

# Théorie des Langages Rationnels

Notes de cours  
Travaux Dirigés et Travaux Pratiques  
Annales

François Yvon et Akim Demaille  
avec la participation  
d'Alexandre Duret-Lutz,  
Alexandre Hamez et  
Pierre Senellart

26 Septembre 2016  
(rev. 427418d)

---

### **Avertissement au lecteur**

Ces notes documentent le cours de théorie des langages rationnels enseigné en EPITA Info Spé. Elles sont, malgré nos efforts (et ceux de plusieurs générations d'étudiants qui nous ont aidé à l'améliorer, en dernier lieu A. Amarilli), encore largement perfectibles et probablement non-exemptes d'erreurs. Merci de bien vouloir signaler toute erreur, de syntaxe (un comble!) ou autre, aux auteurs ([yvon@limsi.fr](mailto:yvon@limsi.fr), [akim@lrde.epita.fr](mailto:akim@lrde.epita.fr), [pierre@senellart.com](mailto:pierre@senellart.com)).

La page web de THLR est <http://epita.lrde.epita.fr/THLR>. On trouvera dans <http://www.lrde.epita.fr/~akim/thlr/> les versions les plus récentes de ce polycopié, les sujets de TDS, TPS et projet.

# Chapitre 1

## Table des matières

<b>1</b>	<b>Table des matières</b>	<b>3</b>
<b>I</b>	<b>Notes de Cours</b>	<b>7</b>
<b>2</b>	<b>Mots, Langages</b>	<b>9</b>
2.1	Quelques langages « réels »	9
2.1.1	La compilation	9
2.1.2	Bio-informatique	10
2.1.3	Les langues « naturelles »	11
2.2	Terminologie	11
2.2.1	Bases	11
2.2.2	Quelques notions de calculabilité	12
2.3	Opérations sur les mots	13
2.3.1	Facteurs et sous-mots	13
2.3.2	Quotient de mots	14
2.3.3	Ordres sur les mots	14
2.3.4	Distances entre mots	15
2.3.5	Quelques résultats combinatoires élémentaires	17
2.4	Opérations sur les langages	18
2.4.1	Opérations ensemblistes	18
2.4.2	Concaténation, Étoile de Kleene	19

---

2.4.3	Plus d'opérations dans $\mathcal{P}(\Sigma^*)$ . . . . .	19
2.4.4	Morphismes . . . . .	20
<b>3</b>	<b>Langages et expressions rationnels</b>	<b>21</b>
3.1	Rationalité . . . . .	22
3.1.1	Langages rationnels . . . . .	22
3.1.2	Expressions rationnelles . . . . .	22
3.1.3	Équivalence et réductions . . . . .	24
3.2	Extensions notationnelles . . . . .	25
<b>4</b>	<b>Automates finis</b>	<b>29</b>
4.1	Automates finis . . . . .	29
4.1.1	Bases . . . . .	29
4.1.2	Spécification partielle . . . . .	33
4.1.3	États utiles . . . . .	35
4.1.4	Automates non-déterministes . . . . .	35
4.1.5	Transitions spontanées . . . . .	40
4.2	Reconnaissables . . . . .	43
4.2.1	Opérations sur les reconnaissables . . . . .	43
4.2.2	Reconnaissables et rationnels . . . . .	46
4.3	Quelques propriétés des langages reconnaissables . . . . .	51
4.3.1	Lemme de pompage . . . . .	51
4.3.2	Quelques conséquences . . . . .	52
4.4	L'automate canonique . . . . .	53
4.4.1	Une nouvelle caractérisation des reconnaissables . . . . .	53
4.4.2	Automate canonique . . . . .	55
4.4.3	Minimisation . . . . .	56
<b>II</b>	<b>TD, DM, TP, Annales</b>	<b>61</b>
<b>5</b>	<b>Travaux Dirigés</b>	<b>63</b>
1	Preuves, calculabilité et distances . . . . .	64

---

2 Expressions rationnelles . . . . .	66
3 Automates finis . . . . .	68
4 Lemme de pompage et déterminisation . . . . .	70
5 Stabilité des langages rationnels . . . . .	72
<b>6 Devoirs à la Maison</b>	<b>75</b>
1 Langages . . . . .	76
2 Expressions rationnelles . . . . .	78
3 Automates . . . . .	80
4 Automates . . . . .	81
<b>7 Travaux Pratiques</b>	<b>85</b>
1 Expressions Rationnelles . . . . .	86
2 Vcsn – 1 . . . . .	89
3 Vcsn – 2 . . . . .	92
<b>8 Annales</b>	<b>95</b>
Contrôle 2007-2008 . . . . .	96
Contrôle 2008-2009 . . . . .	98
Contrôle 2009-2010 . . . . .	101
Contrôle 2010-2011 . . . . .	104
Contrôle 2011-2012 . . . . .	107
Contrôle 2012-2013 . . . . .	110
 <b>III Annexes</b>	 <b>115</b>
<b>9 Compléments historiques</b>	<b>117</b>
9.1 Brèves biographies . . . . .	117
9.2 Pour aller plus loin . . . . .	122
<b>10 Correction des exercices</b>	<b>125</b>
10.1 Correction de l'exercice 4.5 . . . . .	125
10.2 Correction de l'exercice 4.10 . . . . .	126

*TABLE DES MATIÈRES*

---

10.3 Correction de l'exercice 4.14 . . . . .	126
10.4 Correction de l'exercice 4.15 . . . . .	129
10.5 Correction de l'exercice 4.19 . . . . .	129
10.6 Correction de l'exercice 4.20 . . . . .	131
10.7 Correction de l'exercice 4.35 . . . . .	131
<b>IV Références</b>	<b>135</b>
<b>11 Liste des automates</b>	<b>139</b>
<b>12 Liste des tableaux</b>	<b>141</b>
<b>13 Bibliographie</b>	<b>143</b>
<b>14 Index</b>	<b>145</b>

**Première partie**

**Notes de Cours**

---





## Chapitre 2

# Mots, Langages

L'objectif de ce chapitre est de fournir une introduction aux modèles utilisés en informatique pour décrire, représenter et effectuer des calculs sur des séquences finies de symboles. Avant d'introduire de manière formelle les concepts de base auxquels ces modèles font appel (à partir de la [section 2.2](#)), nous présentons quelques-uns des grands domaines d'application de ces modèles, permettant de mieux saisir l'utilité d'une telle théorie. Cette introduction souligne également la filiation multiple de ce sous-domaine de l'informatique théorique dont les principaux concepts et outils trouvent leur origine aussi bien du côté de la théorie des compilateurs que de la linguistique formelle.

### 2.1 Quelques langages « réels »

#### 2.1.1 La compilation

On désigne ici sous le terme de compilateur tout dispositif permettant de transformer un ensemble de commandes écrites dans un langage de programmation en un autre langage (par exemple une série d'instructions exécutables par une machine). Parmi les tâches préliminaires que doit effectuer un compilateur, il y a l'identification des séquences de caractères qui forment des mots-clés du langage ou des noms de variables licites ou encore des nombres réels : cette étape est l'*analyse lexicale*. Ces séquences s'écrivent sous la forme d'une succession finie de caractères entrés au clavier par l'utilisateur : ce sont donc des séquences de symboles. D'un point de vue formel, le problème que doit résoudre un analyseur lexical consiste donc à caractériser et à discriminer des séquences finies de symboles, permettant de segmenter le programme, vu comme un flux de caractères, en des unités cohérentes et de catégoriser ces unités entre, par exemple : mot-clé, variable, constante...

Seconde tâche importante du compilateur : détecter les erreurs de syntaxe et pour cela identifier, dans l'ensemble des séquences définies sur un alphabet contenant les noms de catégories lexicales (mot-clé, variable, constante...), ainsi qu'un certain nombre d'opérateurs (+, \*, -, :, ...) et de symboles auxiliaires ({, }, ...), les séquences qui sont des programmes correctement formés (ce qui ne présuppose en rien que ces programmes seront sans bogue, ni qu'ils font exactement ce que leur programmeur croit qu'ils font!). L'*analyse syntaxique* se

---

préoccupe, en particulier, de vérifier que les expressions arithmétiques sont bien formées, que les blocs de programmation ou les constructions du langage sont respectées... Comme chacun en a fait l'expérience, tous les programmes ne sont pas syntaxiquement corrects, générant des messages de plainte de la part des compilateurs. L'ensemble des programmes corrects dans un langage de programmation tel que Pascal ou C est donc également un sous-ensemble particulier de toutes les séquences finies que l'on peut former avec les atomes du langage.

En fait, la tâche de l'analyseur syntaxique va même au-delà de ces contrôles, puisqu'elle vise à mettre en évidence la *structure interne* des séquences de symboles qu'on lui soumet. Ainsi par exemple, un compilateur d'expression arithmétique doit pouvoir analyser une séquence telle que  $Var + Var * Var$  comme  $Var + (Var * Var)$ , afin de pouvoir traduire correctement le calcul requis.

Trois problèmes majeurs donc pour les informaticiens : définir la syntaxe des programmes bien formés, discriminer les séquences d'atomes respectant cette syntaxe et identifier la structuration interne des programmes, permettant de déterminer la séquence d'instructions à exécuter.

### 2.1.2 Bio-informatique

La biologie moléculaire et la génétique fournissent des exemples « naturels » d'objets modélisables comme des séquences linéaires de symboles dans un alphabet fini.

Ainsi chaque chromosome, porteur du capital génétique, est-il essentiellement formé de deux brins d'ADN : chacun de ces brins peut être modélisé (en faisant abstraction de la structure tridimensionnelle hélicoïdale) comme une succession de nucléotides, chacun composé d'un phosphate ou acide phosphorique, d'un sucre (désoxyribose) et d'une base azotée. Il existe quatre bases différentes : deux sont dites puriques (la guanine G et l'adénine A), les deux autres sont pyrimidiques (la cytosine C et la thymine T), qui fonctionnent « par paire », la thymine se liant toujours à l'adénine et la cytosine toujours à la guanine. L'information encodée dans ces bases déterminant une partie importante de l'information génétique, une modélisation utile d'un brin de chromosome consiste en la simple séquence linéaire des bases qui le composent, soit en fait une (très longue) séquence définie sur un alphabet de quatre lettres (ATCG).

À partir de ce modèle, la question se pose de savoir rechercher des séquences particulières de nucléotides dans un chromosome ou de détecter des ressemblances/dissemblances entre deux (ou plus) fragments d'ADN. Ces ressemblances génétiques vont servir par exemple à quantifier des proximités évolutives entre populations, à localiser des gènes remplissant des mêmes fonctions dans deux espèces voisines ou encore à réaliser des tests de familiarité entre individus. Rechercher des séquences, mesurer des ressemblances constituent donc deux problèmes de base de la bio-informatique.

Ce type de calculs ne se limite pas aux gènes et est aussi utilisé pour les protéines. En effet, la structure primaire d'une protéine peut être modélisée par la simple séquence linéaire des acides aminés qu'elle contient et qui détermine une partie des propriétés de la protéine. Les acides aminés étant également en nombre fini (20), les protéines peuvent alors être modélisées

comme des séquences finies sur un alphabet comportant 20 lettres.

### 2.1.3 Les langues « naturelles »

Par *langue naturelle*, on entend tout simplement les langues qui sont parlées (parfois aussi écrites) par les humains. Les langues humaines sont, à de multiples niveaux, des systèmes de symboles :

- les suites de sons articulées pour échanger de l'information s'analysent, en dépit de la variabilité acoustique, comme une séquence linéaire unidimensionnelle de symboles choisis parmi un inventaire fini, ceux que l'on utilise dans les transcriptions phonétiques. Toute suite de sons n'est pas pour autant nécessairement une phrase articulable, encore moins une phrase compréhensible ;
- les systèmes d'écriture utilisent universellement un alphabet fini de signes (ceux du français sont des symboles alphabétiques) permettant de représenter les mots sous la forme d'une suite linéaire de ces signes. Là encore, si tout mot se représente comme une suite de lettres, la réciproque est loin d'être vraie ! Les suites de lettres qui sont des mots se trouvent dans les dictionnaires<sup>1</sup> ;
- si l'on admet, en première approximation, que les dictionnaires représentent un nombre fini de mots, alors les phrases de la langue sont aussi des séquences d'éléments pris dans un inventaire fini (le dictionnaire, justement). Toute suite de mots n'est pas une phrase grammaticalement correcte, et toutes les phrases grammaticalement correctes ne sont pas nécessairement compréhensibles.

S'attaquer au traitement informatique des énoncés de la langue naturelle demande donc, de multiples manières, de pouvoir distinguer ce qui est « de la langue » de ce qui n'en est pas. Ceci permet par exemple d'envisager de faire ou proposer des corrections. Le traitement automatique demande également d'identifier la structure des énoncés (« où est le sujet ? », « où est le groupe verbal ? »...) pour vérifier que l'énoncé respecte des règles de grammaire (« le sujet s'accorde avec le verbe ») ; pour essayer de comprendre ce que l'énoncé signifie : le sujet du verbe est (à l'actif) l'agent de l'action ; voire pour traduire dans une autre langue (humaine ou informatique !). De nombreux problèmes du traitement des langues naturelles se modélisent comme des problèmes de théorie des langages, et la théorie des langages doit de nombreuses avancées aux linguistes formels.

## 2.2 Terminologie

### 2.2.1 Bases

Étant donné un ensemble *fini* de symboles  $\Sigma$ , que l'on appelle l'*alphabet*, on appelle *mot* toute suite finie (éventuellement vide) d'éléments de  $\Sigma$ . Par convention, le *mot vide* est noté  $\varepsilon$  ; certains auteurs le notent 1, voire  $1_\Sigma$ . La *longueur d'un mot*  $u$ , notée  $|u|$ , correspond au nombre total de symboles de  $u$  (chaque symbole étant compté autant de fois qu'il apparaît). Bien

---

1. Pas toutes : penser aux formes conjuguées, aux noms propres, aux emprunts, aux néologismes, aux argots. . .

entendu,  $|\varepsilon| = 0$ . Autre notation utile,  $|u|_a$  compte le nombre total d'occurrences du symbole  $a$  dans le mot  $u$ . On a naturellement :  $|u| = \sum_{a \in \Sigma} |u|_a$ .

L'ensemble de tous les mots formés à partir de l'alphabet  $\Sigma$  (resp. de tous les mots non-vides) est noté  $\Sigma^*$  (resp.  $\Sigma^+$ ). Un *langage* sur  $\Sigma$  est un sous-ensemble de  $\Sigma^*$ .

L'opération de *concaténation de deux mots*  $u$  et  $v$  de  $\Sigma^*$  résulte en un nouveau mot  $uv$ , constitué par la juxtaposition des symboles de  $u$  et des symboles de  $v$ . On a alors  $|uv| = |u| + |v|$  et une relation similaire pour les décomptes d'occurrences. La concaténation est une opération interne de  $\Sigma^*$ ; elle est associative, mais pas commutative (sauf dans le cas dégénéré où  $\Sigma$  ne contient qu'un seul symbole).  $\varepsilon$  est l'élément neutre pour la concaténation :  $u\varepsilon = \varepsilon u = u$ ; ceci justifie la notation 1 ou encore  $1_\Sigma$ . Conventionnellement, on notera  $u^n$  la concaténation de  $n$  copies de  $u$ , avec bien sûr  $u^0 = \varepsilon$ . Si  $u$  se factorise sous la forme  $u = xy$ , alors on écrira  $y = x^{-1}u$  et  $x = uy^{-1}$ .

$\Sigma^*$ , muni de l'opération de concaténation, possède donc une structure de *monoïde* (rappelons : un monoïde est un ensemble muni d'une opération interne associative et d'un élément neutre ; lorsqu'il n'y a pas d'élément neutre on parle de *semi-groupe*). Ce monoïde est le *monoïde libre* engendré par  $\Sigma$  : tout mot  $u$  se décompose de manière *unique* comme concaténation de symboles de  $\Sigma$ .

Quelques exemples de langages définis sur l'alphabet  $\Sigma = \{a, b, c\}$ .

- $\Sigma^* = \{\varepsilon, a, b, c, aa, ab, ac, ba, \dots\}$ , soit l'ensemble de tous les mots composés de lettres de  $\Sigma$ ;
- $\{\varepsilon, a, b, c\}$ , soit tous les mots de longueur strictement inférieure à 2;
- $\{ab, aab, abb, acb, aaab, aabb, \dots\}$ , soit tous les mots qui commencent par un  $a$  et finissent par un  $b$ ;
- $\{\varepsilon, ab, aabb, aaabbb, \dots\}$ , soit tous les mots commençant par  $n$   $a$  suivis d'autant de  $b$ . Ce langage est noté  $\{a^n b^n \mid n \geq 0\}$ ;
- $\{\varepsilon, abc, aabbcc, aaabbbccc, \dots\}$ , soit tous les mots contenant  $n$  occurrences de la lettre  $a$ , suivies de  $n$  occurrences de la lettre  $b$ , suivies d'autant de fois la lettre  $c$ . Ce langage est noté  $\{a^n b^n c^n \mid n \geq 0\}$ .
- $\{aa, aaa, aaaaa, \dots\}$ , tous les mots composés d'un nombre *premier* de  $a$ .

Il existe un nombre dénombrable de mots dans  $\Sigma^*$ , mais le nombre de langages dans  $\Sigma^*$  est indénombrable. Parmi ceux-ci, tous ne sont pas à la portée des informaticiens : il existe, en effet, des langages qui « résistent » à tout calcul, c'est-à-dire, plus précisément, qui ne peuvent pas être énumérés par un algorithme.

## 2.2.2 Quelques notions de calculabilité

Plus précisément, la théorie de la calculabilité introduit les distinctions suivantes :

**Définition 2.1** (Langage récursivement énumérable). *Un langage  $L$  est récursivement énumérable s'il existe un algorithme  $A$  qui énumère tous les mots de  $L$ .*

En d'autres termes, un langage  $L$  est récursivement énumérable s'il existe un algorithme  $A$  (ou, de manière équivalente, une machine de Turing) tel que tout mot de  $L$  est produit par un

nombre fini d'étapes d'exécution de  $A$ . Autrement dit, si  $L$  est récursivement énumérable et  $u$  est un mot de  $L$ , alors en laissant tourner  $A$  « assez longtemps », l'algorithme énumérateur  $A$  finira par produire  $u$ .

Il est équivalent de définir les langages récursivement énumérables comme les langages  $L$  pour lesquels il existe une machine de Turing qui *reconnaît* les mots de  $L$ , c'est-à-dire qui s'arrête dans un état d'acceptation pour tout mot de  $L$ . Cela ne préjuge en rien du comportement de la machine pour un mot qui n'est pas dans  $L$  : en particulier cette définition est compatible avec une machine de Turing bouclant sans fin pour certains mots n'appartenant pas à  $L$ .

**Définition 2.2** (Langage récursif). *Un langage  $L$  est récursif s'il existe un algorithme  $A$  qui, prenant un mot  $u$  de  $\Sigma^*$  en entrée, répond oui si  $u$  est dans  $L$  et répond non sinon. On dit alors que l'algorithme  $A$  décide le langage  $L$ .*

Tout langage récursif est récursivement énumérable : il suffit, pour construire une énumération de  $L$ , de prendre une procédure quelconque d'énumération de  $\Sigma^*$  (par exemple par longueur croissante) et de soumettre chaque mot énuméré à l'algorithme  $A$  qui décide  $L$ . Si la réponse de  $A$  est *oui*, on produit le mot courant, sinon, on passe au suivant. Cette procédure énumère effectivement tous les mots de  $L$ . Seuls les langages récursifs ont un réel intérêt pratique, puisqu'ils correspondent à des distinctions qui sont calculables entre mots dans  $L$  et mots hors de  $L$ .

De ces définitions, retenons une première limitation de notre savoir d'informaticien : il existe des langages que ne nous savons pas énumérer. Ceux-ci ne nous intéresseront plus guère. Il en existe d'autres que nous savons décider et qui recevront la majeure partie de notre attention.

Au-delà des problèmes de la reconnaissance et de la décision, il existe d'autres types de calculs que nous envisagerons et qui ont des applications bien pratiques dans les différents domaines d'applications évoqués ci-dessus :

- comparer deux mots, évaluer leur ressemblance
- rechercher un motif dans un mot
- comparer deux langages
- apprendre un langage à partir d'exemples
- ...

## 2.3 Opérations sur les mots

### 2.3.1 Facteurs et sous-mots

On dit que  $u$  est un *facteur* de  $v$  s'il existe  $u_1$  et  $u_2$  dans  $\Sigma^*$  tels que  $v = u_1 u u_2$ . Si  $u_1 = \varepsilon$  (resp.  $u_2 = \varepsilon$ ), alors  $u$  est un *préfixe* (resp. *suffixe*) de  $v$ . Si  $w$  se factorise en  $u_1 v_1 u_2 v_2 \dots u_n v_n u_{n+1}$ , où tous les  $u_i$  et  $v_i$  sont des mots de  $\Sigma^*$ , alors  $v = v_1 v_2 \dots v_n$  est *sous-mot*<sup>2</sup> de  $w$ . Contrairement

---

2. Attention : il y a ici désaccord entre les terminologies françaises et anglaises : *subword* ou *substring* signifie en fait *facteur* et c'est *subsequence* ou *scattered subword* qui est l'équivalent anglais de notre *sous-mot*.

aux facteurs, les sous-mots sont donc construits à partir de fragments non nécessairement contigus, mais dans lesquels l'ordre d'apparition des symboles est toutefois respecté. On appelle facteur (resp. préfixe, suffixe, sous-mot) *propre* de  $u$  tout facteur (resp. préfixe, suffixe, sous-mot) de  $u$  différent de  $u$ .

On notera  $|pref|_k(u)$  (resp.  $|suff|_k(u)$ ) le préfixe (resp. le suffixe) de longueur  $k$  de  $u$ . Si  $k \geq |u|$ ,  $|pref|_k(u)$  désigne simplement  $u$ .

Les notions de préfixe et de suffixe généralisent celles des linguistes<sup>3</sup> : tout le monde s'accorde sur le fait que *in* est un préfixe de *infini* ; seuls les informaticiens pensent qu'il en va de même pour *i*, *inf* ou encore *infi*. De même, tout le monde est d'accord pour dire que *ure* est un suffixe de *voilure* ; mais seuls les informaticiens pensent que *ilure* est un autre suffixe de *voilure*.

Un mot non-vide  $u$  est *primitif* si l'équation  $u = v^i$  n'admet pas de solution pour  $i > 1$ .

Deux mots  $x = uv$  et  $y = vu$  se déduisant l'un de l'autre par échange de préfixe et de suffixe sont dits *conjugués*. Il est facile de vérifier que la relation de conjugaison<sup>4</sup> est une relation d'équivalence.

Le *miroir* ou *transposé*  $u^R$  du mot  $u = a_1 \dots a_n$ , où  $a_i \in \Sigma$ , est défini par :  $u^R = a_n \dots a_1$ . Un mot est un *palindrome* s'il est égal à son miroir. *radar*, *sas* sont des palindromes du vocabulaire commun. On vérifie simplement que les préfixes de  $u^R$  sont précisément les transposés des suffixes de  $u$  et réciproquement.

### 2.3.2 Quotient de mots

**Définition 2.3** (Quotient droit d'un mot). *Le quotient droit d'un mot  $u$  par le mot  $v$ , dénoté  $uv^{-1}$  ou encore  $u|_v$ , est défini par :*

$$u|_v = uv^{-1} = \begin{cases} w & \text{si } u = wv \\ \text{pas défini} & \text{si } v \text{ n'est pas un suffixe de } u \end{cases}$$

Par exemple  $abcde(cde)^{-1} = ab$ , et  $abd(abc)^{-1}$  n'est pas défini. Il ne faut pas voir  $uv^{-1}$  comme un produit :  $v^{-1}$  ne représente pas un mot. On peut définir de la même façon le quotient gauche de  $v$  par  $u$  :  $u^{-1}v$  ou  ${}_u v$ .

### 2.3.3 Ordres sur les mots

#### Une famille d'ordres partiels

Les relations de préfixe, suffixe, facteur et sous-mot induisent autant de *relations d'ordre* sur  $\Sigma^*$  : ce sont, en effet, des relations réflexives, transitives et antisymétriques. Ainsi pourra-t-on dire que  $u \preceq_p v$  si  $u$  est un préfixe de  $v$ . Deux mots quelconques ne sont pas nécessairement comparables pour ces relations : ces ordres sont *partiels*.

---

3. On parle aussi en linguistique de *terminaison* au lieu de suffixe.

4. Ici, rien à voir avec la conjugaison des grammairiens.

### Des ordres totaux

Il est possible de définir des ordres *totaux* sur  $\Sigma^*$ , à la condition de disposer d'un ordre total  $\leq$  sur  $\Sigma$ .

**Définition 2.4** (Ordre lexicographique). *L'ordre lexicographique sur  $\Sigma^*$  noté  $\leq_l$  est défini par  $u \leq_l v$  ssi*

- soit  $u$  est un préfixe de  $v$
- soit sinon  $u = tu', v = tv'$  avec  $u' \neq \varepsilon$  et  $v' \neq \varepsilon$ , et le premier symbole de  $u'$  précède celui de  $v'$  pour  $\leq$ .

Cet ordre conduit à des résultats contre-intuitifs lorsque l'on manipule des langages infinis. Par exemple il existe un nombre infini de prédécesseurs au mot  $b$  dans  $\{a, b\}^*$  :  $\{\varepsilon, a, aa, ab, aaa, \dots\}$ .

L'ordre radiciel (ou *ordre alphabétique*, ou encore *ordre militaire*) utilise également  $\leq$ , mais privilégie, lors des comparaisons, la longueur des chaînes.

**Définition 2.5** (Ordre radiciel). *L'ordre radiciel sur  $\Sigma^*$  noté  $\leq_a$  est défini par  $u \leq_a v$  ssi*

- soit  $|u| \leq |v|$
- soit sinon  $|u| = |v|$  et  $u \leq_l v$

Contrairement à l'ordre lexicographique, il s'agit d'un *ordre bien fondé* : les mots plus petits qu'un mot arbitraire  $u$  sont en nombre *fini*. D'autre part pour tout  $w, w'$  si  $u \leq_a v$  alors  $wuw' \leq_a ww'w'$ , ce qui n'est pas le cas pour l'ordre lexicographique (p.ex. :  $a \leq_l ab$ , mais  $c \cdot a \cdot d >_l c \cdot ab \cdot d$ ).

On notera que les dictionnaires utilisent l'ordre lexicographique et non celui nommé ici « alphabétique ».

### 2.3.4 Distances entre mots

#### Une famille de distances

Pour toute paire de mots il existe un plus long préfixe (resp. suffixe, facteur, sous-mot) commun. Dans le cas des suffixes et préfixes, ce plus long facteur commun est de plus unique.

Si l'on note  $\text{plpc}(u, v)$  le plus long préfixe commun à  $u$  et à  $v$ , alors la fonction  $d_p(u, v)$  définie par :

$$d_p(u, v) = |uv| - 2|\text{plpc}(u, v)|$$

définit une distance sur  $\Sigma^*$ , la *distance préfixe*. On vérifie en effet que :

- $d_p(u, v) \geq 0$
- $d_p(u, v) = 0 \Leftrightarrow u = v$

—  $d_p(u, w) \leq d_p(u, v) + d_p(v, w)$ .

La vérification de cette inégalité utilise le fait que le plus long préfixe commun à  $u$  et à  $w$  est au moins aussi long que le plus long préfixe commun à  $\text{plpc}(u, v)$  et à  $\text{plpc}(v, w)$ .

On obtient également une distance lorsque, au lieu de considérer la longueur des plus longs préfixes communs, on considère celle des plus longs suffixes ( $d_s$ ), des plus longs facteurs ( $d_f$ ) ou des plus longs sous-mots communs ( $d_m$ ).

*Démonstration.* Dans tous les cas, la seule propriété demandant un effort de justification est l'inégalité triangulaire. Dans le cas des suffixes, elle se démontre comme pour les préfixes.

Pour traiter le cas des facteurs et des sous-mots, il est utile de considérer les mots sous une perspective un peu différente. Il est en effet possible d'envisager un mot  $u$  de  $\Sigma^+$  comme une fonction de l'intervalle  $I = [1 \dots |u|]$  vers  $\Sigma$ , qui à chaque entier  $i$  associe le  $i^{\text{e}}$  symbole de  $u$  :  $u(i) = u_i$ . À toute séquence croissante d'indices correspond alors un sous-mot ; si ces indices sont consécutifs on obtient un facteur.

Nous traitons dans la suite le cas des sous-mots et notons  $\text{plsmc}(u, v)$  le plus long sous-mot commun à  $u$  et à  $v$ . Vérifier  $d_m(u, w) \leq d_m(u, v) + d_m(v, w)$  revient à vérifier que :

$$|uw| - 2|\text{plsmc}(u, w)| \leq |uv| - 2|\text{plsmc}(u, v)| + |vw| - 2|\text{plsmc}(v, w)|$$

soit encore :

$$|\text{plsmc}(u, v)| + |\text{plsmc}(v, w)| \leq |v| + |\text{plsmc}(u, w)|$$

En notant  $I$  et  $J$  les séquences d'indices de  $[1 \dots |v|]$  correspondant respectivement à  $\text{plsmc}(u, v)$  et à  $\text{plsmc}(v, w)$ , on note tout d'abord que :

$$\begin{aligned} |\text{plsmc}(u, v)| + |\text{plsmc}(v, w)| &= |I| + |J| \\ &= |I \cup J| + |I \cap J| \end{aligned}$$

On note ensuite que le sous-mot de  $v$  construit en considérant les symboles aux positions de  $I \cap J$  est un sous-mot de  $u$  et de  $w$ , donc nécessairement au plus aussi long que  $\text{plsmc}(u, w)$ . On en déduit donc que :  $|I \cap J| \leq |\text{plsmc}(u, w)|$ . Puisque, par ailleurs, on a  $|I \cup J| \leq |v|$ , on peut conclure que :

$$|\text{plsmc}(u, w)| + |v| \geq |I \cup J| + |I \cap J|$$

Et on obtient ainsi précisément ce qu'il fallait démontrer. Le cas des facteurs se traite de manière similaire. □

### Distance d'édition et variantes

Une autre distance communément utilisée sur  $\Sigma^*$  est la *distance d'édition*, ou *distance de Levenshtein*, définie comme étant le plus petit nombre d'opérations d'édition élémentaires nécessaires pour transformer le mot  $u$  en le mot  $v$ . Les opérations d'édition élémentaires sont la suppression ou l'insertion d'un symbole. Ainsi la distance de *chien* à *chameau* est-elle de 6, puisque l'on peut transformer le premier mot en l'autre en faisant successivement les opérations suivantes : supprimer  $i$ , insérer  $a$ , puis  $m$ , supprimer  $n$ , insérer  $a$ , puis  $u$ . Cette métamorphose d'un mot en un autre est décomposée dans le [tableau 2.1](#).



mot courant	opération
<i>chien</i>	supprimer <i>i</i>
<i>chen</i>	insérer <i>a</i>
<i>chaen</i>	insérer <i>m</i>
<i>chamen</i>	supprimer <i>n</i>
<i>chame</i>	insérer <i>a</i>
<i>chamea</i>	insérer <i>u</i>
<i>chameau</i>	

Deux mots consécutifs sont à distance 1. L'ordre des opérations élémentaires est arbitraire.

TABLE 2.1 – Métamorphose de chien en chameau

De multiples variantes de cette notion de distance ont été proposées, qui utilisent des ensembles d'opérations différents et/ou considèrent des poids variables pour les différentes opérations. Pour prendre un exemple réel, si l'on souhaite réaliser une application qui « corrige » les fautes de frappe au clavier, il est utile de considérer des poids qui rendent d'autant plus proches des séquences qu'elles ne diffèrent que par des touches voisines sur le clavier, permettant d'intégrer une modélisation des confusions de touches les plus probables. On considérera ainsi, par exemple, que *batte* est une meilleure correction de *bqtte* que *botte* ne l'est<sup>5</sup>, bien que les deux mots se transforment en *bqtte* par une série de deux opérations élémentaires.

L'utilitaire Unix `diff` implante une forme de calcul de distances. Cet utilitaire permet de comparer deux fichiers et d'imprimer sur la sortie standard toutes les différences entre leurs contenus respectifs.

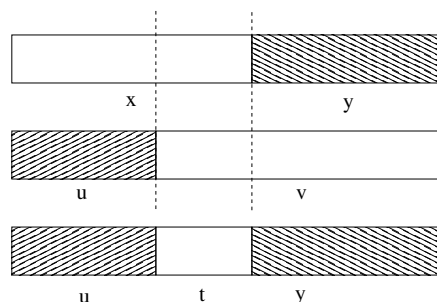
### 2.3.5 Quelques résultats combinatoires élémentaires

La propriété suivante est trivialement respectée :

**Lemme 2.6.**

$$\forall u, v, x, y \in \Sigma^*, uv = xy \Rightarrow \exists t \in \Sigma^* \text{ tq. soit } u = xt \text{ et } tv = y, \text{ soit } x = ut \text{ et } v = ty.$$

Cette propriété est illustrée sur la figure suivante :



5. C'est une première approximation : pour bien faire il faudrait aussi prendre en compte la fréquence relative des mots proposés... Mais c'est mieux que rien.

Ce résultat est utilisé pour démontrer deux autres résultats élémentaires, qui découlent de la non-commutativité de la concaténation.

**Théorème 2.7.** Si  $xy = yz$ , avec  $x \neq \varepsilon$ , alors  $\exists u, v \in \Sigma^*$  et un entier  $k \geq 0$  tels que :  $x = uv$ ,  $y = (uv)^k u = u(vu)^k$ ,  $z = vu$ .

*Démonstration.* Si  $|x| \geq |y|$ , alors le résultat précédent nous permet d'écrire directement  $x = yt$ , ce qui, en identifiant  $u$  et  $y$ , et  $v$  à  $t$ , nous permet de dériver directement les égalités voulues pour  $k = 0$ .

Le cas où  $|x| < |y|$  se traite par induction sur la longueur de  $y$ . Le cas où  $|y|$  vaut 1 étant immédiat, supposons la relation vraie pour tout  $y$  de longueur au moins  $n$ , et considérons  $y$  avec  $|y| = n + 1$ . Il existe alors  $t$  tel que  $y = xt$ , d'où l'on dérive  $xtz = xxt$ , soit encore  $tz = xt$ , avec  $|t| \leq n$ . L'hypothèse de récurrence garantit l'existence de  $u$  et  $v$  tels que  $x = uv$  et  $t = (uv)^k u$ , d'où  $y = uv(uv)^k u = (uv)^{k+1} u$ .  $\square$

**Théorème 2.8.** Si  $xy = yx$ , avec  $x \neq \varepsilon$ ,  $y \neq \varepsilon$ , alors  $\exists u \in \Sigma^*$  et deux indices  $i$  et  $j$  tels que  $x = u^i$  et  $y = u^j$ .

