilon$
- $A_2 \rightarrow aA_1 \mid bA_2$

Pour vérifier, transformons cette grammaire en un système d'équations :

$X = X_1$
$X_1 = aX_2+1$
$X_2 = aX_1 + bX_2$

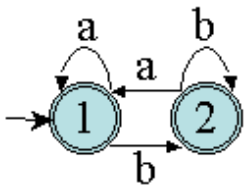
Donc :

$X_2 = b^*aX_1$
$X_1 = ab^*aX_1 + 1 = (ab^*a)^*$
$X = (ab^*a)^*$

Donc L = X = $(ab^*a)^*$ ce qui est la même expression (heureusement !)

Exercices et tests :

Exercice 5.1. Donner la grammaire linéaire droite décrivant l'automate suivant sur l'alphabet {a,b} :

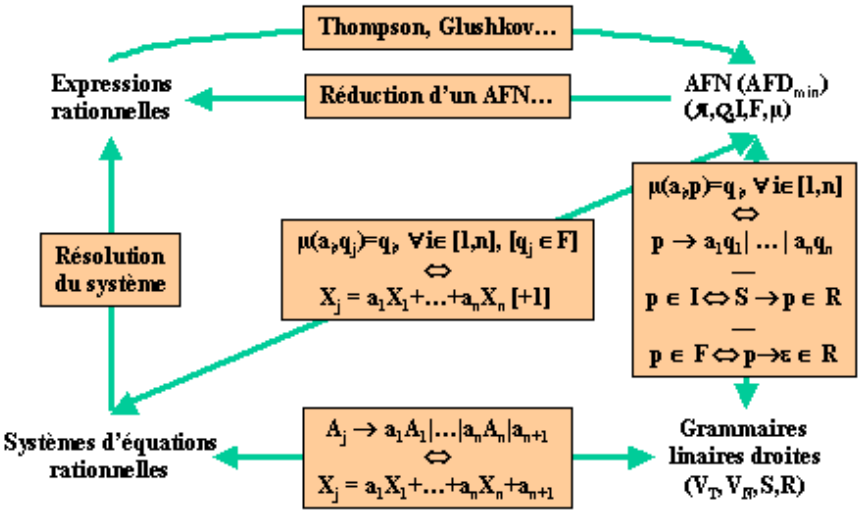


Exercice 5.2. Déterminer la grammaire linéaire droite décrivant le langage reconnu par l'automate T={{a,b}, {1,2,3}, {1}, {1}, {μ(a,1)=1, μ(b,1)=2, μ(a,2)=3, μ(b,2)=1, μ(a,3)=2, μ(b,3)=3}}.

6. Synthèse

Finalement, au cours de ces chapitres, nous avons présenté le passage entre différentes représentations d'une même entité : **les langages rationnels**.

La figure suivante présente ces relations.



Applications des langages formels et des automates finis

1. Introduction

Dans les chapitres précédents, nous avons présenté une introduction à la théorie des langages et des automates. Ces notions ne sont pas seulement des éléments théoriques abstraits mais sont aussi des outils performants trouvant leur utilité dans de nombreux de domaines. Nous allons présenter ici une utilisation dans le domaine du traitement de langages à syntaxe contrainte (langages de programmation) ou souple (langage naturel). Dans un premier temps, nous étudierons la gestion des langages, et surtout l'utilisation des automates dans le cadre du processus de transformation d'un programme formulé dans un langage de programmation donné vers un langage compréhensible par une machine : c'est le processus de compilation. Puis, nous donnerons quelques éléments liés à l'analyse du langage naturel. Enfin, nous terminerons par la présentation d'autres domaines d'application.

2. La compilation

La compilation est l'étude de la traduction de programmes écrits dans un langage de programmation (C, PASCAL, FORTRAN...), en des programmes équivalents écrits dans un langage de plus bas niveau (machine ou cible).

2.1. Éléments principaux

Le langage

Un langage est formé d'un ensemble de phrases qui permet de désigner des objets d'un certain domaine sémantique ou univers et des actions portant sur ces objets.

Une phrase est un assemblage de mots appelés symboles dont l'ensemble constitue le vocabulaire du langage. Chaque mot ou symbole résulte de la concaténation d'une suite finie de signes ou caractères. Un symbole est la plus petite sous-séquence de signes chargée de sens que l'on peut classer suivant trois catégories :

1. ceux qui désignent des objets du domaine sémantique ;
2. ceux qui désignent des relations ou des applications sur le domaine sémantique ;
3. ceux qui servent pour la ponctuation ou la coordination de parties de phrases.

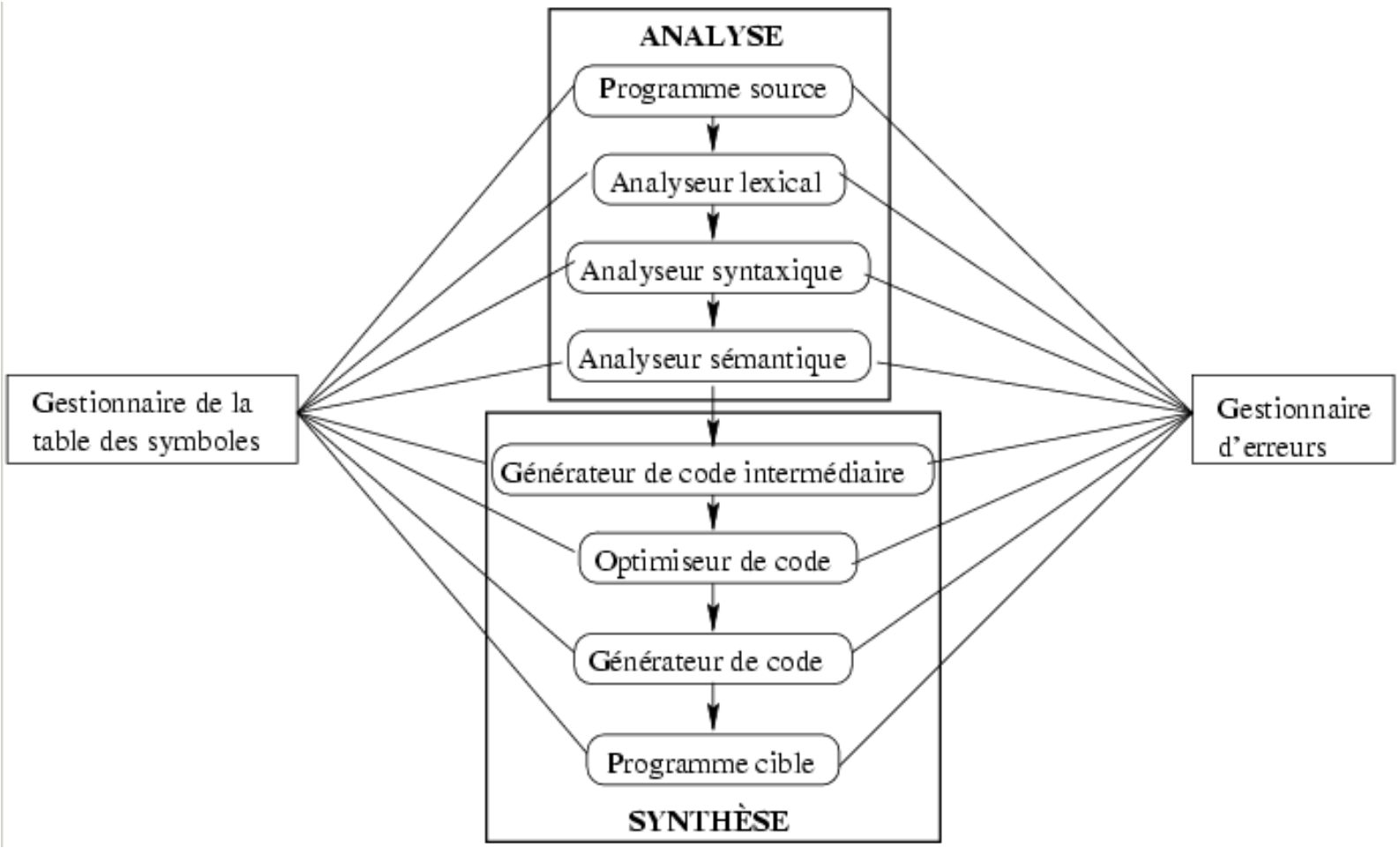
Dans le cadre de l'informatique, de nombreux logiciels admettent en entrée des langages soumis à des règles d'écritures précises appelées règles syntaxiques.