*Démonstration.* Ce résultat s'obtient de nouveau par induction sur la longueur de  $xy$ . Pour une longueur égale à 2 le résultat vaut trivialement. Supposons le valable jusqu'à la longueur  $n$ , et considérons  $xy$  de longueur  $n + 1$ . Par le théorème précédent, il existe  $u$  et  $v$  tels que  $x = uv$ ,  $y = (uv)^k u$ , d'où on déduit :  $uv(uv)^k u = (uv)^k uuv$ , soit encore  $uv = vu$ . En utilisant l'hypothèse de récurrence il vient alors :  $u = t^i$ ,  $v = t^j$ , puis encore  $x = t^{i+j}$  et  $y = t^{i+k(i+j)}$ , qui est le résultat recherché.  $\square$

L'interprétation de ces résultats est que les équations du type  $xy = yx$  n'admettent que des solutions *périodiques*, c'est-à-dire des séquences qui sont construites par itération d'un même motif de base.

## 2.4 Opérations sur les langages

### 2.4.1 Opérations ensemblistes

Les langages étant des ensembles, toutes les opérations ensemblistes « classiques » leur sont donc applicables. Ainsi, les opérations d'union, d'intersection et de complémentation (dans  $\Sigma^*$ ) se définissent-elles pour  $L, L_1$  et  $L_2$  des langages de  $\Sigma^*$  par :

$$\begin{aligned} L_1 \cup L_2 &= \{u \in \Sigma^* \mid u \in L_1 \text{ ou } u \in L_2\} \\ L_1 \cap L_2 &= \{u \in \Sigma^* \mid u \in L_1 \text{ et } u \in L_2\} \\ \bar{L} &= \{u \in \Sigma^* \mid u \notin L\} \end{aligned}$$

Les opérations  $\cup$  et  $\cap$  sont associatives et commutatives.

### 2.4.2 Concaténation, Étoile de Kleene

L'opération de concaténation, définie sur les mots, engendre naturellement la *concaténation de langages* (on dit également le *produit de langages*, mais ce n'est pas le produit cartésien) :

$$L_1 L_2 = \{u \in \Sigma^* \mid \exists (x, y) \in L_1 \times L_2 \text{ tq. } u = xy\}$$

On note, de nouveau, que cette opération est associative, mais pas commutative. Comme précédemment, l'itération de  $n$  copies du langage  $L$  se notera  $L^n$ , avec, par convention :  $L^0 = \{\varepsilon\}$ . Attention : ne pas confondre  $L^n$  avec le langage contenant les puissances nièmes des mots de  $L$  et qui serait défini par  $\{u \in \Sigma^* \mid \exists v \in L, u = v^n\}$ .

L'opération de *fermeture de Kleene* (ou plus simplement *l'étoile*) d'un langage  $L$  se définit par :

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

$L^*$  contient tous les mots qu'il est possible de construire en concaténant un nombre fini (éventuellement réduit à zéro) d'éléments du langage  $L$ . On notera que si  $\Sigma$  est un alphabet,  $\Sigma^*$ , tel que défini précédemment, représente<sup>6</sup> l'ensemble des séquences finies que l'on peut construire en concaténant des symboles de  $\Sigma$ . Remarquons que, par définition,  $\emptyset^*$  n'est pas vide, puisqu'il (ne) contient (que)  $\varepsilon$ .

On définit également

$$L^+ = \bigcup_{i \geq 1} L^i$$

À la différence de  $L^*$  qui contient toujours  $\varepsilon$ ,  $L^+$  ne contient  $\varepsilon$  que si  $L$  le contient. On a :  $L^+ = LL^*$ .

Un des intérêts de ces notations est qu'elles permettent d'exprimer de manière formelle (et compacte) des langages complexes, éventuellement infinis, à partir de langages plus simples. Ainsi l'ensemble des suites de 0 et de 1 contenant la séquence 111 s'écrira par exemple :  $\{0, 1\}^* \{111\} \{0, 1\}^*$ , la notation  $\{0, 1\}^*$  permettant un nombre arbitraire de 0 et de 1 avant la séquence 111. La notion d'expression rationnelle, introduite au [chapitre 3](#), développe cette intuition.

### 2.4.3 Plus d'opérations dans $\mathcal{P}(\Sigma^*)$

Pour un langage  $L$  sur  $\Sigma^*$ , on définit les concepts suivants :

**Définition 2.9** (Langage des préfixes). *Soit  $L$  un langage de  $\Sigma^*$ , on définit le langage des préfixes de  $L$ , noté  $\text{Pref}(L)$  par :*

$$\text{Pref}(L) = \{u \in \Sigma^* \mid \exists v \in \Sigma^*, uv \in L\}$$

6. Notez que, ce faisant, on identifie un peu abusivement les symboles (les éléments de  $\Sigma$ ) et les séquences formées d'un seul symbole de  $\Sigma$ .

Attention à ne pas confondre cette notion avec celle des langages préfixes. On dit qu'un langage  $L$  est un *langage préfixe* si pour tout  $u, v \in L$ ,  $u \neq v$ , on a  $u \notin \text{Pref}(v)$ . En utilisant un langage préfixe fini, il est possible de définir un procédé de codage donnant lieu à des algorithmes de décodage simples : ces codes sont appelés *codes préfixes*. En particulier, les codes produits par les codages de Huffman sont des codes préfixes.

**Exercice 2.10.** *Petite application du concept : montrez que le produit de deux langages préfixes est encore un langage préfixe.*

**Définition 2.11** (Langage des suffixes). *Soit  $L$  un langage de  $\Sigma^*$ , on définit le langage des suffixes de  $L$ , noté  $\text{Suff}(L)$  par :*

$$\text{Suff}(L) = \{u \in \Sigma^* \mid \exists v \in \Sigma^*, vu \in L\}$$

**Définition 2.12** (Langage des facteurs). *Soit  $L$  un langage de  $\Sigma^*$ , on définit le langage des facteurs de  $L$ , noté  $\text{Fac}(L)$  par :*

$$\text{Fac}(L) = \{v \in \Sigma^* \mid \exists u, w \in \Sigma^*, uvw \in L\}$$

Les quotients de mots gauche et droit s'étendent additivement aux langages.

**Définition 2.13** (Quotient droit d'un langage). *Le quotient droit d'un langage  $L$  par le mot  $u$  est défini par :*

$$L/u = Lu^{-1} = \bigcup_{v \in L} \{vu^{-1}\} = \{w \in \Sigma^* \mid wu \in L\}$$

Le quotient droit de  $L$  par  $u$  est donc l'ensemble des mots de  $\Sigma^*$  dont la concaténation par  $u$  est dans  $L$ . De même, le quotient gauche de  $L$  par  $u$ ,  $u^{-1}L = {}_u \setminus L$ , est l'ensemble des mots de  $\Sigma^*$  qui, concaténés à  $u$ , produisent un mot de  $L$ .

**Définition 2.14** (Congruence droite). *Une congruence droite de  $\Sigma^*$  est une relation  $\mathcal{R}$  de  $\Sigma^*$  qui vérifie :*

$$\forall w, w' \in \Sigma^*, w \mathcal{R} w' \Rightarrow \forall u, wu \mathcal{R} w'u$$

**Définition 2.15** (Congruence droite associée à un langage  $L$ ). *Soit  $L$  un langage et soient  $w_1, w_2$  deux mots tels que  $L/w_1 = L/w_2$ . Il est clair, par définition du quotient, que l'on a alors  $\forall u, L/w_1u = L/w_2u$ . S'en déduit une congruence droite  $\mathcal{R}_L$  « naturellement » associée à  $L$  et définie par :*

$$w_1 \mathcal{R}_L w_2 \Leftrightarrow L/w_1 = L/w_2$$

#### 2.4.4 Morphismes

**Définition 2.16** (Morphisme). *Un morphisme d'un monoïde  $M$  dans un monoïde  $N$  est une application  $\phi$  telle que :*

- $\phi(\varepsilon_M) = \varepsilon_N$  : l'image de l'élément neutre de  $M$  est l'élément neutre de  $N$ .
- $\phi(uv) = \phi(u)\phi(v)$

L'application longueur est un morphisme de monoïde de  $(\Sigma^*, \cdot)$  dans  $(\mathbb{N}, +)$  : on a trivialement  $|\varepsilon| = 0$  et  $|uv| = |u| + |v|$ . On vérifie simplement qu'il en va de même pour les fonctions de comptage des occurrences.

**Définition 2.17** (Code). *Un code est un morphisme injectif :  $\phi(u) = \phi(v)$  entraîne  $u = v$ .*

## Chapitre 3

# Langages et expressions rationnels

Une première famille de langages est introduite, la famille des langages *rationnels*. Cette famille contient en particulier tous les langages finis, mais également de nombreux langages infinis. La caractéristique de tous ces langages est la possibilité de les *décrire* par des formules (on dit aussi *motifs*, en anglais *patterns*) très simples. L'utilisation de ces formules, connues sous le nom d'expressions rationnelles<sup>1</sup>, s'est imposée sous de multiples formes comme la « bonne » manière de décrire des motifs représentant des ensembles de mots.

Après avoir introduit les principaux concepts formels (à la [section 3.1](#)), nous étudions quelques systèmes informatiques classiques mettant ces concepts en application.



Les concepts introduits dans ce chapitre et le suivant seront illustrés en utilisant Vcsn (<http://vcsn.lrde.epita.fr>). Il s'agit d'une plate-forme de manipulation d'automates et d'expressions rationnelles résultant d'une collaboration entre Télécom ParisTech, le Laboratoire Bordelais d'Informatique (LaBRI), et le Laboratoire de R&D de l'EPITA (LRDE). La documentation est en ligne : <http://vcsn-sandbox.lrde.epita.fr/notebooks/Doc/index.ipynb>.

La commande shell 'vcsn notebook' ouvre une session interactive sous IPython. À défaut, elle est disponible en ligne, <http://vcsn-sandbox.lrde.epita.fr>.

Créez une nouvelle page, ou bien ouvrez une page existante. Dans cet environnement ENTER insère un saut de ligne, et SHIFT-ENTER lance l'évaluation de la cellule courante et passe à la suivante. Une fois la session interactive lancée, exécuter 'import vcsn'.

---

1. On trouve également le terme d'expression *régulière*, mais cette terminologie, quoique bien installée, est trompeuse et nous ne l'utiliserons pas dans ce cours.

---

## 3.1 Rationalité

### 3.1.1 Langages rationnels

Parmi les opérations définies dans  $\mathcal{P}(\Sigma^*)$  à la [section 2.4](#), trois sont distinguées et sont qualifiées de *rationnelles* : il s'agit de l'union, de la concaténation et de l'étoile. *A contrario*, notez que la complémentation et l'intersection ne sont pas des opérations rationnelles. Cette distinction permet de définir une famille importante de langages : les langages *rationnels*.

**Définition 3.1** (Langage rationnel). *Soit  $\Sigma$  un alphabet. Les langages rationnels sur  $\Sigma$  sont définis inductivement par :*

- (i)  $\{\varepsilon\}$  et  $\emptyset$  sont des langages rationnels
- (ii)  $\forall a \in \Sigma, \{a\}$  est un langage rationnel
- (iii) si  $L_1$  et  $L_2$  sont des langages rationnels, alors  $L_1 \cup L_2$ ,  $L_1L_2$ , et  $L_1^*$  sont également des langages rationnels.

Bien entendu, dans cette définition inductive on n'a le droit qu'à un nombre fini d'applications de la récurrence (iii).

Tous les langages finis sont rationnels, puisqu'ils se déduisent des singletons par un nombre fini d'applications des opérations d'union et de concaténation. Par définition, l'ensemble des langages rationnels est clos pour les trois opérations rationnelles (on dit aussi qu'il est rationnellement clos).

La famille des langages rationnels correspond précisément au plus petit ensemble de langages qui (i) contient tous les langages finis, (ii) est rationnellement clos.

Un langage rationnel peut se décomposer sous la forme d'une formule finie, correspondant aux opérations (rationnelles) qui permettent de le construire. Prenons l'exemple du langage sur  $\{0, 1\}$  contenant tous les mots dans lesquels apparaît au moins une fois le facteur 111. Ce langage peut s'écrire :  $(\{0\} \cup \{1\})^* \{1\}\{1\}\{1\}(\{0\} \cup \{1\})^*$ , exprimant que les mots de ce langage sont construits en prenant deux mots quelconques de  $\Sigma^*$  et en insérant entre eux le mot 111 : on peut en déduire que ce langage est bien rationnel. Les *expressions rationnelles* définissent un système de formules qui simplifient et étendent ce type de notation des langages rationnels.

### 3.1.2 Expressions rationnelles

**Définition 3.2** (Expression rationnelle). *Soit  $\Sigma$  un alphabet. Les expressions rationnelles (RE) sur  $\Sigma$  sont définies inductivement par :*

- (i)  $\varepsilon$  et  $\emptyset$  sont des expressions rationnelles
- (ii)  $\forall a \in \Sigma, a$  est une expression rationnelle
- (iii) si  $e_1$  et  $e_2$  sont deux expressions rationnelles, alors  $(e_1 + e_2)$ ,  $(e_1e_2)$ , et  $(e_1^*)$  sont également des expressions rationnelles.

Une expression rationnelle est donc toute formule construite par un nombre *fini* d'applications de la récurrence (iii).

Illustrons ce nouveau concept, en prenant maintenant l'ensemble des caractères alphabétiques comme ensemble de symboles :

- $r, e, d, \acute{e}$ , sont des RE (par (ii))
- $(re)$  et  $(d\acute{e})$  sont des RE (par (iii))
- $((((fa)ir)e)$  est une RE (par (ii), puis (iii))
- $((re) + (d\acute{e}))$  est une RE (par (iii))
- $((((re) + (d\acute{e}))^*)$  est une RE (par (iii))
- $(((((re) + (d\acute{e}))^*)((fa)ir)e)$  est une RE (par (iii))
- ...

À quoi servent ces formules ? Comme annoncé, elles servent à dénoter des langages rationnels. L'interprétation (la sémantique) d'une expression est définie par les règles inductives suivantes :

- (i)  $\varepsilon$  dénote le langage  $\{\varepsilon\}$  et  $\emptyset$  dénote le langage vide.
- (ii)  $\forall a \in \Sigma, a$  dénote le langage  $\{a\}$
- (iii.1)  $(e_1 + e_2)$  dénote l'union des langages dénotés par  $e_1$  et par  $e_2$
- (iii.2)  $(e_1e_2)$  dénote la concaténation des langages dénotés par  $e_1$  et par  $e_2$
- (iii.3)  $(e^*)$  dénote l'étoile du langage dénoté par  $e$

Pour alléger les notations (et limiter le nombre de parenthèses), on imposera les règles de priorité suivantes : l'étoile ( $\star$ ) est l'opérateur le plus liant, puis la concaténation, puis l'union (+). Les opérateurs binaires sont pris associatifs à gauche. Ainsi,  $aa^* + b^*$  s'interprète-t-il comme  $((a(a^*)) + (b^*))$ , et  $(((((re) + (d\acute{e}))^*)((fa)ir)e)$  peut s'écrire  $(re + d\acute{e})^* faire$ .

Revenons à la formule précédente :  $(re + d\acute{e})^* faire$  dénote l'ensemble des mots formés en itérant à volonté un des deux préfixes  $re$  ou  $d\acute{e}$ , concaténé au suffixe  $faire$  : cet ensemble décrit en fait un ensemble de mots existants ou potentiels de la langue française qui sont dérivés par application d'un procédé tout à fait régulier de préfixation verbale.

Par construction, les expressions rationnelles permettent de dénoter précisément tous les langages rationnels, et rien de plus. Si, en effet, un langage est rationnel, alors il existe une expression rationnelle qui le dénote. Ceci se montre par une simple récurrence sur le nombre d'opérations rationnelles utilisées pour construire le langage. Réciproquement, si un langage est dénoté par une expression rationnelle, alors il est lui-même rationnel (de nouveau par induction sur le nombre d'étapes dans la définition de l'expression). Ce dernier point est important, car il fournit une première méthode pour *prouver* qu'un langage est rationnel : il suffit pour cela d'exhiber une expression qui le dénote.



Vcsn travaille sur des expressions rationnelles (et des automates) de types plus généraux. Le concept de type se nomme « contexte » dans Vcsn. Nous utiliserons le contexte le plus simple, `'lal_char(a-z), b'`, qui désigne les expressions sur l'alphabet  $\{a, b, \dots, z\}$  qui « calculent » un booléen ( $\mathbb{B}$ ). Le cryptique `'lal_char'` signifie que les étiquettes sont des lettres (*labels are letters*) et que les lettres sont de simples `'char'`.

Définissons ce contexte.

```
>>> import vcsn
>>> ctx = vcsn.context('lal_char(abc), b')
```

Puis construisons quelques expressions rationnelles.

```
>>> ctx.expression('(a+b+c)*')
>>> ctx.expression('(a+b+c)*abc(a+b+c)*')
```

La syntaxe des expressions rationnelles est documentée sur la page [Expressions](#).

### 3.1.3 Équivalence et réductions

La correspondance entre expression et langage n'est pas biunivoque : chaque expression dénote un unique langage, mais à un langage donné peuvent correspondre plusieurs expressions différentes. Ainsi, les deux expressions suivantes :  $a^*(a^*ba^*ba^*)^*$  et  $a^*(ba^*ba^*)^*$  sont-elles en réalité deux variantes notationnelles du même langage sur  $\Sigma = \{a, b\}$ .

**Définition 3.3** (Expressions rationnelles équivalentes). *Deux expressions rationnelles sont équivalentes si elles dénotent le même langage.*

Comment déterminer automatiquement que deux expressions sont équivalentes ? Existe-t-il une expression canonique, correspondant à la manière la plus courte de dénoter un langage ? Cette question n'est pas anodine : pour calculer efficacement le langage associé à une expression, il semble préférable de partir de la version la plus simple, afin de minimiser le nombre d'opérations à accomplir.

Un élément de réponse est fourni avec les formules du [tableau 3.1](#) qui expriment (par le signe  $\equiv$ ) un certain nombre d'équivalences élémentaires.

$$\begin{array}{ll}
 \emptyset e \equiv \emptyset & \varepsilon e \equiv e \\
 e \emptyset \equiv \emptyset & e \varepsilon \equiv e \\
 \emptyset^* \equiv \varepsilon & \varepsilon^* \equiv \varepsilon \\
 e + f \equiv f + e & e + \emptyset \equiv e \\
 e + e \equiv e & (e^*)^* \equiv e^* \\
 e(f + g) \equiv ef + eg & (e + f)g \equiv eg + fg \\
 (ef)^*e \equiv e(fe)^* & \\
 (e + f)^* \equiv e^*(e + f)^* & (e + f)^* \equiv (e^* + f)^* \\
 (e + f)^* \equiv (e^*f^*)^* & (e + f)^* \equiv (e^*f)^*e^*
 \end{array}$$

TABLE 3.1 – Identités rationnelles

En utilisant ces identités, il devient possible d'opérer des transformations purement syn-



taxiques<sup>2</sup> qui préservent le langage dénoté, en particulier pour les simplifier. Un exemple de réduction obtenue par application de ces expressions est le suivant :

$$\begin{aligned} bb^*(a^*b^* + \varepsilon)b &\equiv b(b^*a^*b^* + b^*)b \\ &\equiv b(b^*a^* + \varepsilon)b^*b \\ &\equiv b(b^*a^* + \varepsilon)bb^* \end{aligned}$$



Observez certaines simplifications.

```
>>> ctx.expression('(c+a+b+a)*+\z')
```

```
>>> ctx.expression('\e*')
```

La conceptualisation algorithmique d'une stratégie efficace permettant de réduire les expressions rationnelles sur la base des identités du [tableau 3.1](#) étant un projet difficile, l'approche la plus utilisée pour tester l'équivalence de deux expressions rationnelles n'utilise pas directement ces identités, mais fait plutôt appel à leur transformation en automates finis, qui sera présentée dans le chapitre suivant (à la [section 4.2.2](#)).

## 3.2 Extensions notationnelles

Les expressions rationnelles constituent un outil puissant pour décrire des langages simples (rationnels). La nécessité de décrire de tels langages étant récurrente en informatique, ces formules sont donc utilisées, avec de multiples extensions, dans de nombreux outils d'usage courant.

Par exemple, `grep` est un utilitaire disponible sous UNIX pour rechercher les occurrences d'un mot(if) dans un fichier texte. Son utilisation est simplissime :

```
> grep 'chaîne' mon-texte
```

imprime sur la sortie standard toutes les *lignes* du fichier 'mon-texte' contenant au moins une occurrence du mot chaîne.

En fait `grep` permet un peu plus : à la place d'un mot unique, il est possible d'imprimer les occurrences de tous les mots d'un langage rationnel quelconque, ce langage étant défini sous la forme d'une expression rationnelle. Ainsi, par exemple :

```
> grep 'cha*îne' mon-texte
```

---

2. Par *transformation syntaxique* on entend une simple réécriture des mots (ici les expressions rationnelles elles-mêmes) sans devoir faire appel à ce qu'ils représentent. Par exemple les règles de réécriture  $x+0 \rightsquigarrow x$  et  $0+x \rightsquigarrow x$  expriment *syntactiquement* la neutralité de la valeur du mot 0 pour l'opération représentée par + sans connaître ni cette valeur, ni cette opération.

recherche (et imprime) toute occurrence d'un mot du langage  $cha^*ine$  dans le fichier 'mon-texte'. Étant donné un motif exprimé sous la forme d'une expression rationnelle  $e$ , `grep` analyse le texte ligne par ligne, testant pour chaque ligne si elle appartient (ou non) au langage  $\Sigma^*(e)\Sigma^*$ ; l'alphabet (implicitement) sous-jacent étant l'alphabet ASCII (ou encore un jeu de caractères étendu tel que ISO Latin 1).

La syntaxe des expressions rationnelles permises par `grep` fait appel aux caractères '\*' et '|' pour noter respectivement les opérateurs  $\star$  et  $+$ . Ceci implique que, pour décrire un motif contenant le symbole '\*', il faudra prendre la précaution d'éviter qu'il soit interprété comme un opérateur, en le faisant précéder du caractère d'échappement '\'. Il en va de même pour les autres opérateurs ('|', '(', ')'). . . et donc aussi pour '\'. La syntaxe complète de `grep` inclut de nombreuses extensions notationnelles, permettant de simplifier grandement l'écriture des expressions rationnelles, au prix de la définition de nouveaux *caractères spéciaux*. Les plus importantes de ces extensions sont présentées dans le [tableau 3.2](#).

L'expression	dénote	remarque
Classes de caractères		
'[abc]'	$a + b + c$	$a, b, c$ sont des caractères
'[a-z]'	$a + b + c + \dots + z$	utilise l'ordre des caractères ASCII
'[^abc]'	$\Sigma \setminus \{a, b, c\}$	n'inclut pas le symbole de fin de ligne \n
'.'	$\Sigma$	n'importe quel symbole (autre que \n)
Quantificateurs		
'e?'	$\varepsilon + e$	
'e*'	$e^\star$	
'e+'	$ee^\star$	
'e{n}''	$e^n$	
'e{n,}''	$e^n e^\star$	
'e{,m}''	$\varepsilon + e + e^2 + \dots + e^m$	
'e{n,m}''	$e^n + e^{n+1} + \dots + e^m$	à condition que $n \leq m$
Ancres/Prédicats		
'\<e''	$e$	$e$ doit apparaître en début de mot, i.e. précédé d'un séparateur (espace, virgule, début de ligne, . . .)
'e\>''	$e$	$e$ doit apparaître en fin de mot, i.e. suivi d'un séparateur (espace, virgule, fin de ligne, . . .)
'^e''	$e$	$e$ doit apparaître en début de ligne
'e\$''	$e$	$e$ doit apparaître en fin de ligne
Caractères spéciaux		
'\.'''	.	
'\*'''	*	
'\+'''	+	
'\n''		dénote une fin de ligne
...		

TABLE 3.2 – Définition des motifs pour `grep`

Supposons, à titre illustratif, que nous cherchions à mesurer l'utilisation de l'imparfait du subjonctif dans les romans de Balzac, supposément disponibles dans le (volumineux) fichier 'Balzac.txt'. Pour commencer, un peu de conjugaison : quelles sont les terminaisons possibles ? Au premier groupe : *asse, asses, ât, assions, assiez, assent*. On trouvera donc toutes les formes du premier groupe avec un simple<sup>3</sup> :

```
> grep -E '(ât|ass(e|es|ions|iez|ent))' Balzac.txt
```

Guère plus difficile, le deuxième groupe : *isse, isses, ît, issions, issiez, issent*. D'où le nouveau motif :

```
> grep -E '([îâ]t|[ia]ss(e|es|ions|iez|ent))' Balzac.txt
```

Le troisième groupe est autrement complexe : disons simplement qu'il implique de considérer également les formes en *usse* (pour « boire » ou encore « valoir ») ; les formes en *insse* (pour « venir », « tenir » et leurs dérivés. . .). On parvient alors à quelque chose comme :

```
> grep -E '([îâû]n?t|[iau]n?ss(e|es|ions|iez|ent))' Balzac.txt
```

Cette expression est un peu trop générale, puisqu'elle inclut des séquences comme *unssiez* ; pour l'instant on s'en contentera. Pour continuer, revenons à notre ambition initiale : chercher des verbes. Il importe donc que les terminaisons que nous avons définies apparaissent bien comme des suffixes. Comment faire pour cela ? Imposer, par exemple, que ces séquences soient suivies par un caractère de ponctuation parmi : [ , ; . ! : ? ]. On pourrait alors écrire :

```
> grep -E '([îâû]n?t|[iau]n?ss(e|es|ions|iez|ent))[ , ; . ! : ? ]' Balzac.txt
```

indiquant que la terminaison verbale doit être suivie d'un des séparateurs. `grep` connaît même une notation un peu plus générale, utilisant : `[:punct:]`, qui comprend toutes les ponctuations et `[:space:]`, qui inclut tous les caractères d'espacement (blanc, tabulation. . .). Ce n'est pas satisfaisant, car nous exigeons alors la présence d'un séparateur, ce qui n'est pas le cas lorsque le verbe est simplement en fin de ligne dans le fichier.

Nous utiliserons donc `\>`, qui est une notation pour  $\epsilon$  lorsque celui-ci est trouvé à la fin d'un mot. La condition que la terminaison est bien en fin de mot s'écrit alors :

```
> grep -E '([îâû]n?t|[iau]n?ss(e|es|ions|iez|ent))\>' Balzac.txt
```

Dernier problème : réduire le bruit. Notre formulation est en effet toujours excessivement laxiste, puisqu'elle reconnaît des mots comme *masse* ou *passions*, qui ne sont pas des formes de l'imparfait du subjonctif. Une solution exacte est ici hors de question : il faudrait rechercher dans un dictionnaire tous les mots susceptibles d'être improprement décrits par cette expression : c'est possible (un dictionnaire est après tout fini), mais trop fastidieux. Une approximation raisonnable est d'imposer que la terminaison apparaisse sur un radical comprenant au moins trois lettres, soit finalement (en ajoutant également `\<` qui spécifie un début de mot) :

---

3. L'option '-E' donne accès à toutes les extensions notationnelles.

```
> grep -E \  
'\<[a-zéèîôûç]{3,}([fâû]n?t|[iau]n?ss(e|es|ions|iez|ent))\>' \  
Balzac.txt
```

D'autres programmes disponibles sur les machines UNIX utilisent ce même type d'extensions notationnelles, avec toutefois des variantes mineures suivant les programmes : c'est le cas en particulier de (f)lex, un générateur d'analyseurs lexicaux; de sed, un éditeur de flux de texte en batch; de perl, un langage de script pour la manipulation de fichiers textes; de (x)emacs. . . On se reportera aux pages de documentation de ces programmes pour une description précise des notations autorisées. Il existe également des bibliothèques permettant de manipuler des expressions rationnelles. Ainsi, pour ce qui concerne C, la bibliothèque *regex* permet de « compiler » des expressions rationnelles et de les « exécuter » pour tester la reconnaissance d'un mot. Des bibliothèques équivalentes existent en C++, en Java. . .

**Attention** Une confusion fréquente à éviter : pour exprimer des ensembles de noms de fichiers les shells UNIX utilisent des notations comparables, mais avec une sémantique différente. Par exemple, 'foo\*' désigne les fichiers dont le nom a foo pour préfixe (comme par exemple 'foo.java') et non pas ceux dont le nom appartient au langage  $fo(o^*)$ .

# Chapitre 4

## Automates finis

Nous introduisons ici très succinctement les automates finis. Pour les lecteurs intéressés par les aspects formels de la théorie des automates finis, nous recommandons particulièrement la lecture de quelques chapitres de [Hopcroft et Ullman \(1979\)](#), ou en français, des chapitres initiaux de [Sakarovitch \(2003\)](#). L'exposé nécessairement limité présenté dans les sections qui suivent reprend pour l'essentiel le contenu de [Sudkamp \(1997\)](#).

### 4.1 Automates finis

#### 4.1.1 Bases

Dans cette section, nous introduisons le modèle le plus simple d'automate fini : l'automate déterministe complet. Ce modèle nous permet de définir les notions de calcul et de langage associé à un automate. Nous terminons cette section en définissant la notion d'équivalence entre automates, ainsi que la notion d'utilité d'un état.

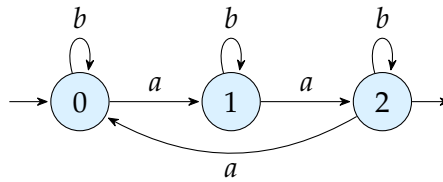
**Définition 4.1** (Automate fini déterministe (complet)). *Un automate fini déterministe (complet) (DFA, deterministic finite automaton) est défini par un quintuplet  $A = (\Sigma, Q, q_0, F, \delta)$ , où :*

- $\Sigma$  est un ensemble fini de symboles (l'alphabet)
- $Q$  est un ensemble fini d'états
- $q_0 \in Q$  est l'état initial
- $F \subset Q$  est l'ensemble des états finaux
- $\delta$  est une fonction totale de  $Q \times \Sigma$  dans  $Q$ , appelée fonction de transition.

Le qualificatif de *déterministe* sera justifié lors de l'introduction des automates *non-déterministes* ([définition 4.8](#)).

Un automate fini correspond à un graphe orienté, dans lequel certains des nœuds (états) sont distingués et marqués comme initial ou finaux et dans lequel les arcs (*transitions*) sont étiquetés par des symboles de  $\Sigma$ . Une transition est donc un triplet de  $Q \times \Sigma \times Q$ ; si  $\delta(q, a) = r$ , on dit que  $a$  est l'*étiquette* de la transition  $(q, a, r)$ ;  $q$  en est l'*origine*, et  $r$  la *destination*. Les automates admettent une représentation graphique, comme celle de l'[automate 4.1](#).

---



Automate 4.1 – Un automate fini (déterministe)

Dans cette représentation, l'état initial 0 est marqué par un arc entrant sans origine et les états finaux (ici l'unique état final est 2) par un arc sortant sans destination. La fonction de transition correspondant à ce graphe s'exprime matriciellement par :

$\delta$	$a$	$b$
0	1	0
1	2	1
2	0	2

Pour définir un automate dans Vcsn, lister ligne par ligne les transitions, avec la syntaxe '*source -> destination étiquette, étiquette...*'. Les états initiaux sont dénotés par '\$ -> 0', et les finaux par '2 -> \$'.

L'automate 4.1 s'écrit donc :

```
>>> a = vcsn.automaton(''
context = "lal_char(ab), b"
$ -> 0
0 -> 0 b
0 -> 1 a
1 -> 1 b
1 -> 2 a
2 -> 2 b
2 -> 0 a
2 -> $
''')
```

Un *calcul* dans  $A$  est une séquence de transitions  $e_1 \dots e_n$  de  $A$ , telle que pour tout couple de transitions successives  $e_i, e_{i+1}$ , l'état destination de  $e_i$  est l'état origine de  $e_{i+1}$ . L'*étiquette d'un calcul* est le mot construit par concaténation des étiquettes de chacune des transitions. Un calcul dans  $A$  est *réussi* si la première transition a pour état d'origine l'état initial, et si la dernière transition a pour destination un des états finaux. Le langage *reconnu* par l'automate  $A$ , noté  $L(A)$ , est l'ensemble des étiquettes des calculs réussis. Dans l'exemple précédent, le mot *baab* appartient au langage reconnu, puisqu'il étiquette le calcul (réussi) :  $(0, b, 0)(0, a, 1), (1, a, 2)(2, b, 2)$ .

La relation  $\vdash_A$  permet de formaliser la notion d'étape élémentaire de calcul. Ainsi on écrira, pour  $a$  dans  $\Sigma$  et  $v$  dans  $\Sigma^*$  :  $(q, av) \vdash_A (\delta(q, a), v)$  pour noter une étape de calcul utilisant la transition  $(q, a, \delta(q, a))$ . La clôture réflexive et transitive de  $\vdash_A$  se note  $\vdash_A^*$  ;  $(q, uv) \vdash_A^* (p, v)$  s'il existe une suite d'états  $q = q_1 \dots q_n = p$  tels que  $(q_1, u_1 \dots u_n v) \vdash_A (q_2, u_2 \dots u_n v) \dots \vdash_A (q_n, v)$ . La réflexivité de cette relation signifie que pour tout  $(q, u)$ ,  $(q, u) \vdash_A^* (q, u)$ . Avec ces notations, on a :

$$L(A) = \{u \in \Sigma^* \mid (q_0, u) \vdash_A^* (q, \varepsilon), \text{ avec } q \in F\}$$

Cette notation met en évidence l'automate comme une machine permettant de *reconnaître* des mots : tout parcours partant de  $q_0$  permet de « consommer » un à un les symboles du mot à reconnaître ; ce processus stoppe lorsque le mot est entièrement consommé : si l'état ainsi atteint est final, alors le mot appartient au langage reconnu par l'automate.

On dérive un algorithme permettant de tester si un mot appartient au langage reconnu par un automate fini déterministe.

```
// u = u1...un est le mot à reconnaître.
// A = (Σ, Q, q0, δ, F) est le DFA.
q := q0;
for i := 1 to n do
  | q := δ(q, ui)
if q ∈ F then return true else return false ;
```

Algorithme 4.2 – Reconnaissance par un DFA

La complexité de cet algorithme découle de l'observation que chaque étape de calcul correspond à une application de la fonction  $\delta()$ , qui elle-même se réduit à la lecture d'une case d'un tableau et une affectation, deux opérations qui s'effectuent en temps constant. La reconnaissance d'un mot  $u$  se calcule en exactement  $|u|$  étapes.