Par exemple pour un SGBD, le langage d'interrogation d'une base de données permet de gérer les éléments de la base. Le logiciel doit contrôler la validité des commandes d'interrogation de la base (analyse syntaxique) puis est amené à exécuter des traitements spécifiques, ou actions, qui entraînent la recherche dans la base d'informations pour construire la réponse.

Dans ce cas, l'analyseur syntaxique dirige ou "pilote" le logiciel et on parle de programmation dirigée par la syntaxe. C'est une méthode de programmation en tant que telle.

Le compilateur

De façon générale, un compilateur est un programme qui lit en entrée un programme écrit dans un langage appelé <<langage source>>. Puis, il le traduit dans un programme équivalent dans un langage différent appelé langage cible. La compilation d'un programme se déroule en deux parties successives : l'analyse et la synthèse .



L'analyse.

Elle permet de trouver la structure du programme à partir du texte du programme, c'est-à-dire d'une suite de caractères. L'analyse est uniquement dépendante du langage à traduire (indépendante du langage cible). Nous distinguons les trois phases d'analyse suivantes :

- l'analyse lexicale qui lit la chaîne de caractères d'entrée, caractère par caractère, et la transforme en une suite de mots ou unités lexicales ;
- l'analyse syntaxique qui vérifie si la suite d'unités lexicales reconnues par l'analyseur lexical est légale. L'analyse syntaxique dérive une structure hiérarchique sur le flot d'unités lexicales qui peut être représentée par des arbres abstraits ;
- l'analyse sémantique qui contrôle si le programme source a un sens. L'un des rôles importants de l'analyse sémantique est, en particulier, le contrôle de type : on vérifie si les opérandes de chaque opérateur sont conformes aux spécifications du langage source (opérations arithmétiques entre nombres entiers, ou réels...).

Au cours de ce processus d'analyse, un rôle important du compilateur est de signaler à son utilisateur, la présence d'erreurs dans le programme source via des messages d'erreurs. Par exemple, les erreurs peuvent être :

- lexicales, comme l'écriture erronée d'un identificateur, d'un mot clé ou d'un opérateur ;
- syntaxiques, comme une expression mathématique mal parenthésée ;
- sémantiques, comme un opérateur appliqué à un opérande incompatible ;
- logiques, comme un appel récursif infini.