**Définition 4.2** (Langage reconnaissable). *Un langage est reconnaissable s'il existe un automate fini qui le reconnaît.*



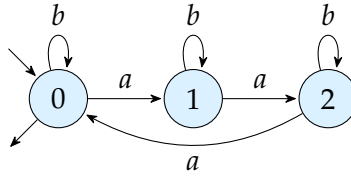
Dans Vcsn, un automate peut-être vu comme une fonction qui calcule un booléen représentant l'appartenance du mot au langage de l'automate :

```
>>> a('baab')
>>> a('a')
```

La routine '[automaton.shortest](#)' permet d'obtenir la liste des premiers mots acceptés.

```
>>> a.shortest(10)
```

L'[automate 4.3](#) est un exemple très similaire au premier. L'état 0 est à la fois initial et final. Il reconnaît le langage correspondant aux mots  $u$  tels que  $|u|_a$  est divisible par 3 : chaque



Automate 4.3 – Un automate fini déterministe comptant les  $a$  (modulo 3)

état correspond à une valeur du reste dans la division par 3 : un calcul réussi correspond nécessairement à un reste égal à 0 et réciproquement.

Par un raisonnement similaire à celui utilisé pour définir un calcul de longueur quelconque, il est possible d'étendre récursivement la fonction de transition  $\delta$  en une fonction  $\delta^*$  de  $Q \times \Sigma^* \rightarrow Q$  par :

- $\delta^*(q, \varepsilon) = q$
- $\delta^*(q, au) = \delta^*(\delta(q, a), u)$

On notera que, puisque  $\delta$  est une fonction totale,  $\delta^*$  est également une fonction totale : l'image d'un mot quelconque de  $\Sigma^*$  par  $\delta^*$  est toujours bien définie, i.e. existe et est unique. Cette nouvelle notation permet de donner une notation alternative pour le langage reconnu par un automate  $A$  :

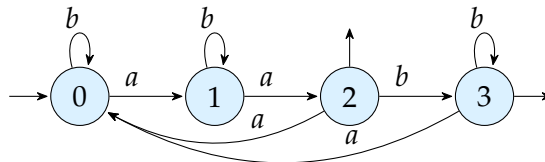
$$L(A) = \{u \in \Sigma^* \mid \delta^*(q_0, u) \in F\}$$

Nous avons pour l'instant plutôt vu l'automate comme une machine permettant de reconnaître des mots. Il est également possible de le voir comme un système de *production* : partant de l'état initial, tout parcours conduisant à un état final construit itérativement une séquence d'étiquettes par concaténation des étiquettes rencontrées le long des arcs.

Si chaque automate fini reconnaît un seul langage, la réciproque n'est pas vraie : plusieurs automates peuvent reconnaître le même langage. Comme pour les expressions rationnelles, on dira dans ce cas que les automates sont *équivalents*.

**Définition 4.3** (Automates équivalents). *Deux automates finis  $A_1$  et  $A_2$  sont équivalents si et seulement s'ils reconnaissent le même langage.*

Ainsi, par exemple, l'[automate 4.4](#) est-il équivalent à l'[automate 4.1](#) : tous deux reconnaissent le langage de tous les mots qui contiennent un nombre de  $a$  congru à 2 modulo 3.

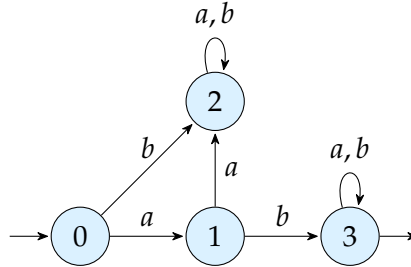


Automate 4.4 – Un automate fini déterministe équivalent à l'[automate 4.1](#)

Nous l'avons noté plus haut,  $\delta^*$  est définie pour tout mot de  $\Sigma^*$ . Ceci implique que l'algorithme de reconnaissance ([algorithme 4.2](#)) demande exactement  $|u|$  étapes de calcul, correspondant à une exécution complète de la boucle. Ceci peut s'avérer particulièrement inefficace,



comme dans l'exemple de l'automate 4.5, qui reconnaît le langage  $\{ab\}^* \{a, b\}^*$ . Dans ce cas en effet, il est en fait possible d'accepter ou de rejeter des mots en ne considérant que les deux premiers symboles.



Automate 4.5 – Un automate fini déterministe pour  $ab(a + b)^*$

### 4.1.2 Spécification partielle

Pour contourner ce problème et pouvoir arrêter le calcul aussi tôt que possible, nous introduisons dans cette section des définitions alternatives, mais qui s'avèrent en fait équivalentes, des notions d'automate et de calcul.

**Définition 4.4** (Automate fini déterministe). *Un automate fini déterministe est un quintuplet  $A = (\Sigma, Q, q_0, F, \delta)$ , où :*

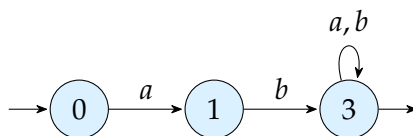
- $\Sigma$  est un ensemble fini de symboles (l'alphabet)
- $Q$  est un ensemble fini d'états
- $q_0 \in Q$  est l'état initial
- $F \subset Q$  sont les états finaux
- $\delta$  est une fonction partielle de  $Q \times \Sigma$  dans  $Q$

La différence avec la définition 4.1 est que  $\delta$  est ici définie comme une fonction partielle. Son domaine de définition est un sous-ensemble de  $Q \times \Sigma$ . Selon cette nouvelle définition, il est possible de se trouver dans une situation où un calcul s'arrête avant d'avoir atteint la fin de l'entrée. Ceci se produit dès que l'automate atteint une configuration  $(q, au)$  pour laquelle il n'existe pas de transition d'origine  $q$  étiquetée par  $a$ .

La définition 4.4 est en fait strictement équivalente à la précédente, dans la mesure où les automates partiellement spécifiés peuvent être complétés par ajout d'un état puits absorbant les transitions absentes de l'automate original, sans pour autant changer le langage reconnu. Formellement, soit  $A = (\Sigma, Q, q_0, F, \delta)$  un automate partiellement spécifié, on définit  $A' = (\Sigma, Q', q'_0, F', \delta')$  avec :

- $Q' = Q \cup \{q_p\}$
- $q'_0 = q_0$
- $F' = F$
- $\forall q \in Q, a \in \Sigma, \delta'(q, a) = \delta(q, a)$  si  $\delta(q, a)$  existe,  $\delta'(q, a) = q_p$  sinon.
- $\forall a \in \Sigma, \delta'(q_p, a) = q_p$

L'état puits,  $q_p$ , est donc celui dans lequel on aboutit dans  $A'$  en cas d'échec dans  $A$  ; une fois dans  $q_p$ , il est impossible d'atteindre les autres états de  $A$  et donc de rejoindre un état final. Cette transformation est illustrée pour l'automate 4.6, dont le transformé est précisément l'automate 4.5, l'état 2 jouant le rôle de puits.



Automate 4.6 – Un automate partiellement spécifié

$A'$  reconnaît le même langage que  $A$  puisque :

- si  $u \in L(A)$ ,  $\delta^*(q_0, u)$  existe et appartient à  $F$ . Dans ce cas, le même calcul existe dans  $A'$  et aboutit également dans un état final
- si  $u \notin L(A)$ , deux cas sont possibles : soit  $\delta^*(q_0, u)$  existe mais n'est pas final, et la même chose se produit dans  $A'$  ; soit le calcul s'arrête dans  $A$  après le préfixe  $v$  : on a alors  $u = va$  et  $\delta(\delta^*(q_0, v), a)$  n'existe pas. Or, le calcul correspondant dans  $A'$  conduit au même état, à partir duquel une transition existe vers  $q_p$ . Dès lors, l'automate est voué à rester dans cet état jusqu'à la fin du calcul ; cet état n'étant pas final, le calcul échoue et la chaîne est rejetée.

Pour tout automate (au sens de la définition 4.4), il existe donc un automate complètement spécifié (ou *automate complet*) équivalent.

**Exercice 4.5** (BCI-0506-1). On dit qu'un langage  $L$  sur  $\Sigma^*$  est local s'il existe  $I$  et  $F$  deux sous-ensembles de  $\Sigma$  et  $T$  un sous-ensemble de  $\Sigma^2$  (l'ensemble des mots sur  $\Sigma$  de longueur 2) tels que  $L = (I\Sigma^* \cap \Sigma^*F) \setminus (\Sigma^*T\Sigma^*)$ . En d'autres termes, un langage local est défini par un ensemble fini de préfixes et de suffixes de longueur 1 ( $I$  et  $F$ ) et par un ensemble  $T$  de « facteurs interdits » de longueur 2. Les langages locaux jouent en particulier un rôle important pour inférer un langage  $L$  à partir d'un ensemble d'exemples de mots de  $L$ .

1. Montrer que le langage  $L_1$  dénoté par  $aa^*$  est un langage local ; que le langage  $L_2$  dénoté par  $ab(ab)^*$  est un langage local. Dans les deux cas, on prendra  $\Sigma = \{a, b\}$ .
2. Montrer que le langage  $L_3$  dénoté par  $aa^* + ab(ab)^*$  n'est pas local.
3. Montrer que l'intersection de deux langages locaux est un langage local ; que l'union de deux langages locaux n'est pas nécessairement un langage local.
4. Montrer que tout langage local est rationnel.
5. On considère maintenant que  $\Sigma = \{a, b, c\}$ . Soit  $C$  l'ensemble fini suivant  $C = \{aab, bca, abc\}$ . Proposer et justifier une construction pour l'automate reconnaissant le plus petit (au sens de l'inclusion) langage local contenant tous les mots de  $C$ . Vous pourrez, par exemple, construire les ensembles  $I$ ,  $F$  et  $T$  et en déduire la forme de l'automate.
6. On rappelle qu'un morphisme est une application  $\phi$  de  $\Sigma_1^*$  vers  $\Sigma_2^*$  telle que (i)  $\phi(\varepsilon) = \varepsilon$  et (ii) si  $u$  et  $v$  sont deux mots de  $\Sigma_1^*$ , alors  $\phi(uv) = \phi(u)\phi(v)$ . Un morphisme lettre-à-lettre est induit par une injection  $\phi$  de  $\Sigma_1$  dans  $\Sigma_2$ , et est étendu récursivement par  $\phi(au) = \phi(a)\phi(u)$ . Par exemple, pour  $\phi(a) = \phi(b) = x, \phi(c) = y$  on a  $\phi(abccb) = xxyyx$ .

Montrer que tout langage rationnel  $L$  est l'image par un morphisme lettre-à-lettre d'un langage local  $L'$ .

Indication : Si  $A = (\Sigma, Q, q_0, F, \delta)$  est un automate fini déterministe reconnaissant  $L$ , on définira le langage local  $L'$  sur l'alphabet dont les symboles sont les triplets  $[p, a, q]$ , avec  $p, q \in Q$  et  $a \in \Sigma$  et on considérera le morphisme induit par  $\phi([p, a, q]) = a$ .

### 4.1.3 États utiles

Un second résultat concernant l'équivalence entre automates demande l'introduction des quelques définitions complémentaires suivantes.

**Définition 4.6** (Accessible, co-accessible, utile, émondé). Un état  $q$  de  $A$  est dit accessible s'il existe  $u$  dans  $\Sigma^*$  tel que  $\delta^*(q_0, u) = q$ .  $q_0$  est trivialement accessible (par  $u = \varepsilon$ ). Un automate dont tous les états sont accessibles est lui-même dit accessible.

Un état  $q$  de  $A$  est dit co-accessible s'il existe  $u$  dans  $\Sigma^*$  tel que  $\delta^*(q, u) \in F$ . Tout état final est trivialement co-accessible (par  $u = \varepsilon$ ). Un automate dont tous les états sont co-accessibles est lui-même dit co-accessible.

Un état  $q$  de  $A$  est dit utile s'il est à la fois accessible et co-accessible. D'un automate dont tous les états sont utiles on dit qu'il est émondé (en anglais *trim*).

Les états utiles sont donc les états qui servent dans au moins un calcul réussi : on peut les atteindre depuis l'état initial, et les quitter pour un état final. Les autres états, les états inutiles, ne servent pas à grand-chose, en tout cas pas au peuplement de  $L(A)$ . C'est précisément ce que montre le théorème suivant.

**Théorème 4.7** (Émondage). Si  $L(A) \neq \emptyset$  est un langage reconnaissable, alors il est également reconnu par un automate émondé.

*Démonstration.* Soit  $A$  un automate reconnaissant  $L$ , et  $Q_u \subset Q$  l'ensemble de ses états utiles ;  $Q_u$  n'est pas vide dès lors que  $L(A) \neq \emptyset$ . La restriction  $\delta'$  de  $\delta$  à  $Q_u$  permet de définir un automate  $A' = (\Sigma, Q_u, q_0, F, \delta')$ .  $A'$  est équivalent à  $A$ .  $Q_u$  étant un sous-ensemble de  $Q$ , on a en effet immédiatement  $L(A') \subset L(A)$ . Soit  $u$  dans  $L(A)$ , tous les états du calcul qui le reconnaît étant par définition utiles, ce calcul existe aussi dans  $A'$  et aboutit dans un état final :  $u$  est donc aussi reconnu par  $A'$ .  $\square$

### 4.1.4 Automates non-déterministes

Dans cette section, nous augmentons le modèle d'automate de la [définition 4.4](#) en autorisant plusieurs transitions sortantes d'un état  $q$  à porter le même symbole : les automates ainsi spécifiés sont dits *non-déterministes*. Nous montrons que cette généralisation n'augmente toutefois pas l'expressivité du modèle : les langages reconnus par les automates non-déterministes sont les mêmes que ceux reconnus par les automates déterministes.

### Non-déterminisme

**Définition 4.8** (Automate fini non-déterministe). *Un automate fini non-déterministe (NFA, nondeterministic finite automaton) est défini par un quintuplet  $A = (\Sigma, Q, I, F, \delta)$ , où :*

- $\Sigma$  est un ensemble fini de symboles (l'alphabet)
- $Q$  est un ensemble fini d'états
- $I \subset Q$  sont les états initiaux
- $F \subset Q$  sont les états finaux
- $\delta \subset Q \times \Sigma \times Q$  est une relation ; chaque triplet  $(q, a, q') \in \delta$  est une transition.

On peut également considérer  $\delta$  comme une fonction de  $Q \times \Sigma$  dans  $2^Q$  : l'image par  $\delta$  d'un couple  $(q, a)$  est un sous-ensemble de  $Q$  (éventuellement  $\emptyset$  lorsqu'il n'y a pas de transition étiqueté  $a$  sortant de  $q$ ). On aurait aussi pu, sans perte de généralité, n'autoriser qu'un seul état initial.

La nouveauté introduite par cette définition est l'indétermination qui porte sur les transitions : pour une paire  $(q, a)$ , il peut exister dans  $A$  plusieurs transitions possibles. On parle, dans ce cas, de *non-déterminisme*, signifiant qu'il existe des états dans lesquels la lecture d'un symbole  $a$  dans l'entrée provoque un choix (ou une indétermination) et que plusieurs transitions alternatives sont possibles.

Notez que cette définition généralise proprement la notion d'automate fini : la [définition 4.4](#) est un cas particulier de la [définition 4.8](#), avec pour tout  $(q, a)$ , l'ensemble  $\delta(q, a)$  ne contient qu'un seul élément. En d'autres termes, tout automate déterministe est non-déterministe. Remarquez que le vocabulaire est très trompeur : être *non-déterministe* ne signifie pas ne pas être déterministe ! C'est pour cela que nous notons « non-déterministe » avec un tiret, et non pas « non déterministe ».

Les notions de *calcul* et de *calcul réussi* se définissent exactement comme dans le cas déterministe. On définit également la fonction de transition étendue  $\delta^*$  de  $Q \times \Sigma^*$  dans  $2^Q$  par :

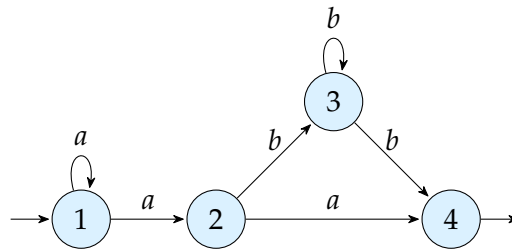
- $\delta^*(q, \varepsilon) = \{q\}$
- $\delta^*(q, au) = \bigcup_{r \in \delta(q, a)} \delta^*(r, u)$

L'[automate 4.7](#) illustre ces notions.

Le langage reconnu par un automate non-déterministe est défini par :

$$L(A) = \{u \in \Sigma^* \mid \exists q_0 \in I : \delta^*(q_0, u) \cap F \neq \emptyset\}$$

Pour qu'un mot appartienne au langage reconnu par l'automate, il suffit qu'il existe, parmi tous les calculs possibles, un calcul réussi, c'est-à-dire un calcul qui consomme tous les symboles de  $u$  entre un état initial et un état final ; la reconnaissance n'échoue donc que si *tous les calculs* aboutissent à une des situations d'échec. Ceci implique que pour calculer l'appartenance d'un mot à un langage, il faut, en principe, examiner successivement tous les chemins possibles, et donc éventuellement revenir en arrière dans l'exploration des parcours de l'automate lorsque l'on rencontre une impasse. Cette exploration peut toutefois se ramener à un problème d'accessibilité dans le graphe sous-jacent à l'automate, puisqu'un mot est



Deux transitions sortantes de 1 sont étiquetées par  $a$  :  $\delta(1, a) = \{1, 2\}$ .  $aa$  donne lieu à un calcul réussi passant successivement par 1, 2 et 4, qui est final ;  $aa$  donne aussi lieu à un calcul  $(1, a, 1)(1, a, 1)$ , qui n'est pas un calcul réussi.

Automate 4.7 – Un automate non-déterministe

reconnu par l'automate si l'on arrive à prouver qu'au moins un état final (ils sont en nombre fini) est accessible depuis au moins un état initial (ils sont en nombre fini) sur un chemin étiqueté par  $u$ . Dans ce nouveau modèle, le problème de la reconnaissance d'un mot reste donc polynomial en la longueur du mot en entrée.

### Le non-déterminisme ne paye pas

La généralisation du modèle d'automate fini liée à l'introduction de transitions non-déterministes est, du point de vue des langages reconnus, sans effet : tout langage reconnu par un automate fini non-déterministe est aussi reconnu par un automate déterministe.

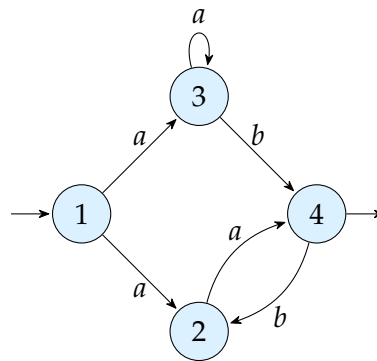
**Théorème 4.9** (Déterminisation). *Pour tout NFA  $A$ , on peut construire un DFA  $A'$  équivalent à  $A$ . De plus, si  $A$  a  $n$  états, alors  $A'$  a au plus  $2^n$  états.*

On pose  $A = (\Sigma, Q, I, F, \delta)$  et on considère  $A'$  défini par  $A' = (\Sigma, 2^Q, I, F', \delta')$  avec :

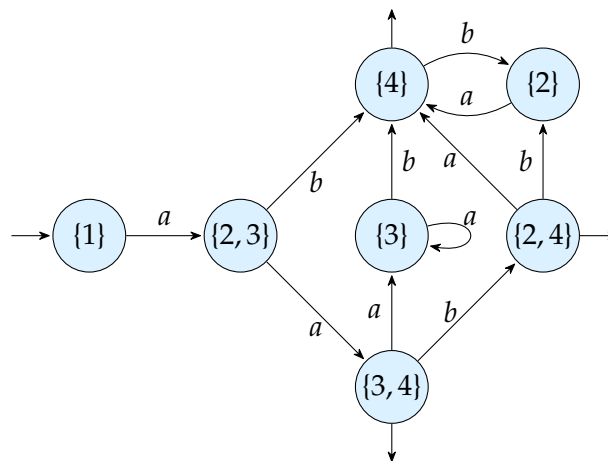
- $F' = \{G \subset Q \mid F \cap G \neq \emptyset\}$ .
- $\forall G \subset Q, \forall a \in \Sigma, \delta'(G, a) = \bigcup_{q \in G} \delta(q, a)$ .

Les états de  $A'$  sont donc associés de manière biunivoque à des sous-ensembles de  $Q$  (il y en a un nombre fini) : l'état initial est l'ensemble  $I$  ; chaque partie contenant un état final de  $A$  donne lieu à un état final de  $A'$  ; la transition sortante d'un sous-ensemble  $E$ , étiquetée par  $a$ , atteint l'ensemble de tous les états de  $Q$  atteignables depuis un état de  $E$  par une transition étiquetée par  $a$ .  $A'$  est le *déterminisé* de  $A$ . Illustrons cette construction sur l'[automate 4.8](#).

L'[automate 4.8](#) ayant 4 états, son déterminisé en aura donc 16, correspondant au nombre de sous-ensembles de  $\{1, 2, 3, 4\}$ . Son état initial est le singleton  $\{1\}$ , et ses états finaux tous les sous-ensembles contenant 4 : il y en a exactement 8, qui sont :  $\{4\}$ ,  $\{1, 4\}$ ,  $\{2, 4\}$ ,  $\{3, 4\}$ ,  $\{1, 2, 4\}$ ,  $\{1, 3, 4\}$ ,  $\{2, 3, 4\}$ ,  $\{1, 2, 3, 4\}$ . Considérons par exemple les transitions sortantes de l'état initial : 1 ayant deux transitions sortantes sur le symbole  $a$ ,  $\{1\}$  aura une transition depuis  $a$  vers l'état correspondant au doubleton  $\{2, 3\}$ . Le déterminisé est l'[automate 4.9](#). On notera que cette figure ne représente que les états *utiles* du déterminisé : ainsi  $\{1, 2\}$  n'est pas représenté, puisqu'il n'existe aucun moyen d'atteindre cet état.



Automate 4.8 – Un automate à déterminer



Automate 4.9 – Le résultat de la détermination de l'automate 4.8

Que se passerait-il si l'on ajoutait à l'automate 4.8 une transition supplémentaire bouclant dans l'état 1 sur le symbole  $a$ ? Construisez le déterminisé de ce nouvel automate.

Démontrons maintenant le théorème 4.9; et pour saisir le sens de la démonstration, reportons nous à l'automate 4.8, et considérons les calculs des mots préfixés par  $aaa$  : le premier  $a$  conduit à une indétermination entre 2 et 3; suivant les cas, le second  $a$  conduit donc en 4 (si on a choisi d'aller initialement en 2) ou en 3 (si on a choisi d'aller initialement en 3). La lecture du troisième  $a$  lève l'ambiguïté, puisqu'il n'y a pas de transition sortante pour 4 : le seul état possible après  $aaa$  est 3. C'est ce qui se lit sur l'automate 4.9 : les ambiguïtés initiales correspondent aux états  $\{2, 3\}$  (atteint après le premier  $a$ ) et  $\{3, 4\}$  (atteint après le second  $a$ ); après le troisième le doute n'est plus permis et l'état atteint correspond au singleton  $\{3\}$ . Formalisons maintenant ces idées pour démontrer le résultat qui nous intéresse.

*Démonstration du théorème 4.9.* Première remarque :  $A'$  est un automate fini déterministe, puisque l'image par  $\delta'$  d'un couple  $(H, a)$  est uniquement définie.

Nous allons ensuite montrer que tout calcul dans  $A$  correspond à exactement un calcul dans

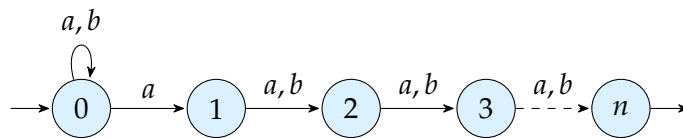
$A'$ , soit formellement que :

$$\begin{array}{ll} \text{si } \exists q_0 \in I : (q_0, u) \vdash_A^* (p, \varepsilon) & \text{alors } \exists G \subset Q : p \in G, (\{I\}, u) \vdash_{A'}^* (G, \varepsilon) \\ \text{si } (\{I\}, u) \vdash_{A'}^* (G, \varepsilon) & \text{alors } \exists q_0 \in I : \forall p \in G : (q_0, u) \vdash_A^* (p, \varepsilon) \end{array}$$

Opérons une récurrence sur la longueur de  $u$  : si  $u$  est égal à  $\varepsilon$  le résultat est vrai par définition de l'état initial dans  $A'$ . Supposons que le résultat est également vrai pour tout mot de longueur strictement inférieure à  $|u|$ , et considérons  $u = va$ . Soit  $(q_0, va) \vdash_A^* (p, a) \vdash_A (q, \varepsilon)$  un calcul dans  $A$  : par l'hypothèse de récurrence, il existe un calcul dans  $A'$  tel que :  $(\{I\}, v) \vdash_{A'}^* (G, \varepsilon)$ , avec  $p \in G$ . Ceci implique, en vertu de la définition même de  $\delta'$ , que  $q$  appartient à  $H = \delta'(G, a)$ , et donc que  $(\{I\}, u = va) \vdash_{A'}^* (H, \varepsilon)$ , avec  $q \in H$ . Inversement, soit  $(\{I\}, u = va) \vdash_{A'}^* (G, a) \vdash_{A'} (H, \varepsilon)$  : pour tout  $p$  dans  $G$  il existe un calcul dans  $A$  tel que  $(q_0, v) \vdash_A^* (p, \varepsilon)$ .  $G$  ayant une transition étiquetée par  $a$ , il existe également dans  $G$  un état  $p$  tel que  $\delta(p, a) = q$ , avec  $q \in H$ , puis que  $(q_0, u = va) \vdash_A^* (q, \varepsilon)$ , avec  $q \in H$ . On déduit alors que l'on a  $(q_0, u = va) \vdash_A^* (q, \varepsilon)$ , avec  $q \in F$  si et seulement si  $(\{I\}, u) \vdash_{A'}^* (G, \varepsilon)$  avec  $q \in G$ , donc avec  $F \cap G \neq \emptyset$ , soit encore  $G \in F'$ . Il s'ensuit directement que  $L(A) = L(A')$ .  $\square$

La construction utilisée pour construire un DFA équivalent à un NFA s'appelle la *construction des sous-ensembles* : elle se traduit directement dans un algorithme permettant de construire le déterminisé d'un automate quelconque. On notera que cette construction peut s'organiser de telle façon à ne considérer que les états accessibles du déterminisé. Il suffit, pour cela, de construire de proche en proche depuis  $\{I\}$ , les états accessibles, résultant en général à des automates (complets) ayant moins de  $2^n$  états.

Il existe toutefois des automates pour lesquels l'explosion combinatoire annoncée a lieu, comme l'[automate 4.10](#). Sauriez-vous expliquer d'où vient cette difficulté ? Quel est le langage reconnu par cet automate ?



Automate 4.10 – Un automate difficile à déterminer

Dans la mesure où ils n'apportent aucun gain en expressivité, il est permis de se demander à quoi servent les automates non-déterministes. Au moins à deux choses : ils sont plus faciles à construire à partir d'autres représentations des langages et sont donc utiles pour certaines preuves (voir plus loin) ou algorithmes. Ils fournissent également des machines bien plus (dans certains cas exponentiellement plus) « compactes » que les DFA, ce qui n'est pas une propriété négligeable.

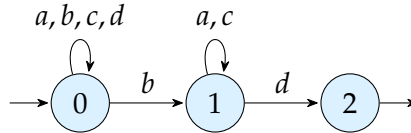


Pour déterminer un automate, utilisez '[automaton.determinize](#)'. Observez que les états sont étiquetés par les états de l'automate d'origine. Utilisez '[automaton.strip](#)' pour éliminer ces décorations.

La routine '[context.de\\_bruijn](#)' génère les automates de la famille de l'[automate 4.10](#).

Lancez `vcsn.context('lal_char(abc), b').de_bruijn(3).determinize().strip()`, puis augmentez l'argument de `'de_bruijn'`.

**Exercice 4.10** (BCI-0203). On considère  $A = (\Sigma, Q, 0, \{2\}, \delta)$ , l'automate 4.11.



Automate 4.11 – Un automate potentiellement à déterminer

1.  $A$  est-il déterministe ? Justifiez votre réponse.
2. Quel est le langage reconnu par  $A$  ?
3. Donnez, sous forme graphique, l'automate déterministe  $A'$  équivalent à  $A$ , en justifiant votre construction.

#### 4.1.5 Transitions spontanées

Il est commode, dans la pratique, de disposer d'une définition encore plus plastique de la notion d'automate fini, en autorisant des transitions étiquetées par le mot vide, qui sont appelées les *transitions spontanées*.

**Définition 4.11** (Automate fini à transitions spontanées). *Un automate fini (non-déterministe) à transitions spontanées ( $\varepsilon$ -NFA, nondeterministic finite automaton with  $\varepsilon$  moves) est défini par un quintuplet  $A = (\Sigma, Q, I, F, \delta)$ , où :*

- $\Sigma$  est un ensemble fini de symboles (l'alphabet)
- $Q$  est un ensemble fini d'états
- $I \subset Q$  est l'ensemble des états initiaux
- $F \subset Q$  est l'ensemble des états finaux
- $\delta \subset Q \times (\Sigma \cup \{\varepsilon\}) \times Q$  est la relation de transition.

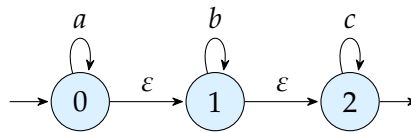
Encore une fois, on aurait pu ne conserver qu'un seul état initial, et définir  $\delta$  comme une fonction (partielle) de  $Q \times (\Sigma \cup \{\varepsilon\})$  dans  $2^Q$ .

Formellement, un automate non-déterministe avec transitions spontanées se définit comme un NFA à la différence près que  $\delta$  permet d'étiqueter les transitions par  $\varepsilon$ . Les transitions spontanées permettent d'étendre la notion de calcul :  $(q, u) \vdash_A (p, v)$  si (i)  $u = av$  et  $(q, a, p) \in \delta$  ou bien (ii)  $u = v$  et  $(q, \varepsilon, p) \in \delta$ . En d'autres termes, dans un  $\varepsilon$ -NFA, il est possible de changer d'état sans consommer de symbole, en empruntant une transition étiquetée par le mot vide. Le langage reconnu par un  $\varepsilon$ -NFA  $A$  est, comme précédemment, défini par

$$L(A) = \{u \in \Sigma^* \mid (q_0, u) \vdash_A^* (q, \varepsilon) \text{ avec } q_0 \in I, q \in F\}$$

L'automate 4.12 est un exemple d' $\varepsilon$ -NFA.



Automate 4.12 – Un automate avec transitions spontanées correspondant à  $a^*b^*c^*$ 

Cette nouvelle extension n'ajoute rien à l'expressivité du modèle, puisqu'il est possible de transformer chaque  $\varepsilon$ -NFA  $A$  en un NFA équivalent. Pour cela, nous introduisons tout d'abord la notion d' $\varepsilon$ -fermeture d'un état  $q$ , correspondant à tous les états accessibles depuis  $q$  par une ou plusieurs transitions spontanées. Formellement :

**Définition 4.12** ( $\varepsilon$ -fermeture d'un état). Soit  $q$  un état de  $Q$ . On appelle  $\varepsilon$ -fermeture (en anglais *closure*) de  $q$  l'ensemble  $\varepsilon\text{-closure}(q) = \{p \in Q \mid (q, \varepsilon) \vdash_A^* (p, \varepsilon)\}$ . Par construction,  $q \in \varepsilon\text{-closure}(q)$ .

Intuitivement, la fermeture d'un état  $q$  contient tous les états qu'il est possible d'atteindre depuis  $q$  sans consommer de symboles. Ainsi, la fermeture de l'état 0 de l'automate 4.12 est-elle égale à  $\{0, 1, 2\}$ .

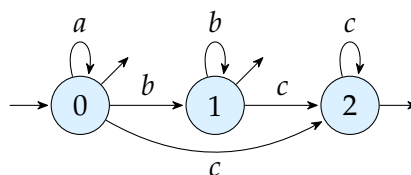
**Théorème 4.13.** Pour tout  $\varepsilon$ -NFA  $A$ , il existe un NFA  $A'$  tel que  $L(A) = L(A')$ .

*Démonstration.* En posant  $A = (\Sigma, Q, q_0, F, \delta)$ , on définit  $A'$  comme  $A' = (\Sigma, Q, q_0, F', \delta')$  avec :

$$F' = \{q \in Q \mid \varepsilon\text{-closure}(q) \cap F \neq \emptyset\} \quad \delta'(q, a) = \bigcup_{p \in \varepsilon\text{-closure}(q)} \delta(p, a)$$

Par une récurrence similaire à la précédente, on montre alors que tout calcul  $(q_0, u) \vdash_A^* p$  est équivalent à un calcul  $(q_0, u) \vdash_{A'}^* p'$ , avec  $p \in \varepsilon\text{-closure}(p')$ , puis que  $L(A) = L(A')$ .  $\square$

On déduit directement un algorithme constructif pour supprimer, à nombre d'états constant, les transitions spontanées. Appliqué à l'automate 4.12, cet algorithme construit l'automate 4.13.




Automate 4.13 – L'automate 4.12 débarrassé de ses transitions spontanées

Dans le cas de la preuve précédente on parle d' $\varepsilon$ -fermeture arrière d'un automate (on pourrait dire « amont ») : la fonction  $\delta$  « commence » (en amont) par effectuer les éventuelles transitions spontanées, puis enchaîne par une transition sur lettre. De même tout état (amont) depuis lequel on arrive à un état final en « descendant » les transitions spontanées devient lui-même final.

On peut également introduire l' $\varepsilon$ -fermeture avant d'un  $\varepsilon$ -NFA (on pourrait dire « aval ») : on poursuit toute transition non-spontanée par toutes les transitions spontanées (en aval), et deviennent initiaux tous les états en aval (par transitions spontanées) d'un état initial.

On préfère en général la fermeture arrière, qui n'introduit pas de nouveaux états initiaux.

Les transitions spontanées introduisent une plasticité supplémentaire à l'objet automate. Par exemple, il est facile de voir que l'on peut, en les utilisant, transformer un automate fini quelconque en un automate équivalent doté d'un unique état final n'ayant que des transitions entrantes.

 Le mot vide se note '\e' dans Vcsn. Pour utiliser des  $\varepsilon$ -NFA, il faut spécifier que le mot vide est une étiquette valide. Le contexte est 'lan\_char, b' au lieu de 'la1\_char, b'. Ainsi l'automate 4.12 s'écrit (notez le 'r' au début de la chaîne) :

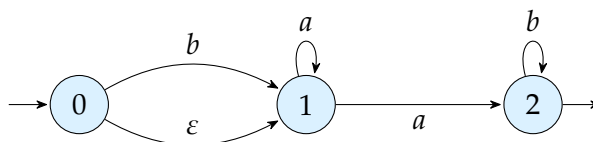
```
>>> a = vcsn.automaton(r''
context = "lan_char, b"
$ -> 0
0 -> 0 a
0 -> 1 \e
1 -> 1 b
1 -> 2 \e
2 -> 2 c
2 -> $
''')
```

Pour éliminer les transitions spontanées, utiliser '`automaton.proper`'.