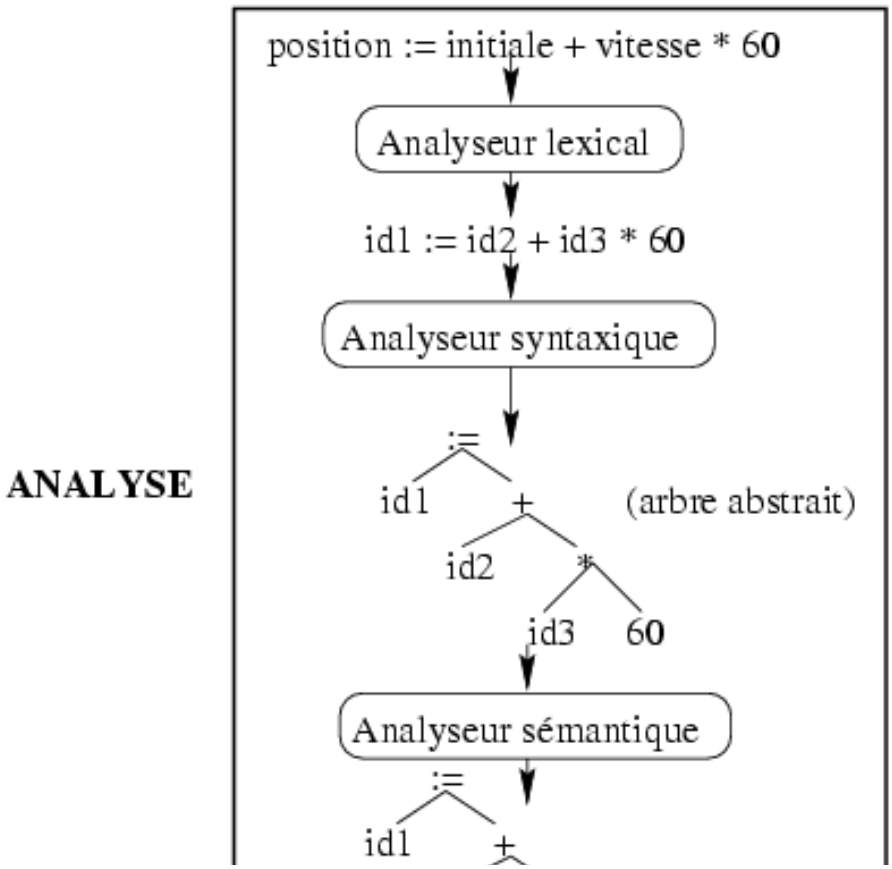
La synthèse.

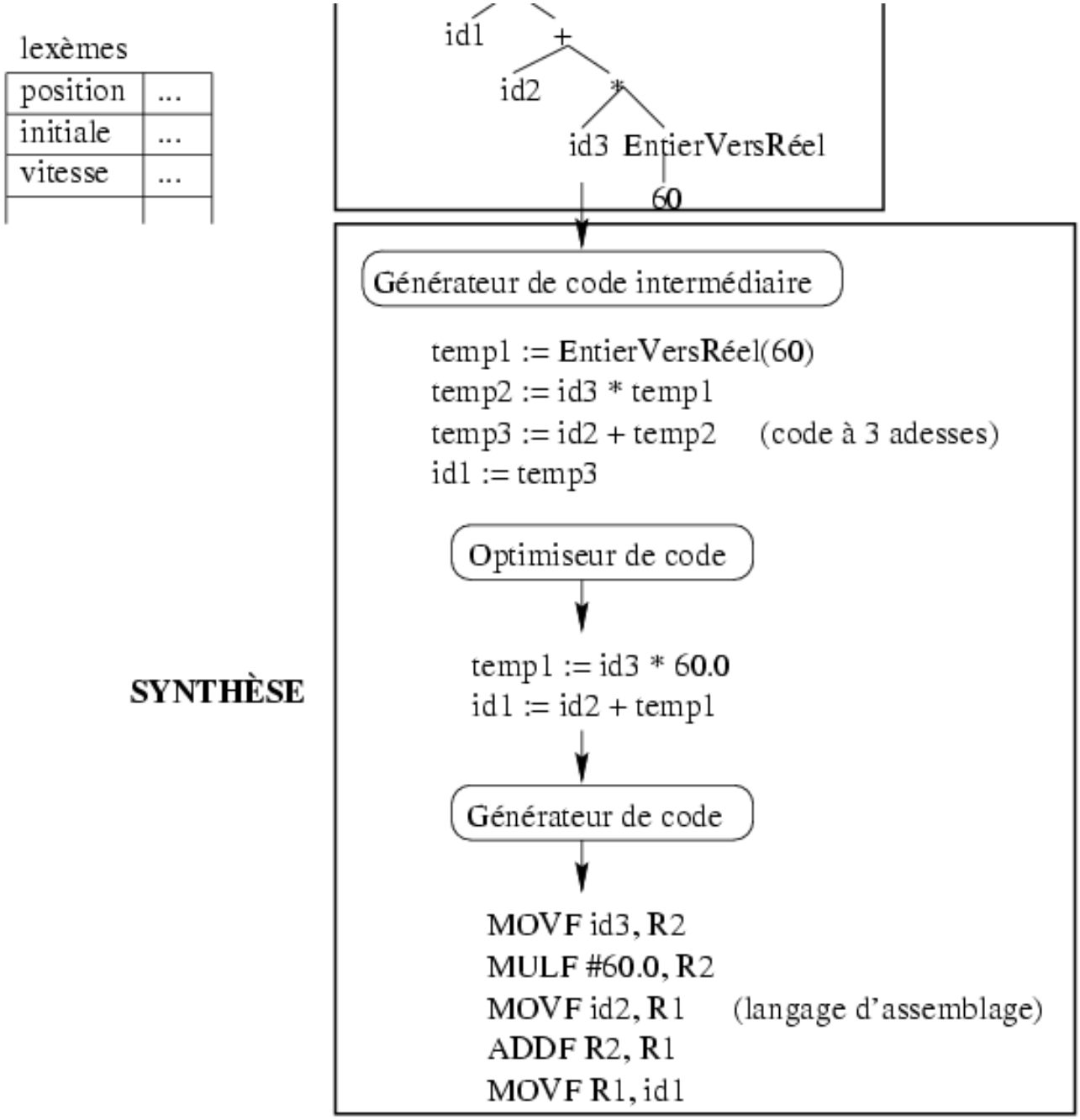
Après l'analyse du programme source vient la partie synthèse. Elle permet de traduire la structure du programme en prenant en compte les possibilités du langage cible (optimisation). Elle dépend du langage cible (choix des constructions machines, ressources disponibles...) et de la définition du langage compilé (c'est-à-dire du sens associé aux structures du programme : la sémantique du langage).

Elle peut se décomposer selon les phases suivantes :

- production de code intermédiaire : certains compilateurs construisent explicitement une représentation intermédiaire qui peut être considérée comme un programme pour une machine abstraite (machine virtuelle) ;
- optimisation de code : on tente ici d'améliorer le code intermédiaire de telle sorte que le code machine s'exécute le plus rapidement possible ;
- production du code cible qui est un code machine translatable ou un code d'assemblage. On cherche en particulier ici à gérer au mieux l'assignation des variables aux registres de la machine.

Voici un exemple du processus de compilation d'une instruction d'affectation et de calcul : "position := initiale + vitesse * 60;"





Par exemple : Pascal (vers du P-CODE ou du langage machine), C, Ada... sont des langages compilés.

L'interpréteur

La production d'un code machine destiné à être exécuté par le système procure une grande rapidité d'exécution parce qu'il peut être optimisé en exploitant au mieux les caractéristiques de l'architecture de la machine sur lequel il s'exécute. Mais cette approche révèle certains inconvénients :

- faible degré de portabilité : le code machine peut varier d'un type d'ordinateur à un autre ;
- contrôle assuré directement par le système à l'exécution (peu d'interactivité) ;
- difficulté de mise au point.

Au lieu de produire un programme cible comme dans une technique de traduction normale, l'interprète effectue lui-même les opérations spécifiées par le programme source. Il exécute (interprète) un code intermédiaire qui est indépendant d'un code machine donné. L'interprète peut être écrit dans un langage

évolué suffisamment universel pour que le changement de type d'ordinateur ne pose pas de trop gros problèmes. Si l'inconvénient majeur de l'interprète est sa lenteur, ses avantages sont sa portabilité et sa souplesse à l'exécution. En effet, l'utilisateur peut plus facilement intervenir durant l'exécution pour modifier son cours.

Par exemple : Basic, Lisp, Caml, Clips, Perl, Prolog, P-CODE... sont des langages interprétés.