**Exercice 4.14** (BCI-0304). *Considérant que l'on ne peut avoir simultanément des adjectifs avant et après le nom, un linguiste propose de représenter l'ensemble des groupes formés d'un nom et d'adjectif(s) par l'expression rationnelle :  $a^*n + na^*$ .*

1. Construisez, en appliquant systématiquement la méthode de Thompson (décrite en section 4.2.2), un automate fini correspondant à cette expression rationnelle.
2. Construisez l' $\varepsilon$ -fermeture de chacun des états de l'automate construit à la question 1.
3. Déduisez-en un automate sans  $\varepsilon$ -transition pour l'expression rationnelle de la question 1. Vous prendrez soin également d'identifier et d'éliminer les états inutiles.
4. Dérivez finalement, par une construction étudiée en cours, l'équivalent déterministe de l'automate de la question précédente.

**Exercice 4.15** (BCI-0405-1). *On considère A l'automate 4.14 non-déterministe.*



Automate 4.14 – Automate non-déterministe A

1. Construisez, en utilisant des méthodes du cours, l'automate déterministe A' équivalent à A.

## 4.2 Reconnaissables

Nous avons défini à la [section 4.1](#) les langages reconnaissables comme étant les langages reconnus par un automate fini déterministe. Les sections précédentes nous ont montré que nous aurions tout aussi bien pu les définir comme les langages reconnus par un automate fini non-déterministe ou encore par un automate fini non-déterministe avec transitions spon-tanées.

Dans cette section, nous montrons dans un premier temps que l'ensemble des langages recon-naissables est clos pour toutes les opérations « classiques », en produisant des constructions portant directement sur les automates. Nous montrons ensuite que les reconnaissables sont exactement les langages rationnels (présentés à la [section 3.1.1](#)), puis nous présentons un ensemble de résultats classiques permettant de caractériser les langages reconnaissables.

### 4.2.1 Opérations sur les reconnaissables

**Théorème 4.16** (Clôture par complémentation). *Les langages reconnaissables sont clos par com-plémentation.*

*Démonstration.* Soit  $L$  un reconnaissable et  $A$  un DFA *complet* reconnaissant  $L$ . On construit alors un automate  $A'$  pour  $\bar{L}$  en prenant  $A' = (\Sigma, Q, q_0, F', \delta)$ , avec  $F' = Q \setminus F$ . Tout calcul réussi de  $A$  se termine dans un état de  $F$ , entraînant son échec dans  $A'$ . Inversement, tout calcul échouant dans  $A$  aboutit dans un état non-final de  $A$ , ce qui implique qu'il réussit dans  $A'$ .  $\square$

Le théorème suivant est une tautologie, puisque par définition les langages rationnels sont clos par l'union. Néanmoins sa preuve fournit la construction d'un automate déterministe acceptant l'union des langages de deux automates déterministes.

**Théorème 4.17** (Clôture par union). *Les langages reconnaissables sont clos par union.*

*Démonstration.* Soit  $L^1$  et  $L^2$  deux langages reconnaissables, reconnus respectivement par  $A^1$  et  $A^2$ , deux DFA *complets*. On construit un automate  $A = (\Sigma, Q = Q^1 \times Q^2, q_0, F, \delta)$  pour  $L^1 \cup L^2$  de la manière suivante :

- $q_0 = (q_0^1, q_0^2)$
- $F = (F^1 \times Q^2) \cup (Q^1 \times F^2)$
- $\delta((q^1, q^2), a) = (\delta^1(q^1, a), \delta^2(q^2, a))$

La construction de  $A$  est destinée à faire fonctionner  $A^1$  et  $A^2$  « en parallèle » : pour tout symbole d'entrée  $a$ , on transite par  $\delta$  dans la paire d'états résultant d'une transition dans  $A^1$  et d'une transition dans  $A^2$ . Un calcul réussi dans  $A$  est un calcul réussi dans  $A^1$  (arrêt dans un état de  $F^1 \times Q^2$ ) ou dans  $A^2$  (arrêt dans un état de  $Q^1 \times F^2$ ).  $\square$

Nous verrons un peu plus loin ([automate 4.18](#)) une autre construction, plus simple, pour cette opération, mais qui à l'inverse de la précédente, ne préserve pas le déterminisme de la machine réalisant l'union.

**Théorème 4.18** (Clôture par intersection). *Les langages reconnaissables sont clos par intersection.*

Nous présentons une preuve constructive, mais notez que le résultat découle directement des deux résultats précédents et de la loi de de Morgan  $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ .

*Démonstration.* Soient  $L^1$  et  $L^2$  deux reconnaissables, reconnus respectivement par  $A^1$  et  $A^2$ , deux DFA complets. On construit un automate  $A = (\Sigma, Q = Q^1 \times Q^2, q_0, F, \delta)$  pour  $L^1 \cap L^2$  de la manière suivante :

- $q_0 = (q_0^1, q_0^2)$
- $F = F^1 \times F^2$
- $\delta((q^1, q^2), a) = (\delta^1(q^1, a), \delta^2(q^2, a))$

La construction de l'automate intersection est identique à celle de l'automate réalisant l'union, à la différence près qu'un calcul réussi dans  $A$  doit ici réussir simultanément dans les deux automates  $A^1$  et  $A^2$ . Ceci s'exprime dans la nouvelle définition de l'ensemble  $F$  des états finaux :  $F^1 \times F^2$ .  $\square$

**Exercice 4.19** (AST-0506). *L'intersection ne fait pas partie des opérations rationnelles. Pourtant, l'intersection de deux langages rationnels est également un langage rationnel. Si le langage  $L_1$  est reconnu par l'automate déterministe complet  $A_1 = (\Sigma, Q^1, q_0^1, F^1, \delta^1)$  et  $L_2$  par l'automate déterministe complet  $A_2 = (\Sigma, Q^2, q_0^2, F^2, \delta^2)$ , alors  $L = L_1 \cap L_2$  est reconnu par l'automate  $A = (\Sigma, Q^1 \times Q^2, q_0, F, \delta)$ . Les états de  $A$  sont des couples d'états de  $Q^1$  et de  $Q^2$ , et :*

- $q_0 = (q_0^1, q_0^2)$
- $F = F^1 \times F^2$
- $\delta((q_1, q_2), a) = (\delta^1(q_1, a), \delta^2(q_2, a))$

*A simule en fait des calculs en parallèle dans  $A_1$  et dans  $A_2$ , et termine un calcul réussi lorsqu'il atteint simultanément un état final de  $A_1$  et de  $A_2$ .*

*Dans la suite de cet exercice, on considère  $\Sigma = \{a, b\}$ .*

1. Soient  $L_1$  le langage des mots qui commencent par un  $a$  et  $L_2$  le langage des mots dont la longueur est un multiple de 3. Proposez, pour chacun de ces langages, un automate fini déterministe et complet.
2. Construisez, en utilisant les directives données ci-dessus, un automate pour  $L_1 \cap L_2$ .
3. Nous allons maintenant retrouver cette construction dans le cas général en utilisant la loi de de Morgan :  $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ . On rappelle que si  $L$  est rationnel et reconnu par  $A$  déterministe complet,  $\overline{L}$  est rationnel et est reconnu par  $\overline{A}$ , qui se déduit de  $A$  en rendant finaux les états non-finaux de  $A$ , et en rendant non-final les états finaux de  $A$ .  $\overline{A}$  est également déterministe et complet.

- a. Nous nous intéressons tout d'abord au calcul d'un automate déterministe  $\overline{A}$  pour  $\overline{L_1} \cup \overline{L_2}$ . L'automate pour l'union de deux langages rationnels reconnus respectivement par  $\overline{A_1}$  et  $\overline{A_2}$  se construit en ajoutant un nouvel état initial  $q_0$  et deux transitions  $\varepsilon$  vers les états

initiaux  $q_0^1$  et  $q_0^2$ . Son ensemble d'états est donc  $Q = Q^1 \cup Q^2 \cup \{q_0\}$ . On note  $\overline{A'}$  l'automate obtenu en éliminant la transition  $\varepsilon$ .

Prouvez que, si  $\overline{A_1}$  et  $\overline{A_2}$  sont déterministes, alors  $\overline{A'}$  ne possède qu'un seul état non-déterministe<sup>1</sup>.

- b. La déterminisation de  $\overline{A'}$  conduit à un automate  $\overline{A}$  dont l'ensemble des états est en correspondance avec les parties de  $Q$ . Prouvez, par induction, que si  $\overline{A_1}$  et  $\overline{A_2}$  sont déterministes et complets, seules les parties correspondant à des doubletons d'états  $\{q^1, q^2\}$ ,  $q^1 \in Q^1, q^2 \in Q^2$  donnent lieu à des états utiles du déterminisé. Montrez qu'alors on retrouve bien pour  $\overline{A}$  une fonction de transition conforme à la construction directe présentée ci-dessus.
- c. Quels sont les états finaux de  $\overline{A'}$ ? Qu'en déduisez-vous pour ceux de l'automate représentant le complémentaire du langage de  $\overline{A}$ ? Concluez en achevant de justifier la construction directe donnée ci-dessus.

**Exercice 4.20** (BCI-0203-2-1). Construire un automate non-déterministe  $A$  reconnaissant le langage  $L$  sur l'alphabet  $\Sigma = \{a, b\}$  tel que tous les mots de  $L$  satisfont simultanément les deux conditions suivantes :

- tout mot de  $L$  a une longueur divisible par 3
- tout mot de  $L$  débute par le symbole  $a$  et finit par le symbole  $b$

Vous justifierez votre construction.

Construire l'automate déterministe équivalent par la méthode des sous-ensembles.

**Théorème 4.21** (Clôture par miroir). Les langages reconnaissables sont clos par miroir.

*Démonstration.* Soit  $L$  un langage reconnaissable, reconnu par  $A = (\Sigma, Q, q_0, F, \delta)$ , et n'ayant qu'un unique état final, noté  $q_f$ .  $A'$ , défini par  $A' = (\Sigma, Q, q_f, \{q_0\}, \delta')$ , où  $\delta'(q, a) = p$  si et seulement si  $\delta(p, a) = q$  reconnaît exactement le langage miroir de  $L$ .  $A'$  est en fait obtenu en inversant l'orientation des arcs de  $A$  : tout calcul de  $A$  énumérant les symboles de  $u$  entre  $q_0$  et  $q_f$  correspond à un calcul de  $A'$  énumérant  $u^R$  entre  $q_f$  et  $q_0$ . On notera que même si  $A$  est déterministe, la construction ici proposée n'aboutit pas nécessairement à un automate déterministe pour le miroir.  $\square$

**Théorème 4.22** (Clôture par concaténation). Les langages reconnaissables sont clos par concaténation.

*Démonstration.* Soient  $L^1$  et  $L^2$  deux langages reconnus respectivement par  $A^1$  et  $A^2$ , où l'on suppose que les ensembles d'états  $Q^1$  et  $Q^2$  sont disjoints et que  $A^1$  a un unique état final de degré extérieur nul (aucune transition sortante). On construit l'automate  $A$  pour  $L^1L^2$  en identifiant l'état final de  $A^1$  avec l'état initial de  $A^2$ . Formellement on a  $A = (\Sigma, Q^1 \cup Q^2 \setminus \{q_0^2\}, q_0^1, F^2, \delta)$ , où  $\delta$  est défini par :

- $\forall q \in Q^1, q \neq q_f, a \in \Sigma, \delta(q, a) = \delta^1(q, a)$
- $\delta(q, a) = \delta^2(q, a)$  si  $q \in Q^2$ .

1. Un état est non-déterministe s'il admet plusieurs transitions sortantes étiquetées par la même lettre.

$$— \delta(q_F^1, a) = \delta^1(q_F, a) \cup \delta^2(q_0^2, a)$$

Tout calcul réussi dans  $A$  doit nécessairement atteindre un état final de  $A^2$  et pour cela préalablement atteindre l'état final de  $A^1$ , seul point de passage vers les états de  $A^2$ . De surcroît, le calcul n'emprunte, après le premier passage dans  $q_F^1$ , que des états de  $A_2$  : il se décompose donc en un calcul réussi dans chacun des automates. Réciproquement, un mot de  $L^1L^2$  se factorise sous la forme  $u = vw$ ,  $v \in L^1$  et  $w \in L^2$ . Chaque facteur correspond à un calcul réussi respectivement dans  $A^1$  et dans  $A^2$ , desquels se déduit immédiatement un calcul réussi dans  $A$ .  $\square$

**Théorème 4.23** (Clôture par étoile). *Les langages reconnaissables sont clos par étoile.*

*Démonstration.* La construction de  $A'$  reconnaissant  $L^*$  à partir de  $A$  reconnaissant  $L$  est la suivante. Tout d'abord, on ajoute à  $A$  un nouvel état initial  $q'_0$  avec une transition spontanée vers l'état initial  $q_0$  de  $A$ . On ajoute ensuite une transition spontanée depuis tout état final de  $A$  vers ce nouvel état initial  $q'_0$ . Cette nouvelle transition permet l'itération dans  $A'$  de mots de  $L$ . Pour compléter la construction, on marque  $q'_0$  comme état final de  $A'$ .  $\square$

En application de cette section, vous pourrez montrer (en construisant les automates correspondants) que les langages reconnaissables sont aussi clos pour les opérations de préfixation, suffixation, pour les facteurs, les sous-mots. . .

## 4.2.2 Reconnaissables et rationnels

Les propriétés de clôture démontrées pour les reconnaissables (pour l'union, la concaténation et l'étoile) à la section précédente, complétées par la remarque que tous les langages finis sont reconnaissables, nous permettent d'affirmer que tout langage rationnel est reconnaissable. L'ensemble des langages rationnels étant en effet le plus petit ensemble contenant tous les ensembles finis et clos pour les opérations rationnelles, il est nécessairement inclus dans l'ensemble des reconnaissables. Nous montrons dans un premier temps comment exploiter les constructions précédentes pour construire simplement un automate correspondant à une expression rationnelle donnée. Nous montrons ensuite la réciproque, à savoir que tout reconnaissable est également rationnel : les langages reconnus par les automates finis sont précisément ceux qui sont décrits par des expressions rationnelles.

### Des expressions rationnelles vers les automates

Les constructions de la section précédente ont montré comment construire les automates réalisant des opérations élémentaires sur les langages. Nous allons nous inspirer de ces constructions pour dériver un algorithme permettant de convertir une expression rationnelle en un automate fini reconnaissant le même langage.

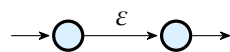
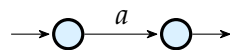
Il existe de nombreuses façons de construire un tel automate, celle que nous donnons ci-dessous est la construction de Thompson « pure » qui présente quelques propriétés simples :

- un unique état initial  $q_0$  sans transition entrante

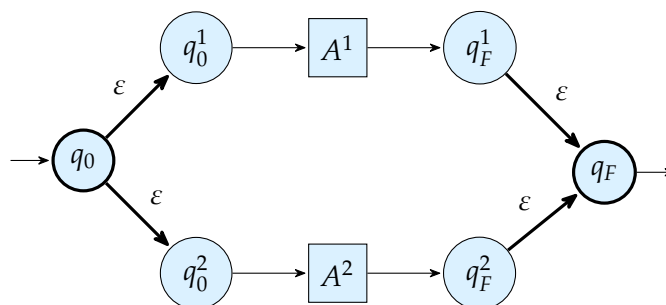
- un unique état final  $q_F$  sans transition sortante
- exactement deux fois plus d'états que de symboles dans l'expression rationnelle (sans compter les parenthèses ni la concaténation, implicite).

Cette dernière propriété donne un moyen simple de contrôler le résultat en contrepartie d'automates parfois plus lourds que nécessaire.

Puisque les expressions rationnelles sont formellement définies de manière inductive (récursive) nous commençons par présenter les automates finis pour les « briques » de base que sont  $\emptyset$  (automate 4.15),  $\varepsilon$  (automate 4.16) et les symboles de  $\Sigma$  (automate 4.17).

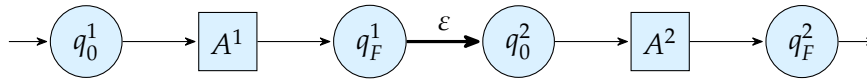
Automate 4.15 – Automate de Thompson pour  $\emptyset$ Automate 4.16 – Automate de Thompson pour  $\varepsilon$ Automate 4.17 – Automate de Thompson pour  $a$ 

À partir de ces automates élémentaires, nous allons construire de manière itérative des automates pour des expressions rationnelles plus complexes. Si  $A_1$  et  $A_2$  sont les automates de Thompson de  $e_1$  et  $e_2$ , alors l'automate 4.18 reconnaît le langage dénoté par l'expression  $e_1 + e_2$ .

Automate 4.18 – Automate de Thompson pour  $e_1 + e_2$ 

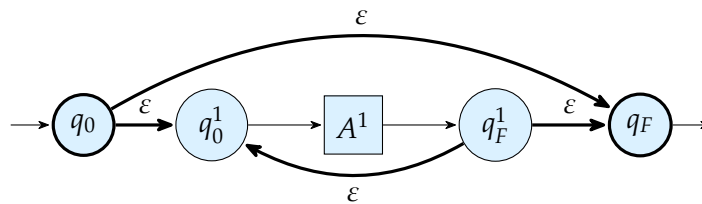
L'union correspond donc à une mise en parallèle de  $A_1$  et de  $A_2$  : un calcul dans cette machine est réussi si et seulement s'il est réussi dans l'une des deux machines  $A_1$  ou  $A_2$ .

La machine reconnaissant le langage dénoté par concaténation de deux expressions  $e_1$  et  $e_2$  correspond à une mise en série des deux machines  $A_1$  et  $A_2$ , où l'état final de  $A_1$  est connecté à l'état initial de  $A_2$  par une transition spontanée comme sur l'automate 4.19.



Automate 4.19 – Automate de Thompson pour  $e_1e_2$

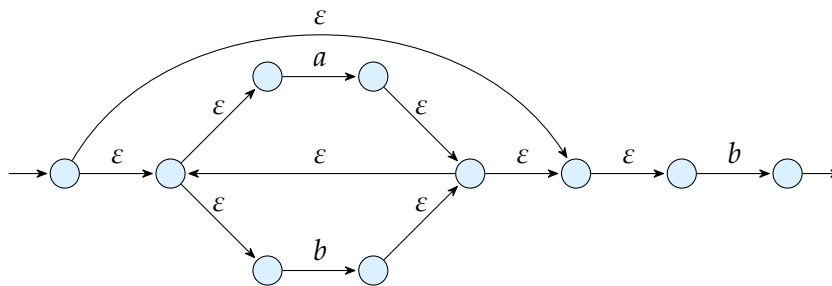
La machine réalisant l'étoile est, comme précédemment, construite en rajoutant une possibilité de boucler depuis l'état final vers l'état initial de la machine, ainsi qu'un arc permettant de reconnaître  $\epsilon$ , comme sur l'automate 4.20.



Automate 4.20 – Automate de Thompson pour  $e_1^*$

À partir de ces constructions simples, il est possible de dériver un algorithme permettant de construire un automate reconnaissant le langage dénoté par une expression régulière quelconque : il suffit de décomposer l'expression en ses composants élémentaires, puis d'appliquer les constructions précédentes pour construire l'automate correspondant. Cet algorithme est connu sous le nom d'*algorithme de Thompson*.

L'automate 4.21 illustre cette construction.



Automate 4.21 – Automate de Thompson pour  $(a + b)^*b$

Cette construction simple produit un automate qui a  $2n$  états pour une expression formée par  $n$  opérations rationnelles autres que la concaténation (car toutes les autres opérations rajoutent exactement deux états); chaque état de l'automate possède au plus deux transitions sortantes. Cependant, cette construction a le désavantage de produire un automate non-déterministe, contenant de multiples transitions spontanées. Il existe d'autres procédures permettant de traiter de manière plus efficace les expressions ne contenant pas le symbole  $\epsilon$  (par exemple la *construction de Glushkov*) ou pour construire directement un automate déterministe.





La méthode '`expression.thompson`' engendre l'automate de Thompson d'une expression.

```
>>> vcsn.B.expression('(a+b)*b').thompson()
```

### Des automates vers les expressions rationnelles

La construction d'une expression rationnelle dénotant le langage reconnu par un automate  $A$  demande l'introduction d'une nouvelle variété d'automates, que nous appellerons *généralisés*. Les automates généralisés diffèrent des automates finis en ceci que leurs transitions sont étiquetées par des sous-ensembles rationnels de  $\Sigma^*$ . Dans un automate généralisé, l'étiquette d'un calcul se construit par concaténation des étiquettes rencontrées le long des transitions ; le langage reconnu par un automate généralisé est l'union des langages correspondants aux calculs réussis. Les automates généralisés reconnaissent exactement les mêmes langages que les automates finis « standard ».

L'idée générale de la transformation que nous allons étudier consiste à partir d'un automate fini standard et de supprimer un par un les états, tout en s'assurant que cette suppression ne modifie pas le langage reconnu par l'automate. Ceci revient à construire de proche en proche une série d'automates généralisés qui sont tous équivalents à l'automate de départ. La procédure se termine lorsqu'il ne reste plus que l'unique état initial et l'unique état final : en lisant l'étiquette des transitions correspondantes, on déduit une expression rationnelle dénotant le langage reconnu par l'automate original.

Pour se simplifier la tâche commençons par introduire deux nouveaux états,  $q_I$  et  $q_F$ , qui joueront le rôle d'unicques états respectivement initial et final. Ces nouveaux états sont connectés aux états initiaux et finaux par des transitions spontanées. On s'assure ainsi qu'à la fin de l'algorithme, il ne reste plus qu'une seule et unique transition, celle qui relie  $q_I$  à  $q_F$ .

L'opération cruciale de cette méthode est celle qui consiste à supprimer l'état  $q_j$ , où  $q_j$  n'est ni initial, ni final. On suppose qu'il y a au plus une transition entre deux états : si ça n'est pas le cas, il est possible de se ramener à cette configuration en prenant l'union des transitions existantes. On note  $e_{jj}$  l'étiquette de la transition de  $q_j$  vers  $q_j$  si celle-ci existe ; si elle n'existe pas on a simplement  $e_{jj} = \varepsilon$ .

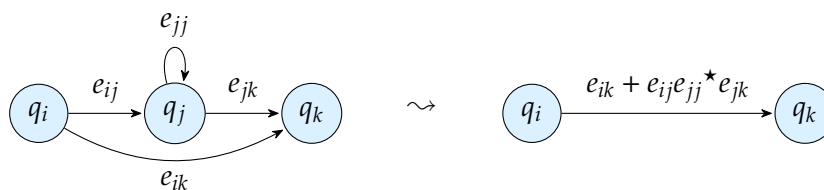
La procédure de suppression de  $q_j$  comprend alors les étapes suivantes :

- pour chaque paire d'états  $(q_i, q_k)$  avec  $i \neq j, k \neq j$ , telle qu'il existe une transition  $q_i \rightarrow q_j$  étiquetée  $e_{ij}$  et une transition  $q_j \rightarrow q_k$  étiquetée  $e_{jk}$ , ajouter la transition  $q_i \rightarrow q_k$ , portant l'étiquette  $e_{ij}e_{jj}^*e_{jk}$ . Si la transition  $q_i \rightarrow q_k$  existe déjà avec l'étiquette  $e_{ik}$ , alors l'étiquette sera  $e_{ik} + e_{ij}e_{jj}^*e_{jk}$ .

Cette transformation doit être opérée pour chaque paire d'états (y compris quand  $q_i = q_k$ ) avant que  $q_j$  puisse être supprimé.

- supprimer  $q_j$ , ainsi que tous les arcs incidents.

Ce mécanisme est illustré sur l'[automate 4.22](#).



Automate 4.22 – Illustration de BMC : élimination de l'état  $q_j$

La preuve de la correction de cet algorithme réside en la vérification qu'à chaque itération le langage reconnu ne change pas. Cet invariant se vérifie très simplement : tout calcul réussi passant par  $q_j$  avant que  $q_j$  soit supprimé contient une séquence  $q_i q_j q_k$ . L'étiquette de ce sous-calcul étant copiée lors de la suppression de  $q_j$  sur l'arc  $q_i \rightarrow q_k$ , un calcul équivalent existe dans l'automate réduit. Réciproquement, tout calcul réussi dans l'automate après suppression passant par une nouvelle transition, ou une transition dont l'étiquette a changé, correspond à un calcul de l'automate original qui passe par  $q_j$ .

Cette méthode de construction de l'expression équivalente à un automate est appelée *méthode d'élimination des états*, connue également sous le nom d'*algorithme de Brzozowski et McCluskey*. Vous noterez que le résultat obtenu dépend de l'ordre selon lequel les états sont examinés.

En guise d'application, il est recommandé de déterminer une expression rationnelle correspondant aux [automates 4.8](#) et [4.9](#).

Un corollaire fondamental de ce résultat est que pour chaque langage reconnaissable, il existe (au moins) une expression rationnelle qui le dénote : tout langage reconnaissable est rationnel.

### Théorème de Kleene

Nous avons montré comment construire un automate correspondant à une expression rationnelle et comment dériver une expression rationnelle dénotant un langage équivalent à celui reconnu par un automate fini quelconque. Ces deux résultats permettent d'énoncer un des résultats majeurs de ce chapitre.

**Théorème 4.24** (Théorème de Kleene). *Un langage est reconnaissable (reconnu par un automate fini) si et seulement si il est rationnel (dénoté par une expression rationnelle).*

Ce résultat permet de tirer quelques conséquences non encore envisagées : par exemple que les langages rationnels sont clos par complémentation et par intersection finie : ce résultat, loin d'être évident si on s'en tient aux notions d'opérations rationnelles, tombe simplement lorsque l'on utilise l'équivalence entre rationnels et reconnaissables.

De manière similaire, les méthodes de transformation d'une représentation à l'autre sont bien pratiques : s'il est immédiat de dériver de l'[automate 4.3](#) une expression rationnelle pour le langage contenant un nombre de  $a$  multiple de 3, il est beaucoup moins facile d'écrire directement l'expression rationnelle correspondante. En particulier, le passage par la représentation sous forme d'automates permet de déduire une méthode pour vérifier si deux expressions sont équivalentes : construire les deux automates correspondants, et

vérifier qu'ils sont équivalents. Nous savons déjà comment réaliser la première partie de ce programme; une procédure pour tester l'équivalence de deux automates est présentée à la [section 4.3.2](#).

## 4.3 Quelques propriétés des langages reconnaissables

L'équivalence entre langages rationnels et langages reconnaissables a enrichi singulièrement notre palette d'outils concernant les langages rationnels : pour montrer qu'un langage est rationnel, on peut au choix exhiber un automate fini pour ce langage ou bien encore une expression rationnelle. Pour montrer qu'un langage *n'est pas rationnel*, il est utile de disposer de la propriété présentée dans la [section 4.3.1](#), qui est connue sous le nom de *lemme de pompage*.

### 4.3.1 Lemme de pompage

Intuitivement, le *lemme de pompage* (ou *lemme de l'étoile*, et en anglais *pumping lemma* ou *star lemma*) pose des limitations intrinsèques concernant la diversité des mots appartenant à un langage rationnel infini : au delà d'une certaine longueur, les mots d'un langage rationnel sont en fait construits par itération de motifs apparaissant dans des mots plus courts.

**Théorème 4.25** (Lemme de pompage). *Soit  $L$  un langage rationnel. Il existe un entier  $k$  tel que pour tout mot  $x$  de  $L$  plus long que  $k$ ,  $x$  se factorise en  $x = uvw$ , avec (i)  $|v| > 0$  (ii)  $|uv| \leq k$  et (iii) pour tout  $i \geq 0$ ,  $uv^i w$  est également un mot de  $L$ .*

Ce que signifie fondamentalement ce lemme, c'est que la seule façon d'aller à l'infini pour un langage rationnel, c'est par itération de motifs :  $L(uv^*w) \subset L$  —d'où le nom de « lemme de l'étoile ».

*Démonstration.* Lorsque le langage est fini, le lemme est trivial : on peut choisir un  $k$  qui majore la longueur de tous les mots du langage.

Sinon, soit  $A$  un DFA de  $k$  états reconnaissant  $L$ , et  $x$  un mot de  $L$ , de longueur supérieure ou égale à  $k$  (le langage étant infini, un tel  $x$  existe toujours). La reconnaissance de  $x$  dans  $A$  correspond à un calcul  $q_0 \dots q_n$  impliquant  $|x| + 1$  états.  $A$  n'ayant que  $k$  états, le préfixe de longueur  $k + 1$  de cette séquence contient nécessairement deux fois le même état  $q$ , aux indices  $i$  et  $j$ , avec  $0 \leq i < j \leq k$ . Si l'on note  $u$  le préfixe de  $x$  tel que  $\delta^*(q_0, u) = q$  et  $v$  le facteur tel que  $\delta^*(q, v) = q$ , alors on a bien (i) car au moins un symbole est consommé le long du cycle  $q \dots q$ ; (ii) car  $j \leq k$ ; (iii) en court-circuitant ou en itérant les parcours le long du circuit  $q \dots q$ .  $\square$

Attention : certains langages non rationnels vérifient la propriété du lemme : elle n'est qu'une condition nécessaire, mais pas suffisante, de rationalité. Considérez par exemple  $\{a^n b^n \mid n \in \mathbb{N}\} \cup \Sigma^* \{ba\} \Sigma^*$ .

Ce lemme permet, par exemple, de prouver que le langage des carrés parfaits,  $L = \{u \in \Sigma^*, \exists v \text{ tel que } u = v^2\}$ , n'est pas rationnel. En effet, soit  $k$  l'entier spécifié par le lemme de

pompagement et  $x$  un mot plus grand que  $2k$  :  $x = yy$  avec  $|y| \geq k$ . Il est alors possible d'écrire  $x = uvw$ , avec  $|uv| \leq k$ . Ceci implique que  $uv$  est un préfixe de  $y$ , et  $y$  un suffixe de  $w$ . Pourtant,  $uv^i w$  doit également être dans  $L$ , alors qu'un seul des  $y$  est affecté par l'itération : ceci est manifestement impossible.

Vous montrerez de même que le langage ne contenant que les mots dont la longueur est un carré parfait et que le langage des mots dont la longueur est un nombre premier ne sont pas non plus des langages reconnaissables.

Une manière simple d'exprimer cette limitation intrinsèque des langages rationnels se fonde sur l'observation suivante : dans un automate, le choix de l'état successeur pour un état  $q$  ne dépend que de  $q$ , et pas de la manière dont le calcul s'est déroulé avant  $q$ . En conséquence, un automate fini ne peut gérer qu'un nombre fini de configurations différentes, ou, dit autrement, possède une mémoire bornée. C'est insuffisant pour un langage tel que le langage des carrés parfaits pour lequel l'action à conduire (le langage à reconnaître) après un préfixe  $u$  dépend de  $u$  tout entier : reconnaître un tel langage demanderait en fait un nombre infini d'états.

### 4.3.2 Quelques conséquences

Dans cette section, nous établissons quelques résultats complémentaires portant sur la décidabilité, c'est-à-dire sur l'existence d'algorithmes permettant de résoudre quelques problèmes classiques portant sur les langages rationnels. Nous connaissons déjà un algorithme pour décider si un mot appartient à un langage rationnel (l'algorithme 4.2); cette section montre en fait que la plupart des problèmes classiques pour les langages rationnels ont des solutions algorithmiques.

**Théorème 4.26.** *Si  $A$  est un automate fini contenant  $k$  états :*

- (i)  $L(A)$  est non vide si et seulement si  $A$  reconnaît un mot de longueur strictement inférieure à  $k$ .
- (ii)  $L(A)$  est infini si et seulement si  $A$  reconnaît un mot  $u$  tel que  $k \leq |u| < 2k$ .

*Démonstration.* **point i** : un sens de l'implication est trivial : si  $A$  reconnaît un mot de longueur inférieure à  $k$ ,  $L(A)$  est non-vide. Supposons  $L(A)$  non vide et soit  $u$  le plus petit mot de  $L(A)$ ; supposons que la longueur de  $u$  soit strictement supérieure à  $k$ . Le calcul  $(q_0, u) \vdash_A^* (q, \varepsilon)$  contient au moins  $k$  étapes, impliquant qu'un état au moins est visité deux fois et est donc impliqué dans un circuit  $C$ . En court-circuitant  $C$ , on déduit un mot de  $L(A)$  de longueur strictement inférieure à la longueur de  $u$ , ce qui contredit l'hypothèse de départ. On a donc bien  $|u| < k$ .

**point ii** : un raisonnement analogue à celui utilisé pour montrer le lemme de pompage nous assure qu'un sens de l'implication est vrai. Si maintenant  $L(A)$  est infini, il doit au moins contenir un mot plus long que  $k$ . Soit  $u$  le plus petit mot de longueur au moins  $k$  : soit il est de longueur strictement inférieure à  $2k$ , et le résultat est prouvé. Soit il est au moins égal à  $2k$ , mais par le lemme de pompage on peut court-circuiter un facteur de taille au plus  $k$  et donc exhiber un mot strictement plus court et de longueur au moins  $k$ , ce qui est impossible. C'est donc que le plus petit mot de longueur au moins  $k$  a une longueur inférieure à  $2k$ .  $\square$

**Théorème 4.27.** *Soit  $A$  un automate fini, il existe un algorithme permettant de décider si :*

- $L(A)$  est vide
- $L(A)$  est fini / infini

Ce résultat découle directement des précédents : il existe, en effet, un algorithme pour déterminer si un mot  $u$  est reconnu par  $A$ . Le résultat précédent nous assure qu'il suffit de tester  $|\Sigma|^k$  mots pour décider si le langage d'un automate  $A$  est vide. De même,  $|\Sigma|^{2k} - |\Sigma|^k$  vérifications suffisent pour prouver qu'un automate reconnaît un langage infini. Pour ces deux problèmes, des algorithmes plus efficaces que celui qui nous a servi à établir ces preuves existent, qui reposent sur des parcours du graphe sous-jacent de l'automate.

On en déduit un résultat concernant l'équivalence :

**Théorème 4.28.** *Soient  $A_1$  et  $A_2$  deux automates finis. Il existe une procédure permettant de décider si  $A_1$  et  $A_2$  sont équivalents.*

*Démonstration.* Il suffit en effet pour cela de former l'automate reconnaissant  $(L(A_1) \cap \overline{L(A_2)}) \cup (\overline{L(A_1)} \cap L(A_2))$  (par exemple en utilisant les procédures décrites à la [section 4.2.1](#)) et de tester si le langage reconnu par cet automate est vide. Si c'est le cas, alors les deux automates sont effectivement équivalents.  $\square$

## 4.4 L'automate canonique

Dans cette section, nous donnons une nouvelle caractérisation des langages reconnaissables, à partir de laquelle nous introduisons la notion d'*automate canonique d'un langage*. Nous présentons ensuite un algorithme pour construire l'automate canonique d'un langage reconnaissable représenté par un DFA quelconque.