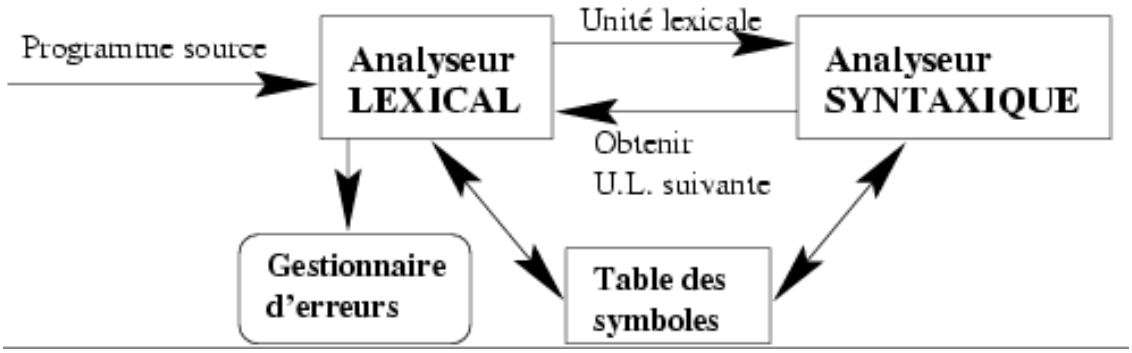
Le compilateur Java génère du **byte-code** (indépendant de toute architecture) et non du code machine. Pour vraiment exécuter un programme Java, il faut utiliser un interpréteur qui exécute le byte-code généré. Un programme Java peut être exécuté sur tout système implémentant l'interpréteur Java et le runtime Java. Cet interpréteur et ce runtime implémentent à eux deux la **machine virtuelle Java** (ou JVM pour "Java Virtual Machine"). De ce fait, l'exécution d'un programme en Java est 20 fois plus lente qu'un programme équivalent en C. Cependant, la compilation en byte-code permet d'atteindre des temps d'exécution bien supérieurs aux programmes écrits en langages de scripts comme Tcl, Perl ou les shells UNIX.

Néanmoins, il existe des moyens de donner aux programmes en Java des performances très proches de celles d'un programme en C. Outre l'apparition de processeurs Java, il existe des compilateurs "just in time" (ie. "qui compilent au dernier moment") capables de compiler le byte-code Java en code machine dédié à un processeur particulier et ce en cours d'exécution. Cette génération de code machine est relativement simple à réaliser (prévue dès la conception du byte-code) et produit un code raisonnablement efficace. Ces compilateurs permettent de générer du code machine soit durant la génération du programme soit au moment de l'exécution. Dans ce dernier cas, la compilation s'effectue au premier passage dans le code en parallèle de la première exécution. Les passages suivants se font directement sur le code machine.

2.2. L'analyse lexicale

Principe

L'analyse lexicale ou lexicographique permet de déterminer les unités lexicales qui ont un sens. Cette analyse se base sur une grammaire. A l'aide de celle-ci, et donc du langage engendré, il est possible de mettre en place une procédure d'analyse d'une chaîne donnée. Sachant que la plupart des grammaires utilisées en analyse lexicale sont régulières, il est possible d'utiliser un automate fini pour cette analyse.



L'analyse lexicale est donc la reconnaissance d'un langage régulier (le plus souvent) et le codage des mots. Elle permet de découper le texte d'entrée en une suite de symboles.

Cette analyse utilise des expressions régulières, définissant les symboles (lexèmes) du langage, converties en un AFD minimal unique et un ensemble de couples : type de symbole / état final. L'analyse lexicale fait

tourner l'AFD jusqu'à un état final et que le prochain caractère mène à une erreur.

Remarque : L'analyse lexicale fonctionne selon le principe de la plus grande correspondance : l'analyseur cherche à reconnaître la plus longue chaîne.

Gestion de la table des symboles

Les unités lexicales reconnues sont stockées dans la table des symboles. Cette table indique la classe lexicale (rôle de l'unité lexicale dans la phrase) et la catégorie du symbole (Par exemple, la table suivante).