### 4.4.1 Une nouvelle caractérisation des reconnaissables

Commençons par une nouvelle définition : celle d'indistinguabilité.

**Définition 4.29** (Mots indistinguables d'un langage). *Soit  $L$  un langage de  $\Sigma^*$ . Deux mots  $u$  et  $v$  sont dits indistinguables dans  $L$  si pour tout  $w \in \Sigma^*$ , soit  $uw$  et  $vw$  sont tous deux dans  $L$ , soit  $uw$  et  $vw$  sont tous deux dans  $\bar{L}$ . On notera  $\equiv_L$  la relation d'indistinguabilité dans  $L$ .*

En d'autres termes, deux mots  $u$  et  $v$  sont *distinguables dans  $L$*  s'il existe un mot  $w \in \Sigma^*$  tel que  $uw \in L$  et  $vw \notin L$ , ou bien le contraire. La relation d'indistinguabilité dans  $L$  est une relation réflexive, symétrique et transitive : c'est une relation d'équivalence.

Considérons, à titre d'illustration, le langage  $L = a(a + b)(bb)^*$ . Pour ce langage,  $u = aab$  et  $v = abb$  sont indistinguables : pour tout mot  $x = uw$  de  $L$ ,  $y = vw$  est en effet un autre mot de  $L$ . En revanche  $u = a$  et  $v = aa$  sont distinguables : en concaténant  $w = abb$  à  $u$ , on obtient  $aabb$  qui est dans  $L$ ; en revanche,  $aaabb$  n'est pas un mot de  $L$ .

De manière similaire, on définit la notion d'indistinguabilité dans un automate déterministe.

**Définition 4.30** (Mots indistinguables d'un automate). Soit  $A = (\Sigma, Q, q_0, F, \delta)$  un automate fini déterministe. Deux mots  $u$  et  $v$  sont dits indistinguables dans  $A$  si et seulement si  $\delta^*(q_0, u) = \delta^*(q_0, v)$ . On notera  $\equiv_A$  la relation d'indistinguabilité dans  $A$ .

Autrement dit, deux mots  $u$  et  $v$  sont indistinguables pour  $A$  si le calcul par  $A$  de  $u$  depuis  $q_0$  aboutit dans le même état  $q$  que le calcul de  $v$ . Cette notion est liée à la précédente : pour deux mots  $u$  et  $v$  indistinguables dans  $L(A)$ , tout mot  $w$  tel que  $\delta^*(q, w)$  aboutisse dans un état final est une continuation valide à la fois de  $u$  et de  $v$  dans  $L$  ; inversement tout mot conduisant à un échec depuis  $q$  est une continuation invalide à la fois de  $u$  et de  $v$ . En revanche, la réciproque est fautive, et deux mots indistinguables dans  $L(A)$  peuvent être distinguables dans  $A$ .

Pour continuer, rappelons la notion de congruence droite, déjà introduite à la [section 2.4.3](#) :

**Définition 4.31** (Invariance droite). Une relation d'équivalence  $\mathcal{R}$  sur  $\Sigma^*$  est dite invariante à droite si et seulement si :  $u \mathcal{R} v \Rightarrow \forall w, uw \mathcal{R} vw$ . Une relation invariante à droite est appelée une congruence droite.

Par définition, les deux relations d'indistinguabilité définies ci-dessus sont invariantes à droite.

Nous sommes maintenant en mesure d'exposer le résultat principal de cette section.

**Théorème 4.32** (Myhill-Nerode). Soit  $L$  un langage sur  $\Sigma$ , les trois assertions suivantes sont équivalentes :

- (i)  $L$  est un langage rationnel
- (ii) il existe une relation d'équivalence  $\equiv$  sur  $\Sigma^*$ , invariante à droite, ayant un nombre fini de classes d'équivalence et telle que  $L$  est égal à l'union de classes d'équivalence de  $\equiv$
- (iii)  $\equiv_L$  possède un nombre fini de classes d'équivalence

*Démonstration.* [point i](#)  $\Rightarrow$  [point ii](#) :  $A$  étant rationnel, il existe un DFA  $A$  qui le reconnaît. La relation d'équivalence  $\equiv_A$  ayant autant de classes d'équivalence qu'il y a d'états, ce nombre est nécessairement fini. Cette relation est bien invariante à droite, et  $L$ , défini comme  $\{u \in \Sigma^* \mid \delta^*(q_0, u) \in F\}$ , est simplement l'union des classes d'équivalence associées aux états finaux de  $A$ .

[point ii](#)  $\Rightarrow$  [point iii](#). Soit  $\equiv$  la relation satisfaisant la propriété du [point ii](#), et  $u$  et  $v$  tels que  $u \equiv v$ . Par la propriété d'invariance droite, on a pour tout mot  $w$  dans  $\Sigma^*$   $uw \equiv vw$ . Ceci entraîne que soit  $uw$  et  $vw$  sont simultanément dans  $L$  (si leur classe d'équivalence commune est un sous-ensemble de  $L$ ), soit tout deux hors de  $L$  (dans le cas contraire). Il s'ensuit que  $u \equiv_L v$  : toute classe d'équivalence pour  $\equiv$  est incluse dans une classe d'équivalence de  $\equiv_L$  ; il y a donc moins de classes d'équivalence pour  $\equiv_L$  que pour  $\equiv$ , ce qui entraîne que le nombre de classes d'équivalence de  $\equiv_L$  est fini.

[point iii](#)  $\Rightarrow$  [point i](#) : Construisons l'automate  $A = (\Sigma, Q, q_0, F, \delta)$  suivant :

- chaque état de  $Q$  correspond à une classe d'équivalence  $[u]_L$  de  $\equiv_L$  ; d'où il s'ensuit que  $Q$  est fini.
- $q_0 = [\varepsilon]_L$ , classe d'équivalence de  $\varepsilon$
- $F = \{[u]_L, u \in L\}$

- $\delta([u]_L, a) = [ua]_L$ . Cette définition de  $\delta$  est indépendante du choix d'un représentant de  $[u]_L$  : si  $u$  et  $v$  sont dans la même classe pour  $\equiv_L$ , par invariance droite de  $\equiv_L$ , il en ira de même pour  $ua$  et  $va$ .

$A$  ainsi défini est un automate fini déterministe et complet. Montrons maintenant que  $A$  reconnaît  $L$  et pour cela, montrons par induction que  $(q_0, u) \vdash_A^* [u]_L$ . Cette propriété est vraie pour  $u = \varepsilon$ , supposons la vraie pour tout mot de taille inférieure à  $k$  et soit  $u = va$  de taille  $k + 1$ . On a  $(q_0, ua) \vdash_A^* (p, a) \vdash_A (q, \varepsilon)$ . Or, par l'hypothèse de récurrence on sait que  $p = [u]_L$ ; et comme  $q = \delta([u]_L, a)$ , alors  $q = [ua]_L$ , ce qui est bien le résultat recherché. On déduit que si  $u$  est dans  $L(A)$ , un calcul sur l'entrée  $u$  aboutit dans un état final de  $A$ , et donc que  $u$  est dans  $L$ . Réciproquement, si  $u$  est dans  $L$ , un calcul sur l'entrée  $u$  aboutit dans un état  $[u]_L$ , qui est, par définition de  $A$ , final.  $\square$

Ce résultat fournit une nouvelle caractérisation des langages reconnaissables et peut donc être utilisé pour montrer qu'un langage est, ou n'est pas, reconnaissable. Ainsi, par exemple,  $L = \{u \in \Sigma^* \mid \exists a \in \Sigma, i \in \mathbb{N} \text{ tq. } u = a^{2^i}\}$  n'est pas reconnaissable. En effet, pour tout  $i, j$ ,  $a^{2^i}$  et  $a^{2^j}$  sont distingués par  $a^{2^i}$ . Il n'y a donc pas un nombre fini de classes d'équivalence pour  $\equiv_L$ , et  $L$  ne peut en conséquence être reconnu par un automate fini.  $L$  n'est donc pas un langage rationnel.

#### 4.4.2 Automate canonique

La principale conséquence du résultat précédent concerne l'existence d'un représentant unique (à une renumérotation des états près) et minimal (en nombre d'états) pour les classes de la relation d'équivalence sur les automates finis.

**Théorème 4.33** (Automate canonique). *L'automate  $A_L$ , fondé sur la relation d'équivalence  $\equiv_L$ , est le plus petit automate déterministe complet reconnaissant  $L$ . Cet automate est unique (à une renumérotation des états près) et est appelé automate canonique de  $L$ .*

*Démonstration.* Soit  $A$  un automate fini déterministe reconnaissant  $L$ .  $\equiv_A$  définit une relation d'équivalence satisfaisant les conditions du [point ii](#) de la preuve du [théorème 4.32](#) présenté ci-dessus. Nous avons montré que chaque état de  $A$  correspond à une classe d'équivalence (pour  $\equiv_A$ ) incluse dans une classe d'équivalence pour  $\equiv_L$ . Le nombre d'états de  $A$  est donc nécessairement plus grand que celui de  $A_L$ . Le cas où  $A$  et  $A_L$  ont le même nombre d'états correspond au cas où les classes d'équivalence sont toutes semblables, permettant de définir une correspondance biunivoque entre les états des deux machines.  $\square$

L'existence de  $A_L$  étant garantie, reste à savoir comment le construire : la construction directe des classes d'équivalence de  $\equiv_L$  n'est en effet pas nécessairement immédiate. Nous allons présenter un algorithme permettant de construire  $A_L$  à partir d'un automate déterministe quelconque reconnaissant  $L$ . Comme au préalable, nous définissons une troisième relation d'indistinguabilité, portant cette fois sur les états :

**Définition 4.34** (États indistinguables). *Deux états  $q$  et  $p$  d'un automate fini déterministe  $A$  sont distinguables s'il existe un mot  $w$  tel que le calcul  $(q, w)$  termine dans un état final alors que le calcul  $(p, w)$  échoue. Si deux états ne sont pas distinguables, ils sont indistinguables.*

Comme les relations d'indistinguabilité précédentes, cette relation est une relation d'équivalence, notée  $\equiv_v$ , sur les états de  $Q$ . L'ensemble des classes d'équivalence  $[q]_v$  est notée  $Q_v$ . Pour un automate fini déterministe  $A = (\Sigma, Q, q_0, F, \delta)$ , on définit l'automate fini  $A_v$  par :  $A_v = (\Sigma, Q_v, [q_0]_v, F_v, \delta_v)$ , avec :  $\delta_v([q]_v, a) = [\delta(q, a)]_v$ ; et  $F_v = [q]_v$ , avec  $q$  dans  $F$ . La fonction  $\delta_v$  est correctement définie en ce sens que si  $p$  et  $q$  sont indistinguables, alors nécessairement  $\delta(q, a) \equiv_v \delta(p, a)$ .

Montrons, dans un premier temps, que ce nouvel automate est bien identique à l'automate canonique  $A_L$ . On définit pour cela une application  $\phi$ , qui associe un état de  $A_v$  à un état de  $A_L$  de la manière suivante :

$$\phi([q]_v) = [u]_L \text{ s'il existe } u \text{ tel que } \delta^*(q_0, u) = q$$

Notons d'abord que  $\phi$  est une application : si  $u$  et  $v$  de  $\Sigma^*$  aboutissent tous deux dans des états indistinguables de  $A$ , alors il est clair que  $u$  et  $v$  sont également indistinguables, et sont donc dans la même classe d'équivalence pour  $\equiv_L$  : le résultat de  $\phi$  ne dépend pas d'un choix particulier de  $u$ .

Montrons maintenant que  $\phi$  est une bijection. Ceci se déduit de la suite d'équivalences suivante :

$$\phi([q]_v) = \phi([p]_v) \Leftrightarrow \exists u, v \in \Sigma^*, \delta^*(q_0, u) = q, \delta^*(q_0, u) = p, \text{ et } u \equiv_L v \quad (4.1)$$

$$\Leftrightarrow \delta^*(q_0, u) \equiv_v \delta^*(q_0, u) \quad (4.2)$$

$$\Leftrightarrow [q]_v = [p]_v \quad (4.3)$$

Montrons enfin que les calculs dans  $A_v$  sont en bijection par  $\phi$  avec les calculs de  $A_L$ . On a en effet :

- $\phi([q_0]_v) = [\varepsilon]_L$ , car  $\delta^*(q_0, \varepsilon) = q_0 \in [q_0]_v$
- $\phi(\delta_v([q]_v, a)) = \delta_L(\phi([q]_v), a)$  car soit  $u$  tel que  $\delta^*(q_0, u) \in [q]_v$ , alors : (i)  $\delta(\delta^*(q_0, u), a) \in \delta_v([q]_v, a)$  (cf. la définition de  $\delta_v$ ) et  $[ua]_L = \phi(\delta_v([q]_v, a)) = \delta_L([u], a)$  (cf. la définition de  $\delta_L$ ), ce qu'il fallait prouver.
- si  $[q]_v$  est final dans  $A_v$ , alors il existe  $u$  tel que  $\delta^*(q_0, u)$  soit un état final de  $A$ , impliquant que  $u$  est un mot de  $L$ , et donc que  $[u]_L$  est un état final de l'automate canonique.

Il s'ensuit que chaque calcul dans  $A_v$  est isomorphe (par  $\phi$ ) à un calcul dans  $A_L$ , et ainsi que, ces deux automates ayant les mêmes états initiaux et finaux, ils reconnaissent le même langage.

### 4.4.3 Minimisation

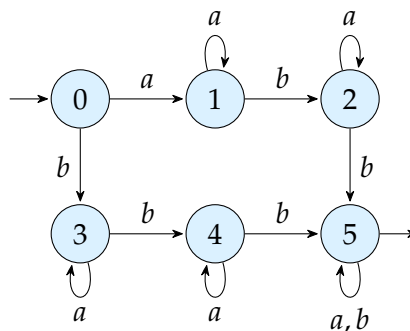
L'idée de l'algorithme de minimisation de l'automate déterministe  $A = (\Sigma, Q, q_0, F, \delta)$  consiste alors à chercher à identifier les classes d'équivalence pour  $\equiv_v$ , de manière à dériver l'automate  $A_v$  (alias  $A_L$ ). La finitude de  $Q$  nous garantit l'existence d'un algorithme pour calculer ces classes d'équivalence. La procédure itérative décrite ci-dessous esquisse une implantation naïve de cet algorithme, qui construit la partition correspondant aux classes d'équivalence par raffinement d'une partition initiale  $\Pi_0$  qui distingue simplement états finaux et non-finaux. Cet algorithme se glose comme suit :



- Initialiser avec deux classes d'équivalence :  $F$  et  $Q \setminus F$
- Itérer jusqu'à stabilisation :
  - pour toute paire d'états  $q$  et  $p$  dans la même classe de la partition  $\Pi_k$ , s'il existe  $a \in \Sigma$  tel que  $\delta(q, a)$  et  $\delta(p, a)$  ne sont pas dans la même classe pour  $\Pi_k$ , alors ils sont dans deux classes différentes de  $\Pi_{k+1}$ .

On vérifie que lorsque cette procédure s'arrête (après un nombre fini d'étapes), deux états sont dans la même classe si et seulement si ils sont indistinguables. Cette procédure est connue sous le nom d'*algorithme de Moore*. Implémentée de manière brutale, elle aboutit à une complexité quadratique (à cause de l'étape de comparaison de toutes les paires d'états). En utilisant des structures auxiliaires, il est toutefois possible de se ramener à une complexité en  $n \log(n)$ , avec  $n$  le nombre d'états.

Considérons pour illustrer cette procédure l'[automate 4.23](#) :

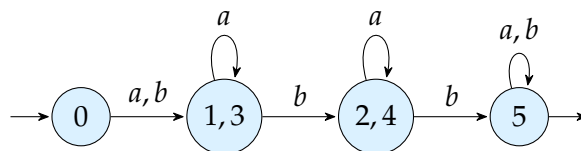


Automate 4.23 – Un DFA à minimiser


Les itérations successives de l'algorithme de construction des classes d'équivalence pour  $\equiv_v$  se déroulent alors comme suit :

- $\Pi_0 = \{\{0, 1, 2, 3, 4\}, \{5\}\}$  (car 5 est le seul état final)
- $\Pi_1 = \{\{0, 1, 3\}, \{2, 4\}, \{5\}\}$  (car 2 et 4, sur le symbole  $b$ , atteignent 5).
- $\Pi_2 = \{\{0\}, \{1, 3\}, \{2, 4\}, \{5\}\}$  (car 1 et 3, sur le symbole  $b$ , atteignent respectivement 2 et 4).
- $\Pi_3 = \Pi_2$  fin de la procédure.

L'automate minimal résultant de ce calcul est l'[automate 4.24](#).

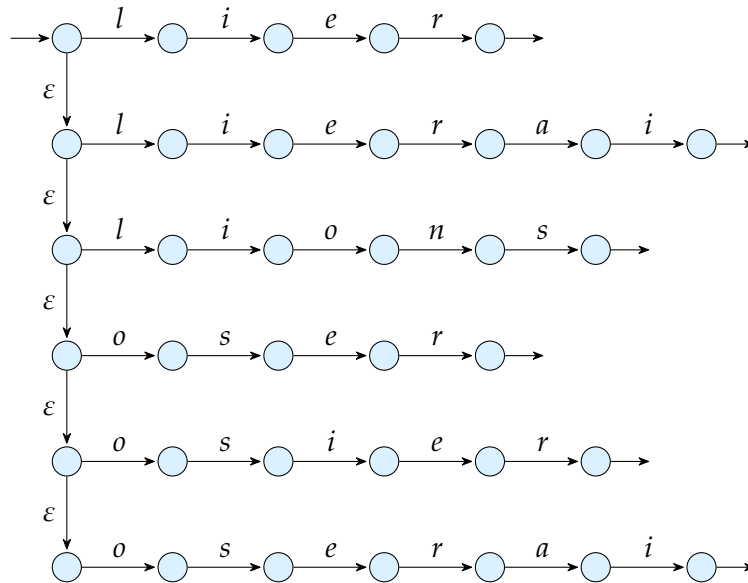


Automate 4.24 – L'automate minimal de  $(a + b)a^*ba^*b(a + b)^*$

 Pour minimiser un automate, utiliser '`automaton.minimize`'. Les états sont étiquetés par les états de l'automate d'origine.

**Exercice 4.35** (BCI-0607-2-1). On s'intéresse, dans cet exercice, à la représentation de grands dictionnaires par des automates finis. Nous considérons ici qu'un dictionnaire est une liste finie de mots : il existe donc un automate fini qui le représente. Une telle représentation garantit en particulier que la recherche d'un mot aura une complexité linéaire en la longueur du mot recherché.

Considérons la liste de mots  $L = \{\text{lier, lierai, lions, oser, osier, oserai}\}$ , qui correspond au langage de l'automate 4.25 (sur laquelle les noms des états sont omis).



Automate 4.25 – Un petit dictionnaire

1. Construisez, par une méthode vue en cours, un automate déterministe pour le langage  $L$ . Vous procéderez en deux temps, en prenant soin de détailler chacune des étapes : suppression des transitions  $\epsilon$ , puis déterminisation de l'automate sans transition  $\epsilon$ .
2. On dit que deux états  $p$  et  $q$  d'un automate déterministe sont indistinguables si pour tout mot  $u$  de  $\Sigma^*$ ,  $\delta^*(p, u) \in F \Leftrightarrow \delta^*(q, u) \in F$ .

Autrement dit, deux états  $p$  et  $q$  sont indistinguables si et seulement si tout mot conduisant à un calcul réussi débutant en  $p$  conduit à un calcul réussi débutant en  $q$  et réciproquement.

- (a) Identifiez dans l'automate déterministe construit à la question 1 un couple d'états indistinguables ; et un couple d'états distinguables (non-indistinguables).
  - (b) La relation d'indistinguabilité est une relation binaire sur l'ensemble des états d'un automate. Prouvez que c'est une relation d'équivalence (i.e. qu'elle est réflexive, symétrique, et transitive).
3. Soit  $A = (\Sigma, Q, q_0, F, \delta)$  un automate fini déterministe sans état inutile, la combinaison de deux états indistinguables  $p$  et  $q$  produit un automate  $A'$  identique à  $A$ , à ceci près que  $p$  et  $q$  sont remplacés par un état unique  $r$ , qui « hérite » de toutes les propriétés (d'être initial ou final) et transitions de l'un des deux états. Formellement,  $A'$  est défini par  $A' = (\Sigma, Q \setminus \{p, q\} \cup \{r\}, q'_0, F', \delta')$  avec :

—  $q'_0 = r$  si  $q = q_0$  ou  $p = q_0$ ,  $q'_0 = q_0$  sinon ;

- $F' = F \setminus \{p, q\} \cup \{r\}$  si  $p \in F$ ,  $F' = F$  sinon ;
- $\forall a \in \Sigma, \forall e \in Q \setminus \{p, q\}, \delta'(e, a) = r$  si  $\delta(e, a) = p$  ou  $\delta(e, a) = q$ ;  $\delta'(e, a) = \delta(e, a)$  si  $\delta(e, a)$  existe;  $\delta'(e, a)$  est indéfini sinon.
- $\forall a \in \Sigma, \delta'(r, a) = \delta(p, a)$  si  $\delta(p, a)$  existe,  $\delta'(r, a)$  est indéfini sinon.

À partir de l'automate construit à la question 1, construisez un nouvel automate en combinant les deux états que vous avez identifiés comme indistinguables.

4. Prouvez que si  $A$  est un automate fini déterministe, et  $A'$  est dérivé de  $A$  en combinant deux états indistinguables, alors  $L(A) = L(A')$  (les automates sont équivalents).
5. Le calcul de la relation d'indistinguabilité s'effectue en déterminant les couples d'états qui sont distinguables; ceux qui ne sont pas distinguables sont indistinguables. L'algorithme repose sur les deux propriétés suivantes :

**[initialisation]** si  $p$  est final et  $q$  non final, alors  $p$  et  $q$  sont distinguables;

**[itération]** si  $\exists a \in \Sigma$  et  $(p', q')$  distinguables tq.  $\delta(q, a) = q'$  et  $\delta(p, a) = p'$ , alors  $(p, q)$  sont distinguables.

Formellement, on initialise l'ensemble des couples distinguables en utilisant la propriété [initialisation]; puis on considère de manière répétée tous les couples d'états non encore distingués en appliquant la clause [itération], jusqu'à ce que l'ensemble des couples distinguables se stabilise.

Appliquez cet algorithme à l'automate déterministe construit à la question 1.

6. Répétez l'opération de combinaison en l'appliquant à tous les états qui sont indistinguables dans l'automate 4.25.

Le calcul de la relation d'indistinguabilité est à la base de l'opération de minimisation des automates déterministes, cette opération permet de représenter les dictionnaires — même très gros — de manière compacte; il permet également de dériver un algorithme très simple pour tester si deux automates sont équivalents.



**Deuxième partie**

**Travaux Dirigés et Pratiques**  
**Devoirs à la Maison**  
**Annales**

---



## Chapitre 5

# Travaux Dirigés

Ces sujets ont été écrits par Alexandre Duret-Lutz.



# TD 1

## Preuves, calculabilité et distances

Version du 26 septembre 2016

### Exercice 1 – Preuves par récurrence

1. (**Récurrence simple pour s'échauffer**) Montrez par récurrence que séparer les  $n$  carrés d'une plaque de chocolat demande de casser la plaque  $n - 1$  fois (peu importe l'ordre dans lequel on choisit les rainures).
2. (**Attention au(x) cas de base**) Montrez que tout entier naturel supérieur ou égal à 8 peut s'écrire comme  $3a + 5b$ , avec  $a, b \in \mathbb{N}$ . (e.g.,  $19 = 3 + 3 + 3 + 5 + 5$ .)
3. (**Récurrence structurelle**) Soit un arbre défini de la façon suivante :
  - Un *nœud* isolé est un arbre et c'est aussi sa racine. Le degré de ce nœud est 0.
  - À partir d'une liste d'arbres  $A_1, A_2, \dots, A_n$  un nouvel arbre peut être construit de la façon suivante : on crée un nouveau nœud  $N$  qui sera la racine de cet arbre, et on le relie par des arcs aux racines de chacun des  $A_i$ .  $n$  est le *degré* du nœud  $N$ .
  - a) Montrez que tout arbre a un nœud de plus qu'il n'a d'arcs.
  - b) On considère un arbre dont tous les nœuds sont de degré 0 ou 2. Montrez qu'un tel arbre possède  $k$  nœuds de degré 2 si et seulement si il possède  $k + 1$  nœuds de degré 0.
4. Quel est le lien entre la question 1 et la question 3?

### Exercice 2 – Calculabilité

1. Les ensembles suivants sont-ils (a) récursivement énumérables, (b) récursifs ?
  - L'ensemble des nombres premiers.
  - L'ensemble des polynômes dont les coefficients sont des entiers naturels.
  - L'ensemble vide.
  - L'ensemble des programmes qui ne bouclent pas infiniment.
  - L'ensemble des programmes qui terminent en moins de 10s.
2. (**Un langage non récursivement énumérable**) Soit  $\Sigma$  un alphabet quelconque, notons  $m_0, m_1, m_2, \dots$  la suite des mots de  $\Sigma^*$  ordonnés par ordre militaire. Par exemple si  $\Sigma = \{a, b\}$ , on considère l'ordre  $\epsilon, a, b, aa, ab, ba, bb, \dots$   
 Notons de même  $A_0, A_1, A_2, \dots$  l'ensemble des algorithmes reconnaissant les langages récursivement énumérables de  $\Sigma^*$ , ordonnés de la même façon (ordre militaire de leur codage, i.e. leur représentation dans votre langage de programmation préféré).

Montrez (par l'absurde) que le langage  $L$  défini ci-après n'est pas récursivement énumérable.

$$L = \{m_i \mid A_i \text{ ne reconnaît pas } m_i\}$$

### Exercice 3 – Distance préfixe

Soient  $u, v$ , et  $w$  trois mots quelconques construits sur un alphabet  $\Sigma$ . Rappelons que  $\text{plpc}(u, v)$  désigne le plus long préfixe commun à  $u$  et  $v$ .



1. Justifiez que  $|\text{plpc}(\text{plpc}(u, v), \text{plpc}(v, w))| \leq |\text{plpc}(u, w)|$ .
2. Justifiez que  $|\text{plpc}(u, v)| + |\text{plpc}(v, w)| \leq \min(|\text{plpc}(u, v)|, |\text{plpc}(v, w)|) + |v|$ .
3. Justifiez que  $|\text{plpc}(\text{plpc}(u, v), \text{plpc}(v, w))| = \min(|\text{plpc}(u, v)|, |\text{plpc}(v, w)|)$ .
4. Dédisez-en que  $|\text{plpc}(u, v)| + |\text{plpc}(v, w)| \leq |\text{plpc}(u, w)| + |v|$
5. Montrez que  $d_p(u, v) = |uv| - 2|\text{plpc}(u, v)|$  est une distance.  
(Les questions précédentes vous serviront à prouver l'inégalité triangulaire.)

#### Exercice 4 – Distance d'édition

Rappel : la distance d'édition (ou de Levenshtein) entre  $u$  et  $v$  est le plus petit nombre d'opérations d'édition élémentaires (insertion ou suppression de symbole) à effectuer pour passer de  $u$  à  $v$ .

1. Proposez une définition récursive de cette distance.

$$d_L(u, v) = ?$$

2. Montrez qu'il s'agit bien d'une distance.

## TD 2 Expressions rationnelles

Version du 26 septembre 2016

### Exercice 1 – Opérateurs basiques

Dans cet exercice nous ne considérons que les opérateurs basiques suivants :

- le choix ( $e_1 + e_2$ )
- la concaténation ( $e_1e_2$ )
- la répétition ( $e^*$ )

On pourra omettre les parenthèses superflues en respectant les priorités classiques de ces opérateurs (répétition plus prioritaire que la concaténation elle-même plus prioritaire que le choix).

Soit  $\Sigma = \{-, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$ . Proposez des expressions rationnelles reconnaissant les sous-langages de  $\Sigma^*$  qui suivent.

1. Les entiers signés en base 10. C'est-à-dire avec le « - » en première position s'il apparaît, et pas de 0 en tête (sauf pour représenter 0).
2. Les nombres à virgule.
3. Les développements décimaux d'un nombre réel, comme 3.141592, -318.29 ou 42. Trois contraintes pour corser :
  - on n'acceptera pas un point qui n'est pas suivi de chiffre,
  - à nouveau la partie entière ne peut pas commencer par 0 sauf pour les nombres compris entre -1 et 1,
  - on n'acceptera pas -0.
4. Tous les entiers naturels multiples de 20.

### Exercice 2 – Sucre syntaxique

Autorisons-nous les opérateurs suivants en plus des opérateurs basiques.

- Pour une expression rationnelle  $e$ ,  $e^2$  est l'abréviation de  $(\varepsilon + e)$ .
- Pour une expression rationnelle  $e$ ,  $e^+$  est l'abréviation de  $ee^*$ .
- Pour des symboles  $s_1, s_2, \dots, s_n$ ,  $[s_1s_2 \dots s_n]$  désigne l'un de ces symboles. Cet opérateur peut facilement se réécrire avec l'opérateur  $+$ . Par exemple si  $\Sigma = \{a, b, \dots, z\}$ , on a  $[aeiou] = (a + e + i + o + u)$ .
- Si les symboles de  $\Sigma$  sont ordonnés (par exemple les chiffres ou notre alphabet latin)  $[s_1 - s_2]$  représente un symbole parmi tous ceux compris entre le symbole  $s_1$  et le symbole  $s_2$  (inclus). Cet opérateur peut lui aussi se réécrire, par exemple si  $\Sigma = \{a, b, \dots, z\}$ , on a  $[a - e] = (a + b + c + d + e)$ .

1. Simplifiez toutes les expressions de l'exercice précédent avec ces opérateurs.
2. Avec  $\Sigma = \{-, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., e\}$ , proposez une expression rationnelle reconnaissant un nombre décimal en notation scientifique, c'est-à-dire de la forme  $-1.234e56$  où
  - « - » est le signe, il peut être absent
  - « 1.234 » est la mantisse, comprise entre 0 et 9.99999 ...
  - «  $e56$  » est l'exposant, il est facultatif et s'interprète comme  $10^{56}$ . L'exposant est un nombre entier qui peut être signé, par exemple  $2e-3$  représente 0.002.
 On s'autorise les 0 superflus, ainsi que -0, vous avez compris que c'était suffisamment pénible à gérer.

**Exercice 3 – Simplification et équivalences**

Pour chaque entrée de la liste suivante, dites si le langage dénoté par l'expression rationnelle  $e$  est égal, inclus, contenant, ou incomparable à celui dénoté par l'expression rationnelle  $f$  pour l'alphabet  $\Sigma = \{a, b, c\}$ . Proposez des contre-exemples quand les langages sont différents.

$e$	$f$
$a^*b(ab)^*$	$a^*(bab)^*$
$a(bb)^*$	$ab^*$
$a(a+b)^*b$	$a^*(a+b)^*b^*$
$abc+acb$	$a(b+c)(c+b)$
$a^*bc+a^*cb$	$a^*(bc+a^*cb)$
$(abc+acb)^*$	$((abc)^*(acb)^*)^*$
$(abc+acb)^+$	$((abc)^*(acb)^*)^+$
$(abc+acb)^*$	$(abc(acb)^*)^*$
$(abc+acb)^*$	$(a(bc)^*(cb)^*)^*$

**Exercice 4 – Intersection de langages**

1. L'intersection de deux langages rationnels est-elle un langage rationnel?
2. Soient  $L_1$  et  $L_2$  les langages respectivement dénotés par  $ab+bc^+$  et  $a^*b^*c^*$ . Proposez une expression rationnelle dénotant le langage  $L_1L_2 \cap L_2L_1$ .

**Exercice 5**

Parmi les langages suivants, déterminez ceux qui sont égaux.

$$(L \cup M)^* \quad (LM)^*L \quad L(LM)^* \quad (L^* \cup M)^* \quad (M^* \cup L)^* \quad (L^*M^*)^* \quad (M^*L^*)^* \quad (L^* \cup M^*)^*$$

## TD 3 Automates finis

Version du 26 septembre 2016

### Exercice 1 – Reconnaître une liste

Si  $U$  est un alphabet fini, on appelle *liste* de  $U$  une séquence débutant par le symbole '(', finissant par le symbole ')', et contenant des éléments de  $U$  séparés par des ':'. Ainsi par exemple : (1 : 2 : 3 : 2 : 1) est une liste d'éléments de  $U = \{1, 2, 3\}$ . Dans tout l'exercice, on considérera qu'une liste contient toujours au moins un élément : *il n'y a pas de liste vide*.

1. Est-il possible de reconnaître l'ensemble des listes d'éléments de  $U = \{1, 2, 3\}$  avec un automate fini? Si votre réponse est positive, donnez une représentation graphique d'un automate reconnaissant ce langage, en indiquant précisément l'alphabet, l'état initial et le ou les états finaux. Les automates les plus simples seront préférés.
2. Dans le cas où  $U$  est totalement ordonné, on s'intéresse aux listes croissantes : tout élément de la liste doit être supérieur ou égal à son prédécesseur. Peut-on reconnaître ces listes avec un automate fini? Justifiez une réponse positive en représentant un tel automate dans le cas où  $U = \{1, 2\}$ .
3. Un informaticien pervers s'intéresse au cas où  $U$  contient les symboles spéciaux '(', ')' et ':'. Que se passe-t-il dans ce cas-là? Faut-il modifier la forme de l'automate de la question 1 lorsqu'on ajoute à  $U$  ces trois symboles? Si oui, comment?
4. La notion de liste est étendue récursivement pour inclure les listes de listes, les listes de listes de listes... comme ((1 : 3) : 3 : (2 : 1) : ((1 : 2))). Est-il possible de reconnaître ces listes avec un automate fini? Justifiez une réponse positive en représentant un tel automate dans le cas où  $U = \{1, 2\}$ .

### Exercice 2 – Distributeur de boissons

On modélise un distributeur de boissons par un automate fini. L'alphabet d'entrée  $\Sigma = \{d, v, c\}$  correspond aux pièces de 10, 20 et 50 centimes acceptées par le distributeur. Une boisson coûte 50 centimes.

1. Construisez un automate *A déterministe* reconnaissant toutes les séquences de pièces dont le total est 50 centimes. Vous pourrez éventuellement commencer par proposer un automate non-déterministe reconnaissant toutes ces séquences.
2. Ce distributeur est primitif : il ne rend pas la monnaie. Pour l'en rendre capable, nous allons compléter l'automate pour qu'il puisse non seulement « lire » (engranger) des pièces, mais également en « écrire » (en rendre). On parle alors de *transducteur fini*. Concrètement, chaque transition de votre nouvelle machine sera étiquetée par une paire entrée/sortie, notée  $a/b$ . Emprunter une transition étiquetée  $a/b$  s'interprète comme : *lire un a et écrire un b*.

En partant de l'automate fini de la question précédente, construire un transducteur fini  $T$  simulant un distributeur qui rendrait la monnaie : pour toute séquence en entrée dont le total excède 50 centimes, votre transducteur doit produire en sortie une séquence dont la somme est la différence entre le montant entré et cinquante centimes.

Illustrez son fonctionnement sur la séquence de pièces :  $dvcv$ .

**Exercice 3 – Traduction d'expressions rationnelles**

Dans cet exercice, on suppose que  $\Sigma = \{a, b, c\}$ .

Pour chacune des expressions rationnelles suivantes, utiliser l'algorithme de Thompson pour construire un automate fini non-déterministe avec transitions spontanées, puis appliquer la construction vue en cours pour faire disparaître les transitions spontanées.

1.  $c(ab + c)$
2.  $(a + (cc)^*)(b + c)$
3.  $((ab + \varepsilon)^*c)^*$

# TD 4 Lemme de pompage et déterminisation

Version du 26 septembre 2016

### Exercice 1 – Listes de listes de listes...

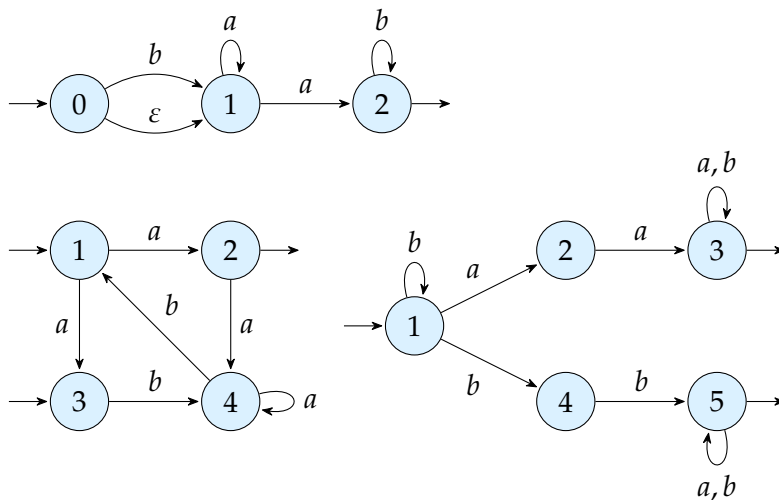
Reprenons une question du TD précédent :

La notion de liste est étendue récursivement pour inclure les listes de listes, les listes de liste de liste... comme  $((1 : 3) : 3 : (2 : 1) : ((1 : 2)))$ . Est-il possible de reconnaître les listes avec un automate fini ?

1. Utilisez le lemme de pompage pour les langages rationnels afin de démontrer que le langage  $L_p = \{(^n 1)^n \mid n \geq 0\}$  n'est pas rationnel.
2. Déduisez-en qu'il n'est pas possible de reconnaître le langage  $L_l$ , composé de listes, listes de listes, listes de listes de listes... avec un automate fini.

### Exercice 2

On suppose  $\Sigma = \{a, b\}$ . En utilisant les méthodes du cours, construisez un automate déterministe équivalent à chacun des automates suivants :



### Exercice 3 – Recherche de motifs

Dans cet exercice on considère  $\Sigma = \{a, b, c\}$ .

1. Soit le mot  $m = abab$  et  $L$  le langage des mots qui ont  $m$  comme suffixe.  $L$  contient les mots de la forme  $v = um$ , avec  $u \in \Sigma^*$  ; soit, par exemple, les mots  $aaabab$  et  $babab$ . En revanche,  $caabc$  n'appartient pas à  $L$ . Prouvez que  $L$  est rationnel. Proposez un automate fini *non-déterministe*  $A_n$  pour  $L$ . Vous justifierez votre construction.
2. Transformez  $A_n$  en un automate déterministe complet  $A_d$  équivalent, en utilisant une construction étudiée en cours. Vous explicitez les étapes de l'application de l'algorithme.
3. Pourquoi est-il évident que  $A_d$  soit complet et émondé ?

4. On modifie l'alphabet avec  $\Sigma = \{a, b, c, d, e\}$  pour cette question uniquement. Comment répercuter cette modification sur  $A_d$ ?

5. On considère l'algorithme suivant :

```
//  $u = u_1 \dots u_n$  est le mot dans lequel on cherche.
//  $A_d = (\Sigma, Q, \{q_0\}, F, \delta)$  est un automate déterministe pour  $L$ .
 $q \leftarrow q_0$ 
 $i \leftarrow 1$ 
 $c \leftarrow 0$ 
tant que ( $i \leq n$ ) faire
     $q \leftarrow \delta(q, u_i)$ 
     $i \leftarrow i + 1$ 
    si ( $q \in F$ ) alors  $c \leftarrow c + 1$  fin si
fin tant que
```

- Illustrez le fonctionnement de cet algorithme lorsque  $u = bcababcabbababac$  et l'automate  $A_d$  calculé à la question 2. Vous donnerez pour chaque passage dans la boucle principale la valeur de  $c$ .
- Que calcule cet algorithme en général? Justifiez votre affirmation.
- Quelle est la complexité de cet algorithme?
- Que vaut  $c$  à la fin de l'exécution de l'algorithme pour  $u = cabbababababc$ ? Que remarquez vous?
- Comment faudrait-il modifier l'automate  $A_d$  pour compter que le nombre maximal d'occurrences disjointes du motif au sein de la chaîne d'entrée? Par exemple la réponse devrait être 2 dans le dernier exemple.

# TD 5

## Stabilité des langages rationnels

Version du 26 septembre 2016

### Exercice 1 – Négation d’expression rationnelle

Posons  $\Sigma = \{a, b\}$ . Soit  $L$  le langage dénoté par l’expression rationnelle  $a^*(ba^*ba^*ba^*)^*$ . Notre but est de construire une expression rationnelle dénotant le langage complémentaire  $\bar{L} = \Sigma^* \setminus L$ .

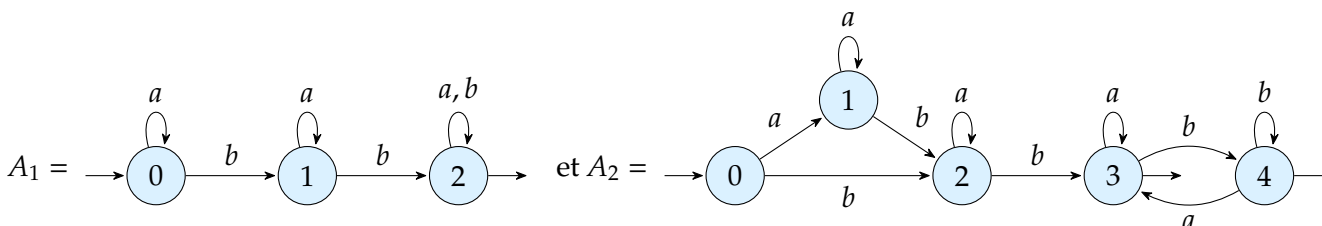
1.  $\bar{L}$  est-il forcément rationnel? Justifiez votre réponse.
2. Proposez un automate fini déterministe  $A_L$  reconnaissant  $L$ .
3. Donnez  $\bar{A}_L$ , l’automate complémentaire de  $A_L$ .
4. Appliquez l’algorithme de Brzozowski et McCluskey présenté en cours pour construire l’expression rationnelle correspondant à l’automate  $\bar{A}_L$ .
5. Le complémentaire construit est-il toujours valide si  $\Sigma = \{a, b, c\}$ ? Dans la négative, que faut-il changer à notre procédure de complémentation d’expression rationnelle pour qu’il le soit?

### Exercice 2 – Relations entre langages rationnels

1. Soit deux langages rationnels  $L_1$  et  $L_2$  tels que  $L_2 \subset L_1$ . Le langage  $L_1 \setminus L_2$  est-il rationnel?
2. Soient deux langages  $L_1$  et  $L_2$  tels que  $L_2 \subset L_1$ . Si l’on sait que  $L_2$  est rationnel, peut-on dire que  $L_1$  l’est aussi? Justifiez votre réponse.

### Exercice 3 – Intersection de langages rationnels

On considère les deux automates suivants :



L’objectif est de montrer que ces deux automates sont équivalents en calculant  $\overline{L(A_1)} \cap L(A_2)$  et  $L(A_1) \cap \overline{L(A_2)}$ .

1. Que doivent valoir  $\overline{L(A_1)} \cap L(A_2)$  et  $L(A_1) \cap \overline{L(A_2)}$  si les automates sont équivalents?
2. Calculez  $\bar{A}_1$  et  $\bar{A}_2$ . Vous émonderez ces automates.
3. Pour deux automates non-déterministes  $A = (\Sigma, Q, Q_0, F, \delta)$  et  $A' = (\Sigma, Q', Q'_0, F', \delta')$ , le produit synchronisé  $A \& A'$  est l’automate  $(\Sigma, Q^\&, Q_0^\&, F^\&, \delta^\&)$  défini par :
  - $Q^\& = Q \times Q'$ ,
  - $Q_0^\& = Q_0 \times Q'_0$ ,
  - $F^\& = F \times F'$ ,
  - $\delta^\& = \{(s, s'), l, (d, d')\} \in Q^\& \times \Sigma \times Q^\& \mid (s, l, d) \in \delta \text{ et } (s', l, d') \in \delta'\}$ .



Avec cette définition il est facile de voir que les mots reconnus par  $A \& A'$  sont des mots à la fois de  $A$  et de  $A'$ . En fait on a  $L(A \& A') = L(A) \cap L(A')$ .

Utilisez cette définition pour calculer les automates  $A_1 \& \overline{A_2}$  et  $A_2 \& \overline{A_1}$ .

4. Qu'en conclure sur l'équivalence de  $A_1$  et  $A_2$  ?
5. Utilisez l'algorithme de minimisation présenté en cours pour réduire l'automate  $A_2$ . Vous indiquerez la partition des états de l'automate à chaque itération de l'algorithme.

#### Exercice 4 – Un langage difficile à définir

1. Posons  $\Sigma = \{a, b\}$ . Soit  $L$  un langage rationnel sur  $\Sigma$ . En utilisant des notations ensemblistes (sur les langages) ou des expressions rationnelles, comment définiriez-vous le langage  $L'$  rassemblant tous les mots qui possèdent **exactement un** facteur dans le langage  $L$ ?  
Par exemple si  $L = \{ab, ba\}$ , alors  $\underline{aabb} \in L'$ ,  $\underline{bbbba} \in L'$ , mais  $\underline{aabbba} \notin L'$ .  
(Indice : essayez la différence ensembliste.)
2. Ce langage est-il rationnel ?



## Chapitre 6

# Devoirs à la Maison

Ces sujets ont été écrits par Alexandre Duret-Lutz.



# DM 1

## Langages

Version du 26 septembre 2016

Ce devoir à la maison est à rendre demain, mardi, au début du TD. C'est un vrai devoir, à rédiger proprement sur copie, et à rendre impérativement en TD. Une absence de rendu est une absence tout court; cela nous évite de faire l'appel. Vous aurez chaque jour un DM à rendre pour le lendemain, et nous estimons qu'ils demandent une à deux heures de travail chacun (relecture du cours incluse). Certains exercices de ces DM peuvent être plus faciles à résoudre en utilisant des résultats qui n'ont pas encore été vus en cours : vous avez le droit (de lire le poly pour prendre de l'avance et) d'utiliser ces résultats.

### Exercice 1 – Foire aux opérateurs

Cette question utilise les notations introduites dans la section « Opérations ensemblistes » du polycopié. L'alphabet utilisé est toujours  $\Sigma = \{a, b\}$ .

Redéfinissez chacun des langages  $L_i$  suivants en n'utilisant que des ensembles finis (de mots ou lettres) que vous combinerez avec les opérateurs  $\cup$ ,  $\star$ , et la concaténation. Les autres opérateurs ( $\cap$ ,  $\overline{\quad}$ , *Suff*, *Pref*, *Fac*...) ne sont pas autorisés.

Par exemple *Pref*( $\{a\}b^*$ ) peut se réécrire  $\{\varepsilon\} \cup \{a\} \cup \{a\}b^*$ . On se débarrasse ainsi de l'opérateur *Pref*.

$$L_1 = \text{Suff}(\{a\}b^*)$$

$$L_2 = \text{Fac}(\{a\}b^*)$$

$$L_3 = (\{a\}b^*a^*) \cap (\{a\}^*b^*a)$$

$$L_4 = \overline{\{a\}^*}$$

$$L_5 = \overline{\{a\}b^*} \cap \{a\}^*$$

### Exercice 2 – Langage préfixe

On dit que  $L$  est un *langage préfixe* si  $\forall (u, v) \in L^2, u \neq v \implies u \notin \text{Pref}(v)$ . Autrement dit si aucun des mots de  $L$  n'est préfixe d'un autre mot de  $L$ .

Pour deux langages préfixes  $L_1$  et  $L_2$ , démontrez (rigoureusement, cela va de soi) que :

1.  $\varepsilon \in L_1 \implies L_1 = \{\varepsilon\}$
2.  $L_1 \cap L_2$  est préfixe
3.  $L_1L_2$  est préfixe
4.  $L_1 \cup L_2$  n'est pas forcément préfixe

### Exercice 3 – Digicode pour matheux

Imaginez un digicode équipé de 11 touches : les dix chiffres « 0 »...« 9 » plus une touche « E » (comme « Entrée », pour valider la saisie).

Vous devez programmer ce digicode de façon à ce que la porte ne s'ouvre que lorsque les deux conditions suivantes sont remplies :

- la touche « E » vient d'être enfoncée
- le nombre formé de *tous* les chiffres tapés avant « E » (c'est-à-dire ceux tapés depuis le précédent « E » ou depuis que le digicode a été mis en route) est divisible par 7.

Le nombre saisi peut être aussi long que l'on veut, sans limite de taille. Par exemple « 123456789123456789E » ouvrira la porte, car 123456789123456789 est un multiple de 7. En revanche « 123456789E » ne doit pas ouvrir la porte. Il n'y a donc pas un seul code qui ouvre la porte, mais une infinité.

À la fin de la semaine, vous serez capable de dire à table une phrase du genre « l'ensemble (infini) de tous les codes acceptés par ce digicode est un langage rationnel, donc il peut être reconnu par une machine avec une mémoire de taille constante »<sup>1</sup>. Bien sûr, il existe des codes acceptés qui sont aussi grands que l'on souhaite, plus grands que la RAM de la machine, par exemple. Il doit donc exister des algorithmes qui n'ont pas besoin de voir tous les chiffres en même temps.

Notre cas est en fait plus contraint : les touches sont frappées une à une, et vous ne connaissez pas à l'avance la taille du nombre qui sera entré. Comme le digicode est implémenté sur un micro-contrôleur avec très peu de RAM, n'oubliez même pas stocker toute la chaîne de caractères et attendre le « E » pour la parcourir à nouveau. Il vous faut traiter les chiffres au fur et à mesure qu'ils arrivent, et trouver<sup>2</sup> un moyen de n'utiliser que quelques octets (pas plus de deux ou trois variables) pour retenir l'état du digicode. Le peu de RAM exclut aussi les appels récursifs<sup>3</sup>.

Complétez le squelette suivant, représentant le programme exécuté par le digicode. Il est donné en C, mais vous avez le droit d'utiliser un autre langage tant que vous en conservez le principe.

```
// variables globales (à compléter si besoin).
...

for (;;) // boucle infinie.
{
    // get_key attend une touche puis renvoie une valeur entre 0 et 9, ou -1 pour "entrée".
    int key = get_key();

    if (key == -1)
    {
        if (/* condition à remplir */)
        {
            open_door();
        }
        // dans tous les cas, les prochains chiffres font partie d'un nouveau nombre.

        ...
    }
    else // 0 <= key <= 9.
    {
        // assimiler le nouveau chiffre.

        ...
    }
}
```

1. Ou l'inverse. De toute façon, vos convives vous auront déjà pris pour un fou.

2. Ça implique de commencer par chercher, mais les Epitéens sont forts pour cela.

3. Ce serait une façon cachée d'utiliser de la mémoire dynamique, donc non bornée.

## DM 2 Expressions rationnelles

Version du 26 septembre 2016

Ce devoir à la maison est à rendre demain, mercredi, au début du TD.

### Exercice 1 – Décodage de digicode

On a trouvé un digicode du même modèle que celui utilisé dans le précédent DM (c'est-à-dire 10 touches numérotées de « 0 » à « 9 » plus une touche « E » servant à valider la saisie).

Le digicode est déjà programmé, mais le code qui permet d'ouvrir la porte a été oublié. Il vous faut donc faire le travail inverse du DM précédent : retrouver le code secret à partir du programme. Attention, ce digicode accepte *plusieurs* codes !

Voici le programme extrait du micro-contrôleur :

```

/* Pour dissiper les doutes sur la lecture de ce tableau, on a tab[9][2] == 5. */
int tab[10][10] =
{
    /* 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 */
    /* 0 */ { 0, 7, 0, 0, 0, 0, 0, 0, 0, 0 },
    /* 1 */ { 0, 7, 0, 0, 0, 0, 0, 0, 0, 0 },
    /* 2 */ { 1, 7, 0, 0, 0, 0, 0, 0, 0, 0 },
    /* 3 */ { 0, 8, 0, 0, 0, 0, 0, 0, 0, 0 },
    /* 4 */ { 0, 8, 0, 0, 0, 0, 0, 0, 0, 0 },
    /* 5 */ { 0, 7, 1, 0, 0, 0, 0, 0, 0, 0 },
    /* 6 */ { 0, 7, 0, 0, 1, 0, 0, 0, 0, 0 },
    /* 7 */ { 6, 9, 5, 3, 2, 0, 0, 0, 0, 0 },
    /* 8 */ { 6, 9, 5, 3, 2, 0, 0, 0, 0, 0 },
    /* 9 */ { 6, 9, 5, 4, 2, 0, 0, 0, 0, 0 }
};

int pos = 0;
for (;;) /* boucle infinie */
{
    int key = get_key();
    if (key == -1) /* touche 'E' */
    {
        if (pos == 1 || pos == 4 || pos == 8)
        {
            open_door();
        }
        pos = 0;
    }
    else /* 0 <= key <= 9 */
    {
        pos = tab[pos][key];
    }
}

```

1. Que nous apprennent les colonnes 5 à 9 du tableau ?

2. Donnez deux séquences de touches permettant d'ouvrir la porte.
3. Justifiez que l'ensemble  $L$  des séquences qui ouvrent la porte est infini.
4. Donnez une expression rationnelle dénotant  $L$ . Note : utilisez l'alphabet  $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ , c'est-à-dire sans y faire apparaître le « E » qui ne nous sert en fait qu'à indiquer la fin d'un mot au digicode.

### Exercice 2

Pour tout alphabet  $\Sigma$ , toute lettre  $a \in \Sigma$ , tous langages  $A$ ,  $L$  et  $M$  sur  $\Sigma$  :

1. Justifiez que  $\{a\}.L = \{a\}.M \implies L = M$ .
2. Prouvez que  $AL = AM \not\Rightarrow L = M$ .
3. Prouvez que  $L^* = M^* \not\Rightarrow L = M$ .
4. Prouvez que  $\forall n > 1, L^n \neq \{u^n \mid u \in L\}$ .
5. Prouvez que  $\forall n > 1, L^n = M^n \not\Rightarrow L = M$ .

### Exercice 3 – Expressions rationnelles

Donnez une expression rationnelle pour les langages suivants :

1. Les mots de  $\{a, b\}^*$  contenant un nombre pair de  $a$ .
2. Les mots de  $\{a, b, c\}^*$  contenant exactement 2 ou 3 fois la lettre  $c$ .
3. Les mots de  $\{a, b, c\}^*$  dans lesquels  $c$  n'apparaît jamais à gauche d'un  $b$ .

## DM 3 Automates

Version du 26 septembre 2016

Ce devoir à la maison est à rendre demain, jeudi, au début du TD.

### Exercice 1 – Traduction d’expressions rationnelles

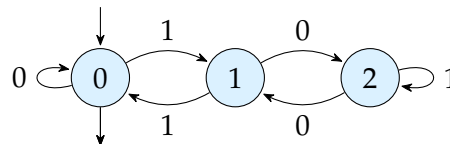
Dans cet exercice, on suppose que  $\Sigma = \{a, b, c\}$ .

Pour chacune des expressions rationnelles suivantes, on vous demande d’utiliser l’algorithme de Thompson pour construire un automate fini non-déterministe à transitions spontanées, puis d’appliquer la construction vue en cours pour éliminer les transitions spontanées, et enfin de supprimer les éventuels états inutiles<sup>1</sup>. Vous montrerez les automates après chaque étape.

1.  $(a + b + cc)^* abab$
2.  $((ab + \varepsilon)^* c)^*$
3.  $(\emptyset(a + b))^*$

### Exercice 2 – Multiples de 3 et 7

Soit  $\mathcal{D}_3$  l’automate déterministe suivant défini sur l’alphabet  $\Sigma = \{0, 1\}$  :



1. Exécutez  $\mathcal{D}_3$  sur les mots 101010, et 11111.
2. Démontrez que  $\mathcal{D}_3$  reconnaît les représentations binaires des entiers naturels multiples de 3. (Indice : donnez un sens aux numéros des états.)
3. Construisez  $\mathcal{D}_7$  un automate déterministe qui reconnaisse les représentations binaires des entiers naturels multiples de 7.

### Exercice 3 – Automate pour digicode

Vous aurez compris que le programme du digicode d’hier (DM 2, exercice 1) ne faisait qu’exécuter un automate. Aujourd’hui nous allons modifier le tableau `tab` du programme du DM 2 pour reconnaître un autre code.

1. Le tableau `tab` du DM précédent représente-t-il un NFA ? un DFA ?
2. Comment est indiqué l’état initial, et les états finaux ?
3. On veut changer le code afin que le digicode n’accepte que les séquences de chiffres qui se terminent par 747. Donnez une expression rationnelle dénotant l’ensemble  $L$  de ces séquences.
4. À l’aide de l’algorithme de Thompson, construisez un  $\varepsilon$ -NFA reconnaissant  $L$ . Puis supprimez les transitions spontanées pour obtenir un NFA. Émondez si besoin. Salez, poivrez.
5. Construisez un DFA reconnaissant  $L$ . On ne vous dit pas comment ; tous les coups sont permis. (Indice : 4 états sont suffisants.)
6. Donnez une nouvelle implémentation du programme du digicode, reconnaissant les mots de  $L$  (suivis de « E », comme d’habitude) sur le même principe que le DM 2.

---

1. Cf. la section *États utiles* du poly.



# DM 4 Automates

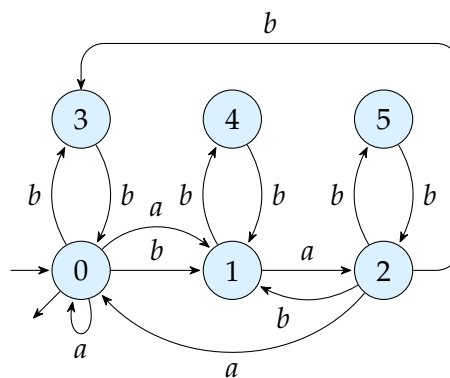
Version du 26 septembre 2016

Ce dernier<sup>1</sup> devoir à la maison est à rendre demain, vendredi, au début du TD.

## Exercice 1 – Minimisation de Brzozowski

**Soyez très méticuleux dans cet exercice.** Comptez le nombre de  $a$  et de  $b$  lorsque vous recopiez un automate du brouillon vers la copie; comptez les flèches entrantes et sortantes de chaque état; n’oubliez pas de marquer les états initiaux et finaux. Le moindre oubli est fatal lorsqu’on enchaîne les opérations comme ici.

Notons  $\mathcal{A}$  l’automate non-déterministe suivant :



Le transposé de  $\mathcal{A}$ , noté  $T(\mathcal{A})$ , est l’automate dans lequel toutes les flèches de  $\mathcal{A}$  ont été retournées (même les états initiaux sont devenus finaux et vice-versa).

Le déterminisé de  $\mathcal{A}$ , noté  $Det(\mathcal{A})$ , est le DFA obtenu à partir de  $\mathcal{A}$  en utilisant l’algorithme de détermination du cours (qui est aussi un théorème du poly...).

1. Construisez  $\mathcal{A}' = Det(T(\mathcal{A}))$ .

2. Construisez  $\mathcal{A}'' = Det(T(\mathcal{A}'))$ .

Note :  $\mathcal{A}''$  possède 3 états. Si vous trouvez autre chose vous avez fait une erreur!<sup>2</sup>

3. Justifiez que  $\mathcal{A}$  et  $\mathcal{A}''$  reconnaissent le même langage.

Ne soyez pas surpris que l’automate  $\mathcal{A}''$  soit plus petit que l’automate  $D(\mathcal{A})$  (que vous pouvez construire au brouillon si le cœur vous en dit) et même, dans notre cas, plus petit que  $\mathcal{A}$ . En chaînant ces deux « co-déterminisations » vous avez construit un DFA équivalent à  $\mathcal{A}$  de taille minimale : il n’en existe pas avec moins d’états.

## Exercice 2 – Conversion d’automates en expressions rationnelles

Soit  $q$  et  $r$  deux expressions rationnelles dénotant les langages  $L(q)$  et  $L(r)$  de  $\Sigma^*$ . Considérons l’équation  $X = qX + r$ . Une expression rationnelle  $t$  dénotant le langage  $L(t)$  est solution de cette équation si

$$L(t) = L(q)L(t) \cup L(r) \tag{1}$$

1. Courage!

2. C’est triste, mais il vaut mieux faire les erreurs chez soi que pendant l’examen : le canapé est bien plus confortable.

1. Montrez (par récurrence sur  $n$ ) que si  $t$  est une solution de (1), alors

$$\forall n \in \mathbb{N}, L(q)^n L(r) \subset L(t) \tag{2}$$

Note : par convention  $L(q)^0 = \{\varepsilon\}$ .

2. Montrez (par récurrence sur  $n$ ) que si  $t$  est une solution de (1) alors

$$\forall n \in \mathbb{N}, L(t) \subset L(q)^n L(t) \cup L(q)^{n-1} L(r) \cup \dots \cup L(r). \tag{5}$$

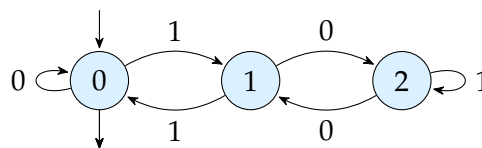
Attention à ne pas mélanger les  $r$  et les  $t$  dans l'équation précédente!

3. Si  $\varepsilon \notin L(q)$  et que  $t$  est une solution de cette équation, montrez que  $L(t) \subset L(q^*r)$ .  
 Indice : si  $\varepsilon \notin L(q)$  les mots de  $L(q)^n$  sont au moins de taille  $n$ , prenez donc chaque mot de  $L(t)$  et regardez comment vous pouvez choisir  $n$  dans l'équation (5).
4. Déduez des questions précédentes le théorème suivant :

**Théorème 1.** Soient  $q$  et  $r$  deux expressions rationnelles telles que  $\varepsilon \notin L(q)$ ; si l'expression rationnelle  $t$  est une solution de l'équation  $L(t) = L(q)L(t) \cup L(r)$ , alors  $L(q^*r) = L(t)$ .

Même si plusieurs expressions  $t$  peuvent définir ce même langage, nous dirons que cette solution est unique (au sens du langage) pour  $q$  et  $r$  données.

5. Si  $\varepsilon \in L(q)$ , l'équation (1) n'admet pas forcément d'unique solution. Donnez une solution  $t$  qui ne dépende ni de  $q$  ni de  $r$ .
6. **Application.** Considérons l'automate  $\mathcal{D}_3$  du DM précédent :



Nous notons  $t_i$  l'expression rationnelle dénotant le langage de tous les mots qui peuvent être acceptés par l'automate  $\mathcal{D}_3$  à partir de de l'état  $i$ . On a par exemple  $01 \in L(t_2)$  car il est possible d'atteindre un état final en lisant  $01$  à partir de l'état 2.

Nous pouvons énoncer des contraintes entre  $t_0, t_1,$  et  $t_2$  en lisant la figure. Par exemple si on rajoute un 1 en tête d'un mot reconnu par  $t_2$ , il restera reconnu par  $t_2$  à cause de la boucle sur l'état 2. De même si on ajoute un 0 en tête d'un mot reconnu par  $t_1$ , il sera cette fois-ci reconnu par  $t_2$ . En fait l'expression  $t_2$  satisfait l'équation  $t_2 = 0t_1 + 1t_2$ .

Si l'on fait cette lecture de l'automate pour tous les états, on obtient le système d'équations suivant :

$$t_0 = 0t_0 + 1t_1 + \varepsilon \tag{6}$$

$$t_1 = 0t_2 + 1t_0 \tag{7}$$

$$t_2 = 0t_1 + 1t_2 \tag{8}$$

Le  $\varepsilon$  a été ajouté à la première équation parce que l'état 0 est final :  $t_0$  accepte donc le mot vide en plus d'accepter les mots de  $t_1$  préfixés par 1 ainsi que ses propres mots préfixés par 0.

L'expression rationnelle  $t_0$ , parce qu'elle est associée à l'état initial, dénote le langage accepté par l'automate. Pour reconstruire une expression rationnelle associée à l'automate, il nous suffit<sup>3</sup> de résoudre le système d'équations (6)-(8) pour trouver  $t_0$ .

3. La seule difficulté, vraiment, c'est de bien se mettre dans la tête que nos produits sont des concaténations. La concaténation ne commute pas et n'est pas inversible.

Faisons la première étape ensemble. En remplaçant (7) dans (6) et (8) on élimine  $t_1$  de notre système. Voici une bonne chose de faite :

$$t_0 = (0 + 11)t_0 + 10t_2 + \varepsilon \quad (9)$$

$$t_2 = (00 + 1)t_2 + 01t_0 \quad (10)$$

C'est maintenant à vous de finir : **trouvez**  $t_0$ .

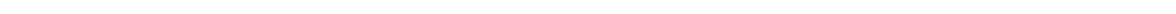
Indices : Ces deux équations sont de la forme  $t = qt + r$ . Commencez donc par appliquer le théorème 1 à l'équation (10) pour exprimer  $t_2$  en fonction de  $t_0$  uniquement, puis injectez votre résultat dans 9 avant d'appliquer à nouveau le théorème.



## Chapitre 7

# Travaux Pratiques

Ces sujets ont été écrits par Alexandre Hamez.



# TP 1

## Expressions rationnelles

Version du 26 septembre 2016

### Objectif

L'objectif de ce TP est d'écrire des expressions rationnelles et de les appliquer sur des fichiers afin d'en extraire du texte et de le manipuler. Nous utiliserons l'outil Perl<sup>1</sup>, installé sur tout bon système d'exploitation.

### Rappels commandes Unix

Voilà un petit rappel des commandes et outils incontournables :

<code>mkdir dir</code>	crée un répertoire nommé <i>dir</i> (MaKe DIRectory)
<code>ls</code>	affiche le contenu du répertoire courant (LiSt)
<code>cd dir</code>	va dans le répertoire pointé par <i>dir</i> (Change Directory)
<code>cd ..</code>	retourne dans le répertoire parent
<code>cd</code>	retourne dans votre <i>HOME</i>
<code>cp src dest</code>	copie un fichier (CoPy)
<code>cp -R src dest</code>	copie un répertoire (CoPy)
<code>mv src dest</code>	déplace/renomme un fichier ou un répertoire (MoVe)
<code>cat file</code>	sort le contenu de <i>file</i> sur la sortie standard

À tout moment, vous pouvez invoquer le manuel en utilisant la commande `man` pour MANual. Par exemple, pour savoir comment utiliser la commande `mkdir`, faites : `'man mkdir'`.

Essayez d'organiser proprement votre espace de stockage pour vous y retrouver au moment de réviser. Créez par exemple un répertoire `tps` (`'mkdir tps'`), allez y (`'cd tps'`), et ajoutez un répertoire `thlr` (`'mkdir thlr'`). Vous pourrez organiser ce répertoire par TP et par exercice. **Pensez à y copier à chaque fois la solution des questions pour pouvoir reprendre le TP plus tard!** Pour cela vous pouvez copier-coller vos expressions vers un fichier texte. (Astuce : la commande `history` affiche la liste des dernières commandes tapées : vous pouvez facilement y trouver les commandes que vous souhaitez recopier.)

Vous pourrez ajouter plus tard d'autres répertoires dans le répertoire `tps` pour les TPs d'autres matières.

### Utiliser les expressions rationnelles en Perl

La syntaxe de Perl pour les expressions rationnelles diffère de celle que nous utilisons en TD. Dans le tableau qui suit  $c_1, c_2, \dots, c_n$  désignent des lettres  $l \in \Sigma$ , et  $L(e)$  est le langage dénoté par l'expression rationnelle  $e$ .

1. <http://www.perl.org>. `man perlretut` affiche un tutoriel sur l'usage des expressions rationnelles dans Perl. Pour sortir du manuel, taper `'q'`.

langage	expression rationnelle Perl
$\Sigma$	.
$\{c\}$	$c$
$\{c_1, c_2\}$	$[c_1c_2]$
$\{c_1, c_2, \dots, c_n\}$	$[c_1-c_n]$
$\Sigma \setminus \{c_1, c_2\}$	$[^{\wedge}c_1c_2]$
$L(e)L(f)$	$ef$
$L(e) \cup L(f)$	$(e f)$
$L(e) \cup \{\varepsilon\}$	$e?$
$L(e)^*$	$e^*$
$L(e)^+$	$e^+$

Notez qu'il est possible d'assembler des classes de caractères entre elles. Par exemple,  $[a-zA-Z]$  représente une lettre minuscule ou majuscule. Enfin, pensez à échapper les caractères spéciaux par un  $\backslash$ . Par exemple, si vous avez besoin de reconnaître le caractère '.' (un point), il faut écrire  $\backslash.$ . Ces caractères spéciaux sont  $\backslash . ? * [ ] | ( )$ .

Pour tester vos expressions rationnelles avec Perl, entrez une commande de la forme suivante dans un terminal :

```
perl -0777 -pe 's/expression-rationnelle/texte-à-substituer/gsm' fichier
```

*texte-à-substituer* permet de spécifier par quoi remplacer le texte reconnu. Par exemple, si l'on applique `'perl -0777 -pe 's/[a-z]/@/gsm'` sur un fichier contenant `'azertyAZERTYyuiop'`, on obtient comme affichage `'@@@@@AZERTY@@@@'` : chaque lettre minuscule reconnue a été remplacé par '@'. Le fichier original n'a pas été modifié par cette opération : le résultat de la substitution est uniquement envoyé à l'écran.

### Fichiers utiles pour réaliser le TP

Pour chaque question nous vous fournissons des fichiers sur lesquels vous appliquerez votre expression rationnelle. Ces fichiers sont contenus dans les répertoires `'TP1_files/exo1'` à `'TP1_files/exo4'` et ont comme nom le numéro de la question associée, précédé par 'q'. Ainsi vous devrez tester votre réponse à la question 2 de l'exercice 1 sur le fichier `'TP1_files/exo2/q2'`. Cependant rien ne vous empêche d'effectuer des tests sur vos propres textes ! Au sein d'un même exercice, pensez à vérifier que votre expression rationnelle fonctionne avec les questions précédentes.