Catégorie du symbole	Classe lexicale
mot clé	classe de ce mot clé
nombre	C^{ste}
chaîne	C^{ste}
autre mot	identificateur

Si le symbole est présent dans la table, celle-ci fournit son code lexical. Sinon, la classe de ce symbole est calculée. Son numéro est alors l'indice de la place libre suivante dans la table et l'information sur le symbole est rangée à cette place.

La gestion de cette table est un élément important de la phase d'analyse du programme. A l'accès associatif, on préfère souvent une gestion plus optimisée avec un accès direct. En pratique, on utilise souvent le "hashcoding".

Remarque : les mots clés du langage sont pré-enregistrés dans la table avec leur classe lexicale.

Traitement des erreurs

Les principales erreurs soulevées sont la rencontre d'un caractère non-autorisé ou le débordement de la table des symboles. Dans le premier cas, il suffit de "passer la main" à l'analyseur syntaxique avec "autre" comme classe de symboles. Dans le second cas, l'unique solution consiste à arrêter le programme (erreur fatale).

2.3. LEX & YACC

LEX est un outil pour construire des analyseurs lexicaux à partir de notations basées sur les expressions régulières. Il est souvent associé à YACC, un autre utilitaire Unix, basé aussi sur le langage C, pour produire des compilateurs. Lex sert à effectuer la phase d'analyse lexicale de la compilation, Yacc permet de réaliser la phase d'analyse syntaxique. Les autres phases de compilations sont définies en C.

La collaboration avec l'analyseur syntaxique s'effectue selon l'algorithme déjà présenté.

2.4. L'analyse syntaxique et limites des AEF

Il est possible de mettre en place une analyse syntaxique basée sur des grammaires régulières (donc

utilisant un AFD). Les caractères du vocabulaire sont alors les symboles du langage. Sur les transitions, on associe alors des actions permettant de réagir à la présence d'un symbole (stockage de valeurs, affichage de résultats, calculs...).

Cependant, les AFDs (et les grammaires régulières) sont très souvent inadaptés pour représenter les langages de haut niveau non réguliers. Ces langages sont souvent des langages hors-contexte.

Le moyen le plus adapté pour de tels langages est l'automate à pile permettant d'implanter une analyse descendante (mais il existe aussi d'autres méthodes d'analyse utilisant d'autres algorithmes) et permettant de provoquer des actions après seulement une certaine série de symboles (problème du parenthésage par exemple ainsi que des opérateurs binaires et des expressions arithmétiques).

Ces automates sont déterministes seulement pour des grammaires hors-contexte de type LL(1). Nous n'aborderons pas cela dans le cadre de ce cours mais ce sera étudié en cours de compilation.

3. Le langage naturel [Per95]

Dans l'article fondateur de Shannon (C. E. Shannon, "A Mathematical Theory of Communication", Bell Systems Tech. Jour., XXVII:379-423, 623-656, 1948.), on trouve, comme nous l'avons vu, un modèle très voisin des automates finis, qui est directement issu des chaînes de Markov. Ce modèle est appliqué en particulier à ce que Shannon appelle des **approximations** du langage naturel. Il s'agit de chaînes de Markov d'ordre croissant réalisant un automate local pour la structure de la langue au niveau lexical (lettre par lettre ou mot par mot). Ainsi, en prenant successivement au hasard des digrammes chevauchant dans un texte en anglais, on obtient l'exemple saisissant :

ON IE ANTSOUTINYS ARE T INCTORE ST BE S DEAMY ACHIN D ILONASIVE TUCOOWE AT TEASONARE
FUSO TIZIN ANDY TOBE SEACE CTISBE

dans lequel certains mots existent réellement ('on, are, be... ') et d'autres le mériteraient ('deamy, ilonasive...'). Le résultat obtenu avec des trigrammes au lieu de digrammes est encore plus étonnant :

IN NO IST LAT WHEY CRACTIC FROURE BIRS GROCID PONDENOME OF DEMONSTRURES OF THE REPTAGIN IS REGOACTIONA OF CRE

Shannon présente d'autres exemples utilisant des chaînes de Markov au niveau des mots au lieu des lettres. En fait, dès le départ, A. A. Markov a introduit son modèle dans cette perspective et son article, daté de 1913, s'intitule ``Essai d'une recherche statistique sur le texte du roman Eugène Onegine'' (Bull. Acad. Imper. Sci. St Petersbourg, 7, 1913.).

L'idée d'utiliser des automates pour décrire la langue naturelle est reprise un peu plus tard par Chomsky [Cho56]. En fait, Chomsky n'introduit les automates que pour les écarter assez rapidement au profit du niveau supérieur de ce que l'on nomme aujourd'hui la **hiérarchie de Chomsky** : automates finis au rez-de-chaussée, automates à pile et grammaires context-free au premier étage, grammaires context-sensitives au second, machines de Turing au dernier. Les arguments employés par Chomsky pour écarter les automates finis comme modèle adéquat des langues naturelles sont fondés sur la présence de **structures parenthésées** provenant de constructions grammaticales comme les propositions conditionnelles "si S_1 alors S_2" analogues à celles des langages de programmation et qui permettent, en principe, une imbrication non bornée. De nombreux modèles autres que les automates ont été proposés pour rendre compte des

structures syntaxiques y compris les grammaires de Lambek ou les grammaires transformationelles de Z. Harris.

Le fait que l'anglais ne soit pas un langage rationnel semble aujourd'hui être, à la fois, non prouvable et en balance avec le fait que, de toutes façons, une description complète de la syntaxe de l'anglais (ou du français) est aussi difficile à réaliser sous forme d'automate fini que de grammaire. Les automates finis ayant par contre l'avantage sur les grammaires de se prêter à un grand nombre de manipulations automatisables. Certains, comme Maurice Gross, pensent que ce sont finalement les automates finis qui fournissent l'essentiel de la description (voir M. Gross et D. Perrin, "Electronic Dictionnaries and Automata in Computational Linguistics", Lecture Notes in Computer Science, 377. Springer, 1989.). Du point de vue des automates, la différence essentielle avec les grammaires est le fait que la théorie logique associée aux automates (la logique monadique de Büchi) est décidable alors que la plupart des problèmes associés aux grammaires ne le sont pas. Si par exemple on peut assez facilement déterminer si deux automates finis sont équivalents, il n'en est pas de même des grammaires : on peut démontrer que le problème de savoir si deux grammaires sont équivalentes est indécidable.

4. Autres applications

Nous venons de voir que langage et automates sont utilisés pour générer des programmes et pour comprendre le langage naturel. Cependant, ce ne sont certainement pas les seules utilisations possibles.

Les automates sont aussi souvent utilisés pour modéliser des phénomènes naturels liés au comportement. En particuliers, ils permettent de modéliser des connaissances complexes ainsi que des règles de comportement dépendant d'événement <<extérieurs>>. Pour cela, il est possible d'utiliser des **réseaux de Pétri**. En particulier, les **modèles de comportement** en animation se basent sur des **graphes d'événements**.

Dans certains cas, le comportement est aussi lié à des paramètres stochastiques. Aussi, lorsque l'automate n'est pas déterministe, il est parfois associé aux transitions sur un même symbole (ou événement) un paramètre probabiliste : ce sont les **automates stockastiques** ou **chaînes de Markov**. Ces automates permettent, en particulier, de modéliser la gestion de files d'attentes...

Les automates sont aussi utilisés en modélisation et conception de systèmes. Une modélisation OMT ou UML consiste non seulement en une description des objets mis en jeu mais aussi leurs relations vis-à-vis des divers évènements.

Toujours lié au comportement de systèmes, on trouve des applications des automates pour la gestion du dialogue Homme-Machine et pour la conception de jeux.