Pour recopier tous ces fichiers dans votre répertoire personnel, entrez dans un terminal :

```
wget http://www.lrde.epita.fr/~akim/thlr/TP1_files.tar.bz2
tar -xjvf TP1_files.tar.bz2
```

### Exercice 1 – Commentaires du langage Pascal

On cherche à reconnaître les commentaires en Pascal qui sont de la forme  $\{ \text{texte} \}$ , où *texte* peut être composé de n'importe quelle suite de caractères.

1. Dans un premier temps, on suppose que l'on a un seul commentaire par fichier. Utilisez Perl pour remplacer les commentaires reconnus par le texte COMMENTAIRE.
2. Si maintenant, on a plusieurs commentaires dans un fichier, que se passe-t-il ? Écrivez une nouvelle expression permettant de pallier ce problème, toujours en remplaçant les commentaires reconnus par COMMENTAIRE.

## Exercice 2 – Chaînes de caractères du langage C

On cherche cette fois à trouver les chaînes de caractères du langage C. Celles-ci sont comprises entre guillemets et contiennent un nombre indéfini de caractères : "texte". `texte` est encore une suite de n'importe quel caractère.

1. Écrivez une expression rationnelle permettant de reconnaître de telles chaînes (il peut y en avoir plusieurs dans le même fichier) : elle devra remplacer les motifs reconnus par le texte CHAINE.
2. Il est possible dans ces chaînes « d'échapper » (ou « déspecialiser ») les caractères spéciaux en les faisant précéder du caractère \. Par exemple, si on veut pouvoir afficher le caractère ", on doit écrire "Texte \" suite du texte" (sinon il aurait été reconnu comme le guillemet de fin de chaîne). En fait, tous les caractères peuvent être échappés de cette manière. Ajoutez cette possibilité à votre précédente expression rationnelle.

## Exercice 3 – Modifications d'identifiants en série

Il est possible de faire référence aux motifs reconnus en utilisant \$1, \$2, etc. Ces variables correspondent aux groupes de parenthèses placées dans une expression rationnelle. \$n référence la n<sup>e</sup> paire de parenthèses. Ainsi, `perl -0777 -pe 's/([a-z])/$1@/gsm'` appliqué sur le texte 'azertyAZERTY' produit 'a@z@e@r@t@y@AZERTY'. On a placé '@' derrière chaque lettre minuscule reconnue.

Ce rappel du motif attrapé au sein du texte de substitution est très utile lorsqu'on a besoin d'appliquer des modifications en série sur du texte. Par exemple, on peut vouloir passer en minuscule la première lettre d'un identifiant.

1. On veut ici modifier les identifiants de fonctions qui ont été écrites sous la forme `get_Identifiant()` ou `set_Identifiant(arguments)`, où `Identifiant` est une suite quelconque de caractères alphanumériques. Le but ici est de supprimer le caractère '\_' au sein de ces identifiants. On repère qu'il s'agit d'une fonction puisque le nom est suivi d'un texte entre parenthèses et qu'elle est précédée par un espace. Le nom de la fonction est uniquement composé de caractères alphanumériques, ainsi que les arguments.
2. On veut maintenant modifier tous les appels de fonctions commençant par une majuscule, en remplaçant cette majuscule par la lettre minuscule correspondante. Par exemple, `Apply()` se trouve modifié en `apply()` et `Process(int i)` devient `process(int i)`.

Les caractères sont toujours des caractères alphanumériques.

Pour modifier la casse d'un caractère, il faut mettre \l devant ce caractère<sup>2</sup>, dans le motif de substitution.

## Exercice 4 – Commentaires imbriqués

1. On souhaite reconnaître les chaînes de l'exercice 1, mais cette fois-ci, elles doivent pouvoir être imbriquées : {Texte {Texte imbriqué}}. Dans cas, on veut donc voir affiché @COMMENTAIRE@COMMENTAIRE@@

---

2. 'man perlre'.



## TP 2 Vcsn– 1

Version du 26 septembre 2016

Dans le cadre de ce TP, nous allons utiliser une partie de Vcsn<sup>1</sup>. Ce projet est une plate-forme de manipulation d'automates développée au LRDE en collaboration avec Télécom ParisTech et le Laboratoire Bordelais d'Informatique (LaBRI). Une interface avec IPython utilisant cette bibliothèque est dédiée à l'interaction avec des automates, des expressions rationnelles, etc.

La documentation de Vcsn est disponible en ligne : <http://vcsn-sandbox.lrde.epita.fr/notebooks/Doc/!Read-me-first.ipynb>.

Vcsn travaille sur des automates bien plus généraux que ceux de ce cours. Le concept de typage d'automate (et plus particulièrement des transitions d'un automate) se nomme « contexte » dans Vcsn. Nous utiliserons le contexte le plus simple : celui des NFAs sur l'alphabet  $\{a, b, \dots, z\}$ , `'lal_char(a-z)`, `'b'`. Lorsque le support des  $\varepsilon$ -NFAs est nécessaire, Vcsn utilise alors le contexte `'lan_char(a-z)`, `'b'`.

Nous utiliserons la commande `'vcsn notebook'` qui ouvre une session interactive d'utilisation de Vcsn sous IPython. Dans cet environnement `ENTER` insère un saut de ligne, et `SHIFT-ENTER` lance l'évaluation de la cellule courante. Une fois la session interactive lancée, exécuter `'import vcsn'`.

Attention à la typographie : `automaton` doit être tapé littéralement, alors que *automaton* est une « méta-variable » qui désigne une expression (e.g., une variable) qui s'évalue en un automate.

### Exercice 1

1. La méthode `'vcsn.context(ctx)'` construit un contexte à partir de sa spécification `ctx`, une chaîne de caractères. Définissez une variable `'b'` qui corresponde aux NFAs sur l'alphabet  $\{a, b, c\}$ , et affichez sa valeur.
2. La méthode `'context.expression(re)'` permet de construire l'objet `expression` (expression rationnelle) à partir de la syntaxe algébrique usuelle :
  - `'\z'` : le langage vide,  $\emptyset$
  - `'\e'` : le mot vide,  $\varepsilon$
  - `'a'` : le langage du mot `'a'`
  - `'e+f'` : `e` ou `f`
  - `'ef'` : `e` suivi de `f`
  - `'e*'` : `e` répété  $n \geq 0$  fois
  - `'(e)'` : groupement

Les priorités habituelles sont appliquées. Il existe quelques sucres syntaxiques :

- `'[a-dmx-z]'` : `'a+b+c+d+m+x+y+z'`
- `'[^a-dmx-z]'` : `'[efghijklmnopqrstuvwxyz]'` si l'alphabet est  $\{a, \dots, z\}$
- `'[^]'` : `'[abcdefghijklmnopqrstuvwxyz]'` si l'alphabet est  $\{a, \dots, z\}$
- `'e{+}'` : `e` répété  $n \geq 1$  fois
- `'e?'` : `e` optionnel
- `'e{3,5}'` : `e` entre 3 et 5 fois
- `'e{3,}'` : `e` au moins 3 fois
- `'e{,5}'` : `e` au plus 5 fois

Saisissez les expressions rationnelles suivantes exprimées avec la syntaxe de Perl (il faut donc les réécrire pour Vcsn). On supposera que l'alphabet est  $\{a, b, c\}$ .

---

1. <http://vcsn.lrde.epita.fr>

1. 'ab\*'
2. 'ab+'
3. 'ab\*|ab+'
4. 'abc|(bac)\*|(cab)+'
5. [a-c]\*[^a]
6. 'a?bc|ab?c'

3. La méthode `'expression.thompson'` permet de générer l'automate correspondant à une expression rationnelle à l'aide de l'algorithme de Thompson. Bien entendu elle produit un  $\epsilon$ -NFA, et par conséquent, lorsque nécessaire, utilise un contexte différent pour l'automate.

Générez l'automate de Thompson des expressions rationnelles de la question précédente.

4. La méthode `'automaton.proper(direction="backward", prune=True)'` effectue l'élimination des transitions spontanées. Si l'argument `prune` est vrai (sa valeur par défaut), alors les états qui deviennent inaccessibles à cette occasion seront éliminés.

Éliminez les transitions spontanées des automates précédents, *sans* éliminer les états devenant inaccessibles.

5. Vous pouvez émonder un automate (ce n'est pas sale) avec la méthode `'automaton.trim'`.

Éliminez les états inutiles des automates précédents.

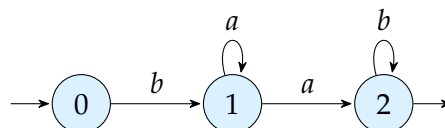
6. La méthode `'automaton.format(format)'` permet de sérialiser un automate dans un certain format, comme `"daut"`.

Lorsque Python affiche une chaîne de caractères, il n'interprète pas les retours à la ligne et affiche un résultat particulièrement illisible. N'hésitez pas à vous servir de `'print(expression)'`.

Convertir un des automates précédents, et en comprendre la structure.

7. La commande `'%%automaton'` permet de *désérialiser* un automate, i.e., de construire un automate à partir d'une description textuelle :

```
%%automaton a
context = "lal_char(abc), b"
$ -> 0
0 -> 0 [a-c]
0 -> 1 a, c
1 -> $
```



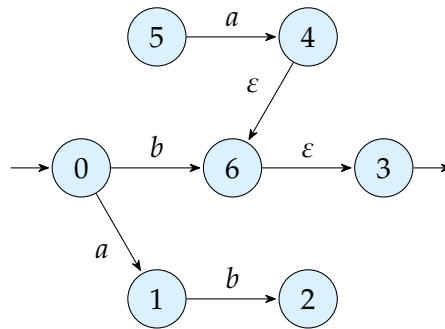
Automate 1: Un automate à saisir

Créez l'**automate 1**.

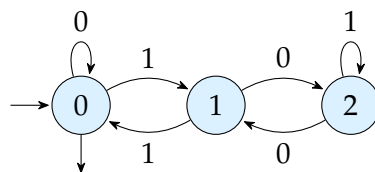
8. Émondez l'**automate 2**. Vérifiez que le résultat correspond bien à celui que vous attendez (c'est-à-dire l'automate privé des états qui ne sont pas co-accessibles ou pas accessibles).

Essayez aussi les méthodes `'automaton.accessible'` (pour ne garder que les états accessibles) et `'automaton.coaccessible'` (pour ne garder que les états co-accessibles).

9. Construisez un automate reconnaissant les nombres *décimaux* divisibles par 3. Vous considérerez que le mot vide (`'\e'`) est équivalent au nombre 0 et fait donc partie du langage reconnu par l'automate.



Automate 2: Un automate à émonder.



Automate 3: div3base2, un automate qui reconnaît les binaires divisibles par 3.

N’hésitez pas à vous inspirer de l’[automate 3](#) qui fait la même chose sur les nombres *binaires*.

La méthode `'automaton.shortest(len=length)'` énumère tous les mots de taille  $\leq length$  reconnus par un automate.

Vérifiez les mots de moins de trois lettres reconnus par votre automate.

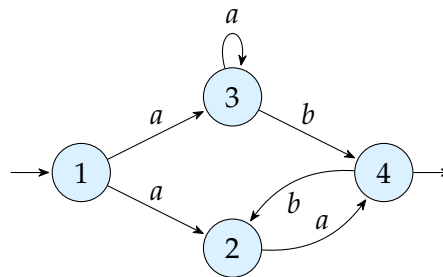
## TP 3 Vcsn– 2

Version du 26 septembre 2016

Dans ce TP, nous allons utiliser à nouveau Vcsn. Si nécessaire, reprenez les instructions du TP 2.

### Exercice 1 – Évaluation sur automate déterministe

1. Tout automate admet un automate déterministe équivalent. La méthode `'automaton.determinize'` permet d'en calculer un.



Automate 1: Automate à déterminer

Déterminez l'[automate 1](#) et affichez le résultat cette opération. Vous pouvez aussi vérifier ceux des TDs!

2. Il est possible d'utiliser Vcsn pour évaluer un mot sur un automate à l'aide de `'automaton(word)'`, ou encore `'automaton.eval'`. Réutilisez les automates créés à partir d'expressions rationnelles du TP précédent pour évaluer des mots, qui selon vous, sont reconnus, ou pas, par ces automates.
3. La notation pointée (`obj.fun()`) offre l'intérêt de pouvoir chaîner lisiblement plusieurs opérations à la suite, dans l'ordre de leur évaluation, de la gauche vers la droite. Par exemple :

```
vcsn.context('lal_char(abc), b').expression('ab*').thompson().proper()
```

crée l'automate de Thompson de l'expression 'ab\*', puis en élimine les transitions spontanées (et les états devenus inutiles).

Ajoutez à l'exemple précédent une étape de déterminisation, ainsi qu'une étape de minimisation (concept qui sera abordé plus tard en cours, mais qui consiste en la recherche de l'*unique* automate déterministe au plus petit nombre d'états).

Ensuite, chaînez des commandes que vous connaissez déjà de manière à générer l'automate correspondant à l'expression rationnelle de votre choix. (Vous pouvez écrire une fonction si vous savez.)

Sur cet automate, vous appliquerez les étapes suivantes :

1. Élimination des transitions spontanées
2. Déterminisation
3. Évaluation d'un mot

### Exercice 2 – Intersection de langages rationnels

Nous verrons demain une construction entre deux automates, le *produit synchronisé*. Elle produit un automate acceptant l'intersection des langages des deux automates.

1. Construire un automate qui accepte les mots sur  $\{a, b\}$  ayant un nombre impair de  $b$ . Vous pouvez utiliser la méthode `'expression.automaton'` pour convertir une expression rationnelle en NFA. Vérifier votre automate grâce à `'automaton.shortest'`.
2. L'opération `'automaton & automaton'` (ou encore `'automaton.conjunction'`) calcule le produit synchronisé de deux automates. Calculez un automate qui accepte les mots ayant un nombre impair de  $a$  et un nombre impair de  $b$ . Le vérifier grâce à `'shortest'`.
3. En déduire une expression rationnelle qui dénote l'ensemble des mots sur  $\{a, b\}$  ayant un nombre impair de  $a$ , et un nombre impair de  $b$ .
4. Vcsn supporte les expressions rationnelles *étendues* qui incluent deux opérateurs supplémentaires : `'e&f'` pour la conjonction (intersection) et `'e{c}'` pour la complémentation. Reprenez la question précédente, et traitez-la directement sur les expressions rationnelles, grâce à `'expression & expression'` (ou encore `'expression.conjunction'`).

### Exercice 3 – Le problème de Mans Hulden

Durant la conférence FSMNLP'08, Mans Hulden a mentionné un petit problème de langage dont une solution utilise une composition complexe de langages rationnels : étant donné un langage rationnel  $L$ , construire le langage  $L'$  des mots de  $\Sigma^*$  qui contiennent exactement un seul facteur dans  $L$ . Il se trouve que  $L'$  est rationnel.

Le propos de cet exercice est de traiter ce problème. Nous pourrions représenter les langages sur la forme d'automates et les transformer (concaténation, conjonction d'automates, etc.). Puisque Vcsn supporte les expressions étendues, nous le ferons avec des expressions, ce qui donnera des résultats équivalents, mais plus faciles à lire.

1. Définissez la fonction suivante, analysez-la, et essayez-la sur l'expression  $ab + ba$ .

Pensez à `'expression.shortest'`.

```
ctx = vcsn.context('lal_char(abc), b')
```

```
# A helper function to define expressions on 'lal_char(abc), b'.
```

```
def exp(e):
```

```
    # If 'e' is not a real expression object, convert it.
```

```
    return e if isinstance(e, vcsn.expression) else ctx.expression(e)
```

```
def t1(re):
```

```
    re = exp(re)
```

```
    all = exp('[^]*')
```

```
    # * denotes the concatenation.
```

```
    return all * re * all
```

```
t1('a')
```

2. Soit  $e$  une expression rationnelle. Proposez une expression rationnelle qui désigne tous les mots qui contiennent au moins deux facteurs disjoints (sans recouvrement) qui sont des mots engendrés par  $e$ .
3. Définir une fonction Python qui construise cette expression rationnelle.
4. Vérifiez, à l'aide de `'shortest'`, que votre réponse est (probablement) correcte pour  $e = ab + ba$ .
5. Proposez une expression rationnelle qui désigne tous les mots qui ont un préfixe dans  $e$  et un suffixe propre dans  $e$ . Pensez à l'opérateur `'&'`.

6. Adaptez votre fonction pour engendrer cette expression rationnelle, et vérifiez-la pour  $e = ab + ba$ .
7. Adaptez votre fonction pour qu'elle reconnaisse tout mot qui contienne deux facteurs (au moins) de  $e$ , qui se recouvrent potentiellement.
8. L'opération '*expression % expression*' (ou encore '*expression.difference*') engendre une expression rationnelle qui désigne les mots acceptés par la première expression mais pas la seconde. Adaptez votre fonction pour qu'elle reconnaisse tout mot qui contienne exactement un seul facteur de  $e$ . Essayez-la.
9. Vérifiez la correction de votre réponse en prenant  $L = \{ab, aba\}$ . Qu'observez-vous? Comment l'expliquer?
10. Étant donné un langage rationnel  $L$ , caractérisez le langage des mots de  $L$  qui ont un (ou plusieurs) facteur propre dans  $L$ . Implémentez une fonction '*composi te*' qui réalise ce calcul pour des expressions rationnelles, et testez-la sur  $L = \{ab, aba\}$ .
11. Proposer finalement une fonction qui prenne en argument une expression rationnelle  $e$  et retourne une expression rationnelle reconnaissant tous les mots qui contiennent exactement un unique facteur dans  $L(e)$ .

## Chapitre 8

# Annales

Ces sujets ont été écrits par Akim Demaille.



## Partiel Théorie des Langages Rationnels

Version du 26 septembre 2016

### Exercice 1 – Questions à choix multiples

Bien lire le sujet, chaque mot est important. Répondre sur les formulaires de QCM, aucune réponse manuscrite ne sera corrigée. Renseigner les champs d'identité.

Il y a exactement une et une seule réponse juste par question. Si plusieurs réponses sont valides, sélectionner la plus restrictive. Par exemple s'il est demandé si 0 est *nul*, *non nul*, *positif*, ou *négatif*, sélectionner *nul* qui est plus restrictif que *positif* et *négatif*, tous deux vrais.

Les réponses justes créditent, les réponses incorrectes pénalisent, et les réponses blanches valent 0; il est plus sûr de ne pas répondre que de laisser le hasard décider.

- Q.1 Le langage  $0^n$  est
- a. fini
  - b. rationnel
  - c. non reconnaissable par automate fini
  - d. vide
- Q.2 Le langage  $0^n 1^n$  pour  $n < 42^{51} - 1$  est
- a. infini
  - b. rationnel
  - c. non reconnaissable par automate fini
  - d. vide
- Q.3 Le langage  $0^n 1^n$  est
- a. fini
  - b. rationnel
  - c. non reconnaissable par automate fini
  - d. vide
- Q.4 L'ensemble de tous les prénoms de la promotion est un langage
- a. rationnel
  - b. non reconnaissable par un automate fini déterministe
  - c. non reconnaissable par un automate fini nondéterministe
  - d. non reconnaissable par un automate fini à transitions spontanées
- Q.5 Un langage quelconque
- a. est toujours inclus ( $\subset$ ) dans un langage rationnel
  - b. est toujours récursif
  - c. peut avoir une intersection non vide avec son complémentaire
  - d. peut ne pas être inclus dans un langage défini par une expression rationnelle
- Q.6 Quelle est l'écriture la plus raisonnable ?
- a. machine à état fini
  - b. machine à état finis
  - c. machine à états finie
  - d. machine à états finis
- Q.7 Un automate fini déterministe. . .
- a. n'est pas nondéterministe
  - b. n'est pas à transitions spontanées
  - c. n'a pas plusieurs états initiaux
  - d. n'a pas plusieurs états finaux
- Q.8 Un algorithme peut décider si un automate est déterministe en regardant sa structure.
- a. Faux
  - b. Rarement
  - c. Souvent
  - d. Vrai
- Q.9 L'expression rationnelle étendue  $[-+]?[0-9]^+ + ([0-9]^+)?(e[-+]?[0-9]^+)$  n'engendre pas :
- a.  $42e42$
  - b.  $42,e42$
  - c.  $42,4e42$
  - d.  $42,42e42$



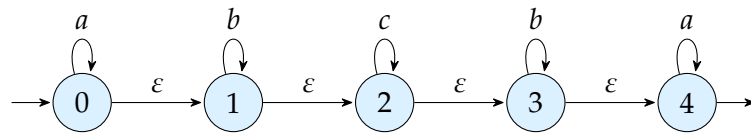
Q.10 L'automate de Thompson de l'expression rationnelle  $(ab)^*c$

- a. ne contient pas de boucle
- b. n'a aucune transition spontanée
- c. a 8, 10, ou 12 états
- d. est déterministe

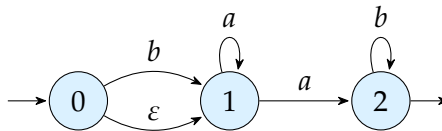
**Exercice 2 – Constructions sur les automates**

Dans cet exercice, on suppose que  $\Sigma = \{a, b, c\}$ .

1. (**Algorithme de Thompson**) Construire l'automate de Thompson de  $b(ab + b)$ .
2. (**Élimination des transitions spontanées**) Appliquer l'élimination (arrière) des transitions spontanées dans l'automate suivant :



3. (**Déterminisation**) Déterminiser rigoureusement l'automate suivant :



4. (**Minimisation**) Construire rigoureusement l'automate minimal reconnaissant le langage engendré par  $ab + ac$ .

## Partiel Théorie des Langages Rationnels

### Aucun document ni appareil autorisé

Version du 26 septembre 2016

Bien lire le sujet, chaque mot est important. Répondre sur les formulaires de QCM, aucune réponse manuscrite ne sera corrigée. Renseigner les champs d'identité.

Il y a exactement une et une seule réponse juste par question. Si plusieurs réponses sont valides, sélectionner la plus restrictive. Par exemple s'il est demandé si 0 est *nul*, *non nul*, *positif*, ou *négatif*, sélectionner *nul* qui est plus restrictif que *positif* et *négatif*, tous deux vrais.

Les réponses justes créditent, les réponses incorrectes pénalisent, et les réponses blanches valent 0; il est plus sûr de ne pas répondre que de laisser le hasard décider.

Q.1 Le langage  $\{\heartsuit^n \mid \forall n \in \mathbb{N}\}$  est

- |              |   |
|--------------|---|
| a. fini      | c. non reconnaissable par automate fini |
| b. rationnel | d. vide                                 |

Q.2 Le langage  $\{\sigma^n \heartsuit^n \mid \forall n \in \mathbb{N} : n < 242^{51} - 1\}$  est

- |              |   |
|--------------|---|
| a. infini    | c. non reconnaissable par automate fini |
| b. rationnel | d. vide                                 |

Q.3 Le langage  $\{\heartsuit^n \heartsuit^n \mid \forall n \in \mathbb{N}\}$  est

- |              |   |
|--------------|---|
| a. fini      | c. non reconnaissable par automate fini |
| b. rationnel | d. vide                                 |

Q.4 L'ensemble des mots du petit Robert (édition 1975) est

- a. rationnel
- b. non reconnaissable par un automate fini déterministe
- c. non reconnaissable par un automate fini nondéterministe
- d. ne peut être représenté par une expression rationnelle

Q.5 Un langage quelconque

- a. n'est pas nécessairement dénombrable (i.e., il n'existe pas toujours de bijection entre ses mots et une partie de  $\mathbb{N}$ )
- b. est toujours inclus ( $\subset$ ) dans un langage rationnel
- c. peut n'être inclus dans aucun langage dénoté par une expression rationnelle
- d. peut avoir une intersection non vide avec son complémentaire

Q.6 Un automate fini qui a plusieurs états initiaux. . .

- |                              |                                       |
|------------------------------|---------------------------------------|
| a. n'est pas déterministe    | c. n'est pas à transitions spontanées |
| b. n'est pas nondéterministe | d. n'a pas plusieurs états finaux     |

Q.7 En soumettant à un automate un nombre fini de mots de notre choix et en observant ses réponses, mais sans en regarder la structure (test boîte noire), on peut savoir. . .

- |                                      |                                   |
|--------------------------------------|-----------------------------------|
| a. s'il est déterministe             | c. s'il accepte le mot vide       |
| b. s'il a des transitions spontanées | d. s'il accepte un langage infini |

Q.8 L'expression rationnelle étendue  $[-+]?[0-9A-F] + ([-+/*][0-9A-F]+)^*$  n'engendre pas :

- |          |              |                     |                |
|----------|--------------|---------------------|----------------|
| a. $-42$ | b. $42 + 42$ | c. $42 + (42 * 42)$ | d. $-42 - -42$ |
|----------|--------------|---------------------|----------------|

Q.9 Si  $e$  et  $f$  sont deux expressions rationnelles, quelle identité n'est pas nécessairement vérifiée ?

- |                                |                         |                         |                              |
|--------------------------------|-------------------------|-------------------------|------------------------------|
| a. $\emptyset^* = \varepsilon$ | b. $(ef)^* e = e(fe)^*$ | c. $(ef)^* = e(fe)^* f$ | d. $(e + f)^* = (e^* f^*)^*$ |
|--------------------------------|-------------------------|-------------------------|------------------------------|

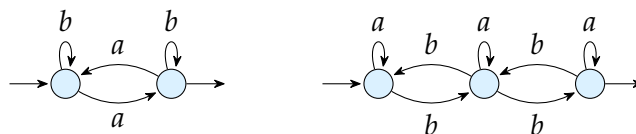
Q.10 Si un automate de  $n$  états accepte  $a^n$ , alors il reconnaît. . .

- a.  $(a^n)^*$
- b.  $a^{n+1}$
- c.  $a^n a^*$
- d.  $a^p (a^q)^*$  avec  $p \in \mathbb{N}, q \in \mathbb{N}^* : p + q \leq n$

Q.11 Quelle séquence d’algorithmes teste l’appartenance d’un mot au langage représenté par une expression rationnelle ?

- a. Thompson, élimination des transitions spontanées, déterminisation, évaluation.
- b. Thompson, minimisation, déterminisation, évaluation.
- c. Thompson, déterminisation, élimination arrière puis avant des transitions spontanées, évaluation.
- d. Thompson, déterminisation, élimination des  $\varepsilon$ -transitions, évaluation.

Q.12 Quel mot est reconnu par l’automate produit des deux automates suivants ?



- a.  $(bab)^{22}$
- b.  $(bab)^{333}$
- c.  $(bab)^{4444}$
- d.  $(bab)^{666666}$

Q.13 Combien d’états a l’automate de Thompson de l’expression rationnelle  $(a + b)^* + (b + a)^*$  :

- a. 9
- b. 13
- c. 18
- d. 26

Q.14 Combien d’états au moins a un automate déterministe émondé qui accepte les mots dont la  $n$ -ième lettre avant la fin est un  $a$  (i.e.,  $(a + b)^* a (a + b)^{n-1}$ ) :

- a.  $\frac{n(n+1)}{2}$
- b.  $n + 1$
- c.  $2^n$
- d. Il n’existe pas.

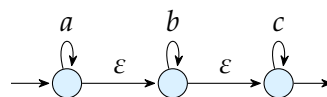
Q.15 Combien d’états a l’automate minimal qui accepte le langage  $\{a, ab, abc\}$  ?

- a. Il n’existe pas.
- b. 4
- c. 6
- d. 7

Q.16 Si  $L$  et  $L'$  sont rationnels, quel langage ne l’est pas nécessairement ?

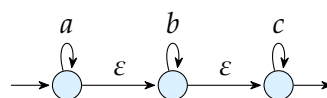
- a.  $\{u \in \Sigma^* \mid u \in L \wedge u \in L'\}$
- b.  $\{u^n v^n \mid u \in L, v \in L', n \in \mathbb{N}\}$
- c.  $\{u \in \Sigma^* \mid u \in L \wedge u \notin L'\}$
- d.  $\{u \in \Sigma^* \mid u \in L\}$

Q.17 Quel est le résultat d’une élimination *arrière* des transitions spontanées sur l’automate suivant ?



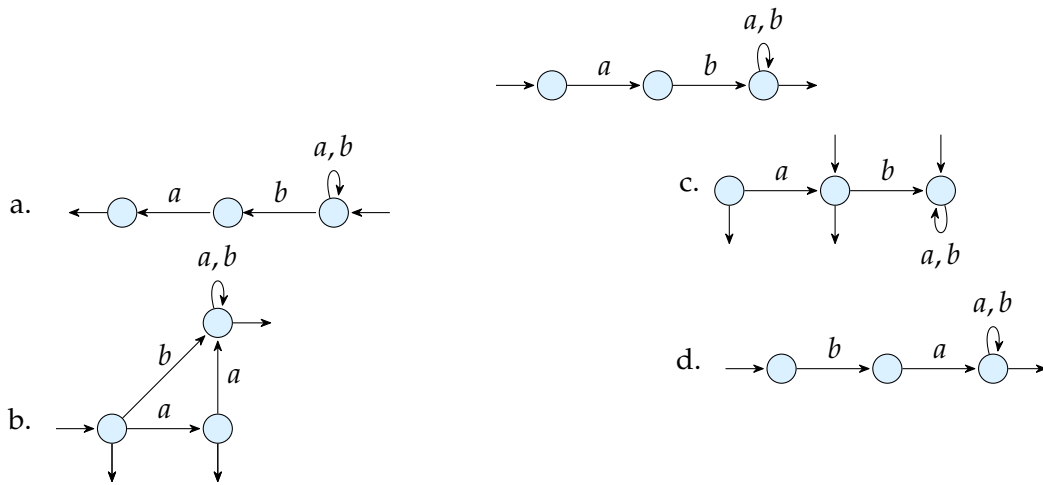
- a.
- b.
- c.
- d.

Q.18 L’automate suivant est. . .

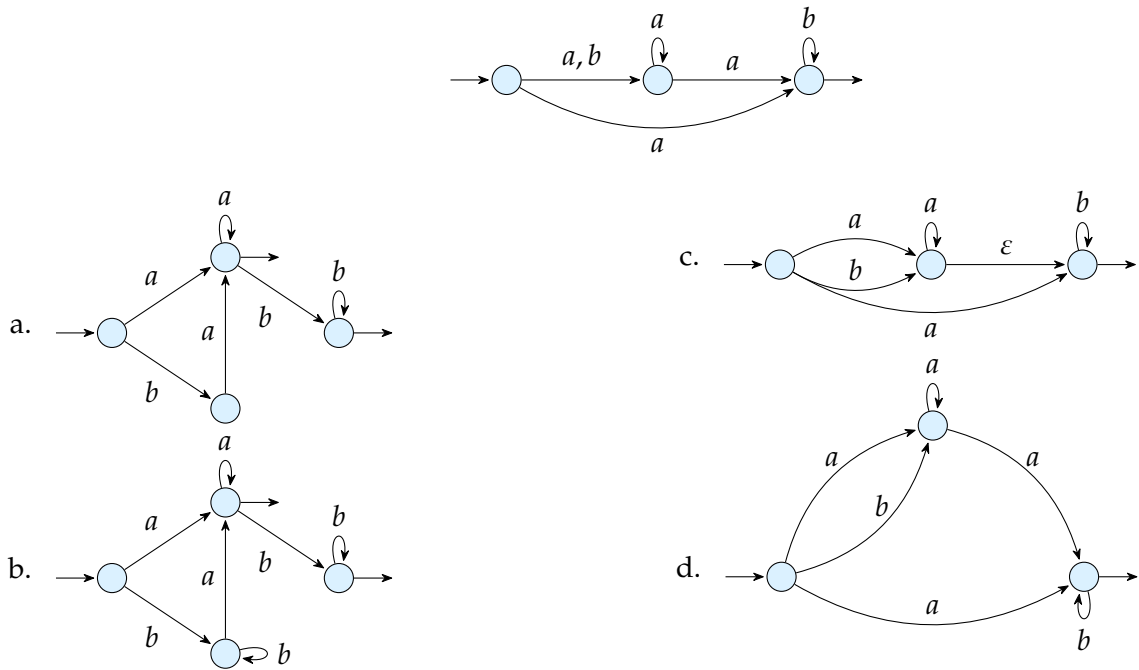


- a. nondéterministe à transitions spontanées
- b. déterministe à transitions spontanées
- c.  $\varepsilon$ -déterministe
- d.  $\varepsilon$ -minimal

Q.19 Quel automate reconnaît le langage complémentaire de celui accepté par l'automate suivant ?



Q.20 Déterminer l'automate suivant.



## Partiel Théorie des Langages Rationnels

### Aucun document ni appareil autorisé

Version du 26 septembre 2016

Bien lire le sujet, chaque mot est important. Répondre sur les formulaires de QCM, aucune réponse manuscrite ne sera corrigée. Renseigner les champs d'identité.

Il y a exactement une et une seule réponse juste par question. Si plusieurs réponses sont valides, sélectionner la plus restrictive. Par exemple s'il est demandé si 0 est *nul*, *non nul*, *positif*, ou *négatif*, sélectionner *nul* qui est plus restrictif que *positif* et *négatif*, tous deux vrais.

Les réponses justes créditent, les réponses incorrectes pénalisent, et les réponses blanches valent 0; il est plus sûr de ne pas répondre que de laisser le hasard décider.

Q.1 Le langage  $\{\ominus^n \mid \forall n \in \mathbb{N}\}$  est

- |              |   |
|--------------|---|
| a. fini      | c. non reconnaissable par automate fini |
| b. rationnel | d. vide                                 |

Q.2 Le langage  $\{\text{☠}^n \text{☠}^n \mid \forall n \in \mathbb{N}\}$  est

- |              |   |
|--------------|---|
| a. fini      | c. non reconnaissable par automate fini |
| b. rationnel | d. vide                                 |

Q.3 Le langage  $\{\text{Ctrl}^n \text{Alt}^n \text{Del}^n \mid \forall n \in \mathbb{N} : n < 242^{51} - 1\}$  est

- |              |   |
|--------------|---|
| a. fini      | c. non reconnaissable par automate fini |
| b. rationnel | d. vide                                 |

Q.4 Le langage  $\{\text{Bart}^n \text{Bart}^m \mid \forall n, m \in \mathbb{N}\}$  est

- |              |   |
|--------------|---|
| a. fini      | c. non reconnaissable par automate fini |
| b. rationnel | d. vide                                 |

Q.5 Un langage quelconque

- a. n'est pas nécessairement dénombrable (i.e., il n'existe pas toujours de bijection entre ses mots et une partie de  $\mathbb{N}$ )
- b. est toujours inclus ( $\subset$ ) dans un langage rationnel
- c. peut n'être inclus dans aucun langage dénoté par une expression rationnelle
- d. peut avoir une intersection non vide avec son complémentaire

Q.6 Un automate fini qui a plusieurs états initiaux. . .

- |                              |                                       |
|------------------------------|---------------------------------------|
| a. n'est pas déterministe    | c. n'est pas à transitions spontanées |
| b. n'est pas nondéterministe | d. n'a pas plusieurs états finaux     |

Q.7 En soumettant à un automate de taille inconnue un nombre fini de mots de notre choix et en observant ses réponses, mais sans en regarder la structure (test boîte noire), on peut savoir. . .

- |                                      |                                   |
|--------------------------------------|-----------------------------------|
| a. s'il est déterministe             | c. s'il accepte le mot vide       |
| b. s'il a des transitions spontanées | d. s'il accepte un langage infini |

Q.8 L'expression rationnelle étendue  $'([ - + ]^* [ 0 - 9 A - F ] + [ - + / * ])^* [ - + ]^* [ 0 - 9 A - F ] +'$  n'engendre pas :

- |                     |  |
|---------------------|--|
| a. $'-+-1+--+2'$    | c. $'DEADBEEF'$                          |
| b. $'(20 + 3) * 3'$ | d. $'0 + 1 + 2 + 3 + 4 + 5 + 7 + 8 + 9'$ |

Q.9 Si  $e$  et  $f$  sont deux expressions rationnelles, quelle identité n'est pas nécessairement vérifiée?

- |                                |                                    |
|--------------------------------|------------------------------------|
| a. $\emptyset^* = \varepsilon$ | c. $(ef)^* = e(fe)^* f$            |
| b. $(ef)^* e = e(fe)^*$        | d. $(e + f)^* = (f^*(ef)^* e^*)^*$ |

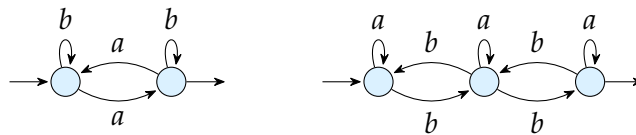
Q.10 Si un automate de  $n$  états accepte  $a^n$ , alors il reconnaît. . .

- a.  $(a^n)^m$  avec  $m \in \mathbb{N}^*$
- b.  $a^{n+1}$
- c.  $a^n a^m$  avec  $m \in \mathbb{N}^*$
- d.  $a^p (a^q)^*$  avec  $p \in \mathbb{N}, q \in \mathbb{N}^* : p + q \leq n$

Q.11 Quelle séquence d’algorithmes teste l’appartenance d’un mot au langage représenté par une expression rationnelle ?

- a. Thompson, élimination des transitions spontanées, déterminisation, évaluation.
- b. Thompson, déterminisation, BMC.
- c. Thompson, déterminisation, élimination avant puis arrière des transitions spontanées, évaluation.
- d. Minimisation, co-bi-minimisation,  $\varepsilon$ -dé-ter-minimisation.

Q.12 Quel mot est reconnu par l’automate produit des deux automates suivants ?



- a.  $(bab)^{22}$
- b.  $(bab)^{333}$
- c.  $(bab)^{4444}$
- d.  $(bab)^{666666}$

Q.13 Combien d’états a l’automate de Thompson de l’expression rationnelle à laquelle je pense ?

- a. 0
- b. 51
- c. 18
- d. 255

Q.14 Combien d’états au moins a un automate déterministe émondé qui accepte les mots sur  $\Sigma = \{a, b, c, d\}$  dont la  $n$ -ième lettre avant la fin est un  $a$  (i.e.,  $(a + b + c + d)^* a (a + b + c + d)^{n-1}$ ) :

- a.  $\frac{n(n+1)(n+2)(n+3)}{4}$
- b.  $2^n$
- c.  $4^n$
- d. Il n’existe pas.

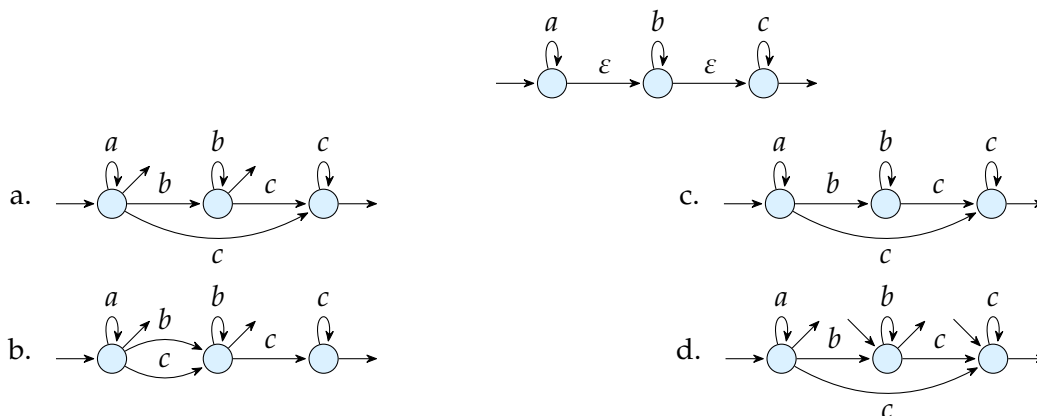
Q.15 Combien d’états a l’automate minimal qui accepte le langage  $(a + b)^+$  ?

- a. Il en existe plusieurs!
- b. 1
- c. 2
- d. 3

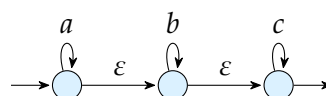
Q.16 Si  $L$  et  $L'$  sont rationnels, quel langage ne l’est pas nécessairement ?

- a.  $\{u \in \Sigma^* \mid u \in L \wedge u \in L'\}$
- b.  $\{u^n v^n \mid u \in L, v \in L', n \in \mathbb{N}\}$
- c.  $\{u \in \Sigma^* \mid u \in L \wedge u \notin L'\}$
- d.  $\{u \in \Sigma^* \mid u \in L\}$

Q.17 Quel est le résultat d’une élimination *arrière* des transitions spontanées sur l’automate suivant ?

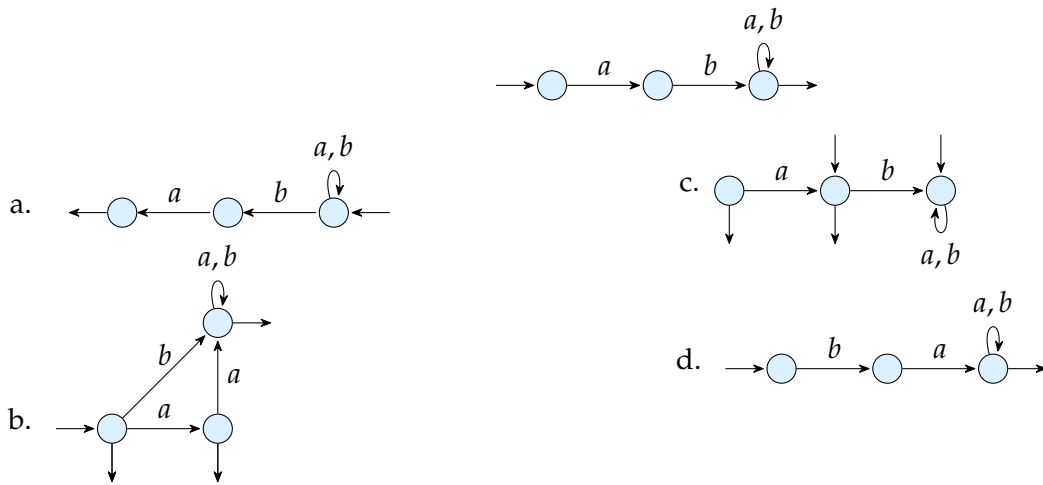


Q.18 L’automate suivant est. . .

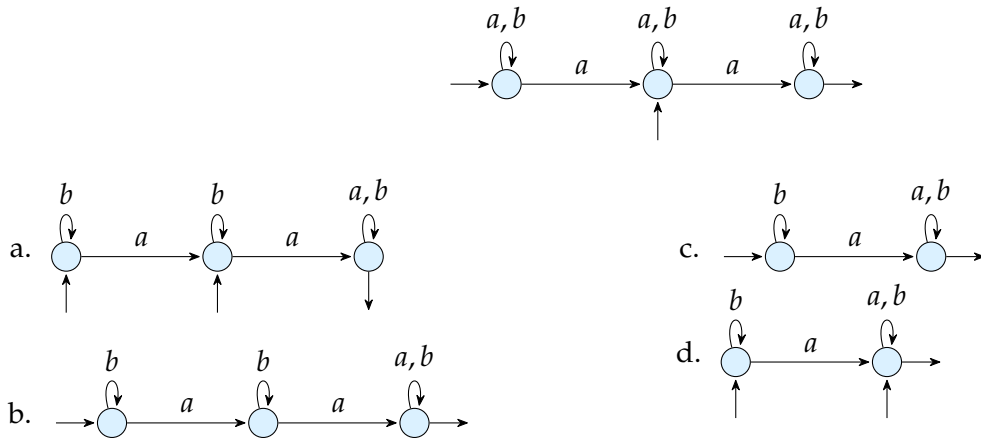


- a. nondéterministe à transitions spontanées
- b. déterministe à transitions spontanées
- c.  $\varepsilon$ -déterministe
- d.  $\varepsilon$ -minimal

Q.19 Quel automate reconnaît le langage complémentaire de celui accepté par l'automate suivant ?



Q.20 Déterminer l'automate suivant.



## Partiel Théorie des Langages Rationnels

### Aucun document ni appareil autorisé

Version du 26 septembre 2016

Bien lire le sujet, chaque mot est important. Répondre sur les formulaires de QCM, aucune réponse manuscrite ne sera corrigée. Renseigner les champs d'identité.

Il y a exactement une et une seule réponse juste par question. Si plusieurs réponses sont valides, sélectionner la plus restrictive. Par exemple s'il est demandé si 0 est *nul*, *non nul*, *positif*, ou *négatif*, sélectionner *nul* qui est plus restrictif que *positif* et *négatif*, tous deux vrais.

Les réponses justes créditent, les réponses incorrectes pénalisent, et les réponses blanches valent 0; il est plus sûr de ne pas répondre que de laisser le hasard décider.

Q.1 Le langage  $\{ \text{stick figure}^{2n} \mid \forall n \in \mathbb{N} \}$  est

- a. fini  
b. rationnel  
c. non reconnaissable par automate fini  
d. vide

Q.2 Le langage  $\{ \text{flower}^n \mid \forall n \in \mathbb{N} \}$  est

- a. fini  
b. rationnel  
c. non reconnaissable par automate fini  
d. vide

Q.3 Le langage  $\{ \text{box}^n \mid \forall n \in \mathbb{N} : 42! \leq n \leq 51! \}$  est

- a. fini  
b. rationnel  
c. non reconnaissable par automate fini  
d. vide

Q.4 Un alphabet est :

- a. un ensemble ordonné  
b. un ensemble fini  
c. un ensemble infini  
d. une suite finie

Q.5 Ces expressions rationnelles :

$$(a^* + b)^* + c((ab)^*(bc))^*(ab)^*$$

$$c(ab + bc)^* + (a + b)^*$$

- a. sont identiques  
b. sont équivalentes  
c. ne sont pas équivalentes  
d. dénotent des langages différents

Q.6 Pour une expression rationnelle composée de  $n$  opérations autres que la concaténation, l'automate de Thompson compte :

- a.  $n$  états  
b.  $2n$  états  
c.  $n^2$  états  
d.  $2^n$  états

Q.7 Un automate déterministe est un automate non-déterministe à transitions spontanées.

- a. toujours vrai  
b. toujours faux  
c. parfois vrai  
d. c'est le contraire

Q.8 Si un langage vérifie le lemme de pompage, alors il est rationnel.

- a. toujours vrai  
b. toujours faux  
c. vrai seulement pour le langage vide  
d. c'est le contraire



Q.9 « Émonder un automate » signifie lui enlever

- a. ses états utiles
- b. ses états inutiles
- c. ses transitions spontanées
- d. ses états inaccessibles

Q.10 Si  $L_1 \subset L \subset L_2$ , alors  $L$  est rationnel si :

- a.  $L_1$  est rationnel
- b.  $L_2$  est rationnel
- c.  $L_1, L_2$  sont rationnels
- d.  $L_1, L_2$  sont rationnels et  $L_2 \subset L_1$

Q.11 Un automate fini qui a des transitions spontanées. . .

- a. est déterministe
- b. n'est pas déterministe
- c. accepte  $\varepsilon$
- d. n'accepte pas  $\varepsilon$

Q.12 Considérons  $\mathcal{P}$  l'ensemble des *palindromes* (mot  $u$  égal à son image miroir  $u^R$ ) de longueur paire sur  $\Sigma$ , i.e.,  $\mathcal{P} = \{v \cdot v^R \mid v \in \Sigma^*\}$ .

- a. Il existe un DFA qui reconnaisse  $\mathcal{P}$
- b. Il existe un NFA qui reconnaisse  $\mathcal{P}$
- c. Il existe un  $\varepsilon$ -NFA qui reconnaisse  $\mathcal{P}$
- d.  $\mathcal{P}$  ne vérifie pas le lemme de pompage

Q.13 Si  $L_1, L_2$  sont rationnels, alors :

- a.  $\bigcup_{n \in \mathbb{N}} L_1^n \cdot L_2^n$  aussi
- b.  $(L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2)$  aussi
- c.  $L_1 \subset L_2$  ou  $L_1 \supset L_2$
- d.  $L_1 \cap L_2 = \overline{L_1} \cap \overline{L_2}$

Q.14 Si  $e$  et  $f$  sont deux expressions rationnelles, quelle identité n'est pas nécessairement vérifiée ?

- a.  $(ef)^* e = e(fe)^*$
- b.  $\emptyset^* = \varepsilon^*$
- c.  $(e + f)^* = (f^* e^* f^* e^*)^*$
- d.  $(ef)^* = e(fe)^* f$

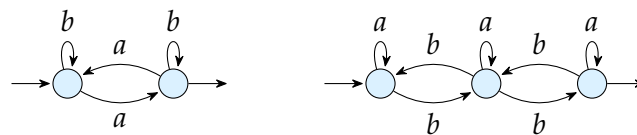
Q.15 Si un automate de  $n$  états accepte  $a^n$ , alors il accepte. . .

- a.  $(a^n)^m$  avec  $m \in \mathbb{N}^*$
- b.  $a^{n+1}$
- c.  $a^n a^m$  avec  $m \in \mathbb{N}^*$
- d.  $a^p (a^q)^*$  avec  $p \in \mathbb{N}, q \in \mathbb{N}^* : p + q \leq n$

Q.16 Quelle séquence d'algorithmes teste l'appartenance d'un mot au langage représenté par une expression rationnelle ?

- a. Thompson, déterminisation, Brzozowski-McCluskey.
- b. Thompson, déterminisation, élimination des transitions spontanées, évaluation.
- c. Thompson, élimination des transitions spontanées, déterminisation, minimisation, évaluation.
- d. Thompson, déterminisation, évaluation.

Q.17 Quel mot est reconnu par l'automate produit des deux automates suivants ?



- a.  $(bab)^{22}$
- b.  $(bab)^{333}$
- c.  $(bab)^{4444}$
- d.  $(bab)^{666666}$

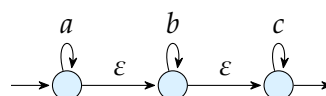
Q.18 Combien d'états n'a pas l'automate de Thompson de l'expression rationnelle à laquelle je pense ?

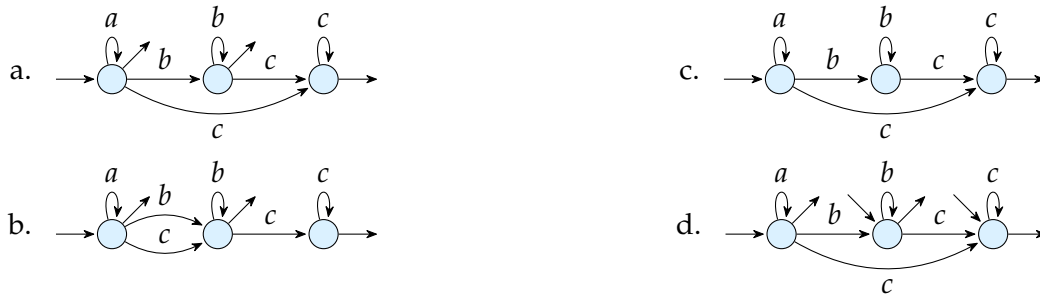
- a. 1248
- b. 2481
- c. 4812
- d. 8124

Q.19 Combien d'états a l'automate déterministe minimal qui accepte le langage  $L$  dénoté par  $(a + b + c)^+$  ?

- a. Il n'existe pas d'automate minimal pour  $L$
- b. 1
- c. 2
- d. 3

Q.20 Quel est le résultat d'une élimination *arrière* des transitions spontanées sur l'automate suivant ?





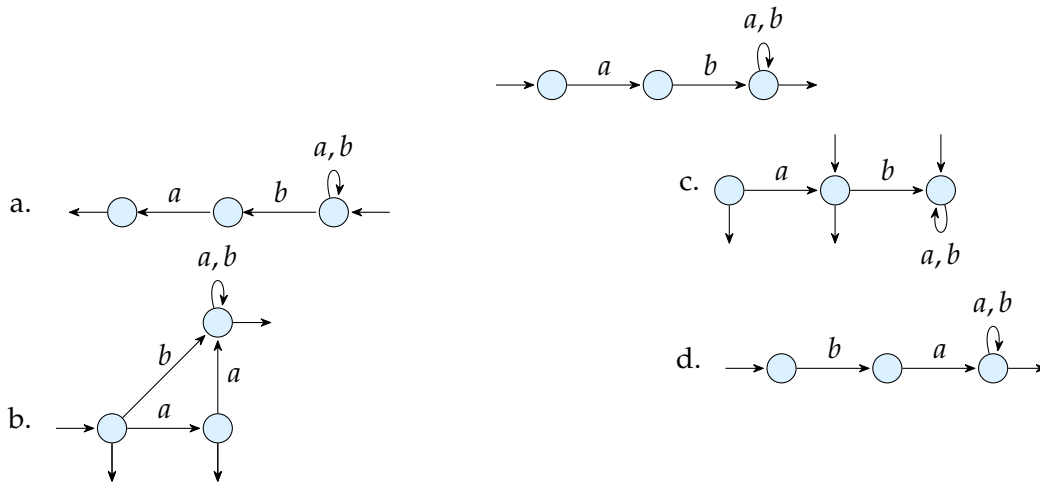
Q.21 Si l'on détermine la réponse de la question 20, puis qu'on le minimise, alors l'application de BMC conduira à une expression rationnelle équivalente à :

- a.  $a^*b^*c^*$
- b.  $a^* + b^* + c^*$
- c.  $(abc)^*$
- d.  $(a + b + c)^*$

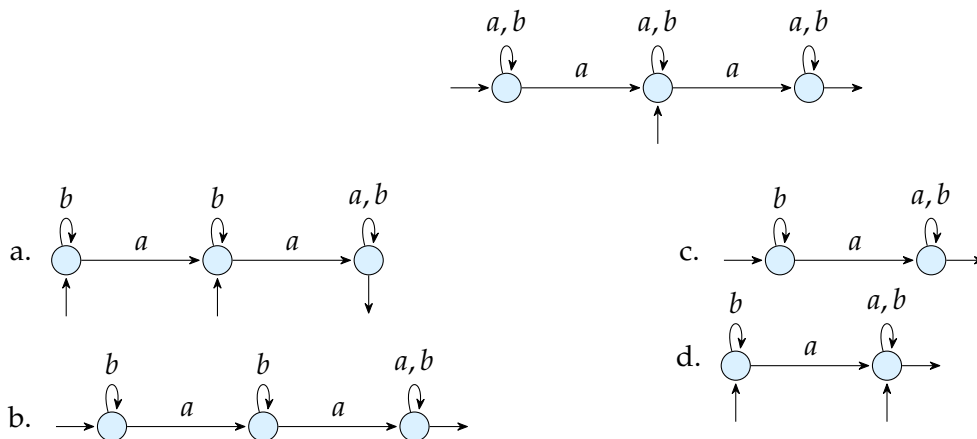
Q.22 L'automate de départ de la question 20 est...

- a. nondéterministe à transitions spontanées
- b. déterministe à transitions spontanées
- c.  $\epsilon$ -déterministe
- d.  $\epsilon$ -minimal

Q.23 Quel automate reconnaît le langage complémentaire de celui accepté par l'automate suivant?



Q.24 Déterminer l'automate suivant.



## Partiel Théorie des Langages Rationnels

### Aucun document ni appareil autorisé

Version du 26 septembre 2016

Bien lire le sujet, chaque mot est important. Répondre sur les formulaires de QCM, aucune réponse manuscrite ne sera corrigée. Renseigner les champs d'identité.

Il y a exactement une et une seule réponse juste par question. Si plusieurs réponses sont valides, sélectionner la plus restrictive. Par exemple s'il est demandé si 0 est *nul*, *non nul*, *positif*, ou *négatif*, sélectionner *nul* qui est plus restrictif que *positif* et *négatif*, tous deux vrais.

Les réponses justes créditent, les réponses incorrectes pénalisent, et les réponses blanches valent 0; il est plus sûr de ne pas répondre que de laisser le hasard décider.

- Q.1 Le langage  $\{ \text{♞}^n \text{♞}^n \mid \forall n \in \mathbb{N} \}$  est
- a. fini  
b. rationnel  
c. non reconnaissable par automate fini  
d. vide
- Q.2 Le langage  $\{ \text{♚}^n \text{♚}^n \mid \forall n \in \mathbb{N} \}$  est
- a. fini  
b. rationnel  
c. non reconnaissable par automate fini  
d. vide
- Q.3 Le langage  $\{ \text{♚}^n \text{♚}^n \text{♚}^n \mid \forall n \text{ premier, codable en binaire sur 64 bits} \}$  est
- a. fini  
b. rationnel  
c. non reconnaissable par automate fini  
d. vide
- Q.4 Pour  $e = (ab)^*$ ,  $f = a^*b^*$  :
- a.  $L(e) \subseteq L(f)$   
b.  $L(e) \supseteq L(f)$   
c.  $L(e) = L(f)$   
d.  $L(e) \not\subseteq L(f)$
- Q.5 Pour  $e = (ab)^*$ ,  $f = (a + b)^*$  :
- a.  $L(e) \subseteq L(f)$   
b.  $L(e) \supseteq L(f)$   
c.  $L(e) = L(f)$   
d.  $L(e) \not\subseteq L(f)$
- Q.6 Pour  $e = (a + b)^*$ ,  $f = a^*b^*$  :
- a.  $L(e) \subseteq L(f)$   
b.  $L(e) \supseteq L(f)$   
c.  $L(e) = L(f)$   
d.  $L(e) \not\subseteq L(f)$
- Q.7 Pour  $e = (a + b)^* + \varepsilon$ ,  $f = (a^*b^*)^*$  :
- a.  $L(e) \subseteq L(f)$   
b.  $L(e) \supseteq L(f)$   
c.  $L(e) = L(f)$   
d.  $L(e) \not\subseteq L(f)$
- Q.8 Pour une expression rationnelle composée de  $n$  opérations autres que la concaténation, l'automate de Thompson compte :
- a.  $n$  états  
b.  $2n$  états  
c.  $n^2$  états  
d.  $2^n$  états
- Q.9 Un automate déterministe est non-déterministe.
- a. toujours vrai  
b. toujours faux  
c. parfois vrai  
d. c'est le contraire
- Q.10 Un langage quelconque
- a. n'est pas nécessairement dénombrable (i.e., il n'existe pas toujours de bijection entre ses mots et une partie de  $\mathbb{N}$ )  
b. est toujours inclus ( $\subset$ ) dans un langage rationnel

- c. peut n’être inclus dans aucun langage dénoté par une expression rationnelle
- d. peut avoir une intersection non vide avec son complémentaire

Q.11 Si  $L_1 \subseteq L \subseteq L_2$ , alors  $L$  est rationnel si :

- a.  $L_1$  est rationnel
- b.  $L_2$  est rationnel
- c.  $L_1, L_2$  sont rationnels
- d.  $L_1, L_2$  sont rationnels et  $L_2 \subseteq L_1$

Q.12 Un automate fini qui a des transitions spontanées. . .

- a. est déterministe
- b. n’est pas déterministe
- c. accepte  $\epsilon$
- d. n’accepte pas  $\epsilon$

Q.13 Quels langages ne vérifient pas le lemme de pompage?

- a. Tous les langages reconnus par un DFA
- b. Certains langages reconnus par un DFA
- c. Tous les langages non reconnus par un DFA
- d. Certains langages non reconnus par un DFA

Q.14 Si  $L_1, L_2$  sont rationnels, alors :

- a.  $\bigcup_{n \in \mathbb{N}} L_1^n \cdot L_2^n$  aussi
- b.  $(L_1 \cap L_2) \cup (\overline{L_1} \cap L_2)$  aussi
- c.  $L_1 \subseteq L_2$  ou  $L_2 \subseteq L_1$
- d.  $\overline{L_1 \cap L_2} = \overline{L_1} \cap \overline{L_2}$

Q.15 Si un automate de  $n$  états accepte  $a^n$ , alors il accepte. . .

- a.  $(a^n)^m$  avec  $m \in \mathbb{N}^*$
- b.  $a^{n+1}$
- c.  $a^n a^m$  avec  $m \in \mathbb{N}^*$
- d.  $a^p (a^q)^*$  avec  $p \in \mathbb{N}, q \in \mathbb{N}^* : p + q \leq n$

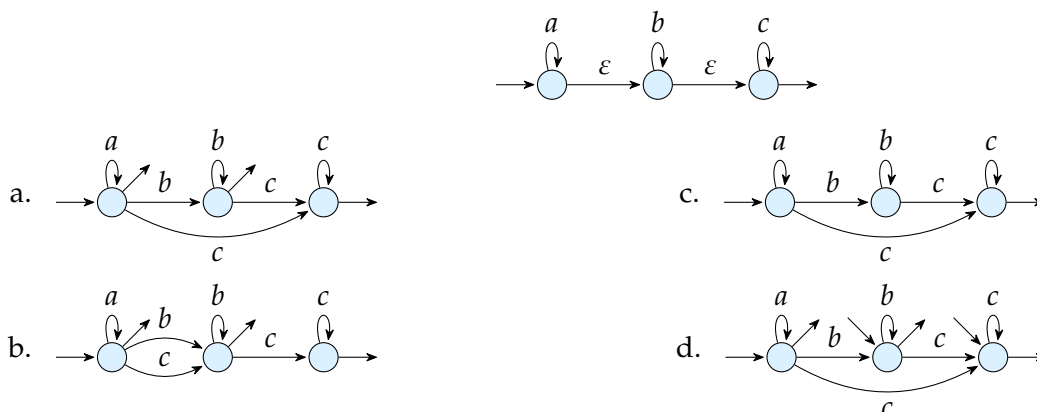
Q.16 Quelle séquence d’algorithmes teste l’appartenance d’un mot au langage représenté par une expression rationnelle?

- a. Thompson, déterminisation, Brzozowski-McCluskey.
- b. Thompson, déterminisation, élimination des transitions spontanées, évaluation.
- c. Thompson, élimination des transitions spontanées, déterminisation, minimisation, évaluation.
- d. Thompson, déterminisation, élimination des transitions spontanées, évaluation.

Q.17 Combien d’états a l’automate déterministe minimal qui accepte le langage  $(a + b + c + d)^+$ ?

- a. 1
- b. 2
- c. 3
- d. 4

Q.18 Quel est le résultat d’une élimination *arrière* des transitions spontanées sur l’automate suivant?



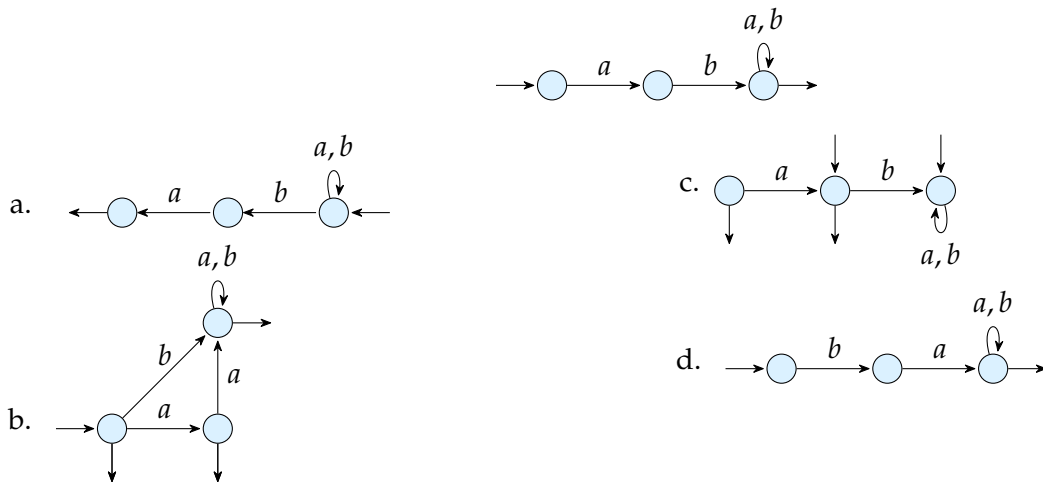
Q.19 Si l’on détermine la réponse de la question 18, puis qu’on le minimise, alors l’application de BMC conduira à une expression rationnelle équivalente à :

- a.  $a^*b^*c^*$
- b.  $a^* + b^* + c^*$
- c.  $(abc)^*$
- d.  $(a + b + c)^*$

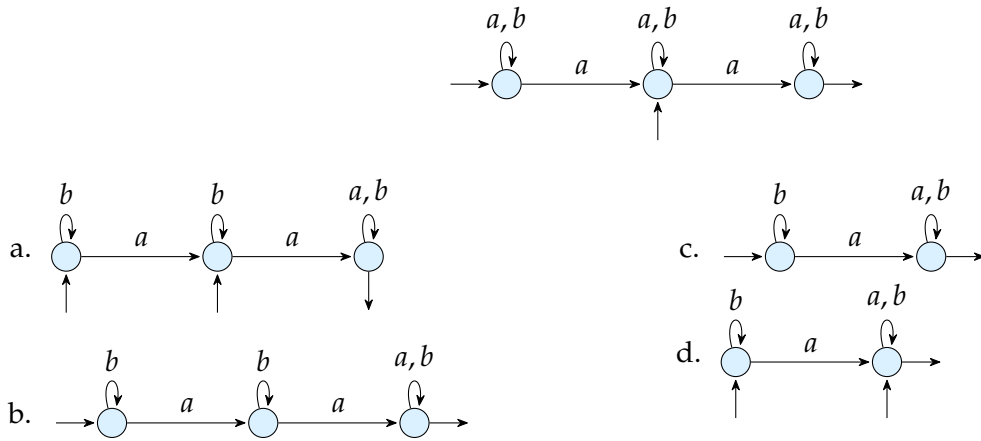
Q.20 L’automate de départ de la question 18 est. . .

- a. nondéterministe à transitions spontanées
- b. déterministe à transitions spontanées
- c.  $\epsilon$ -déterministe
- d.  $\epsilon$ -minimal

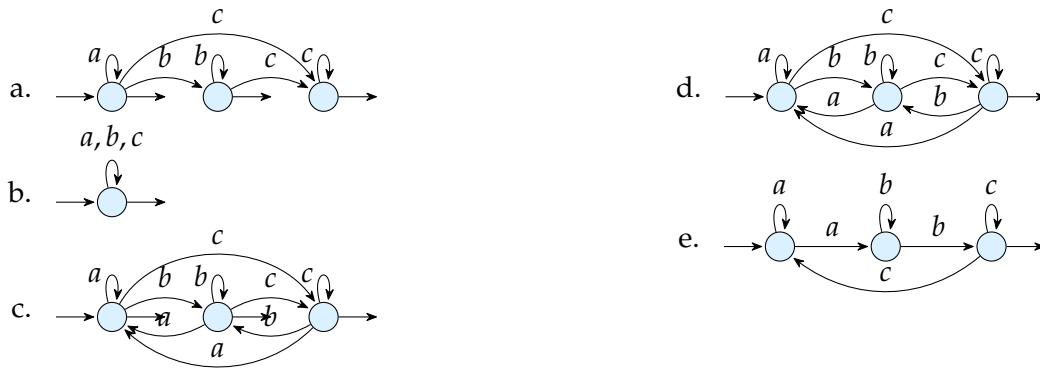
Q.21 Quel automate reconnaît le langage complémentaire de celui accepté par l'automate suivant ?



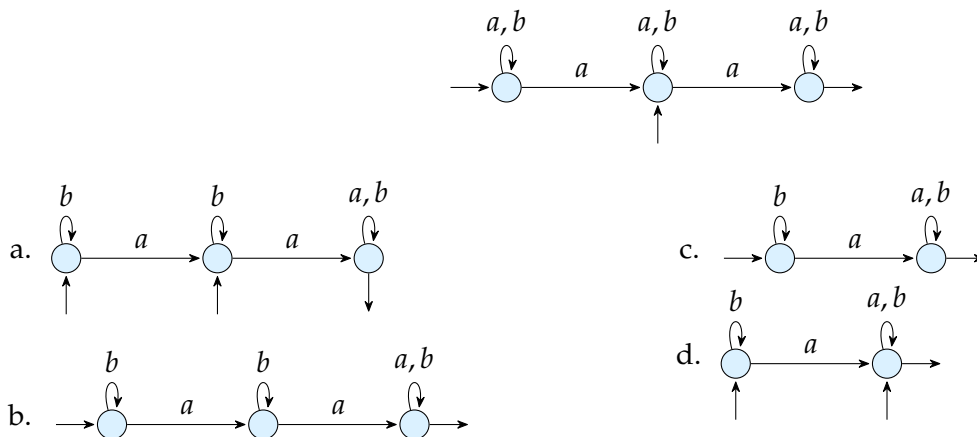
Q.22 Déterminer l'automate suivant.







Q.10 Déterminer l'automate suivant.



Q.11 A propos du lemme de pompage

- a. Si un langage le vérifie, alors il est rationnel
- b. Si un langage ne le vérifie pas, alors il n'est pas rationnel
- c. Si un langage ne le vérifie pas, alors il n'est pas forcément rationnel

Q.12 Soit le langage  $L = \{a^n b^m \mid (n, m) \in \mathbb{N}^2\}$

- a. Si je prends  $x \in L$  et  $(u, v, w) \in \Sigma^{*3}$  avec  $x = uvw$ . En posant  $u = a^i$ ,  $v = a^{n-i} b^{m-i}$  et  $w = b^i$ , clairement le mot  $uv^2w$  n'appartient pas à  $L$  donc  $L$  n'est pas rationnel.
- b.  $L$  est reconnaissable par un automate à états fini
- c.  $L$  est un langage fini

Q.13 Le langage sur  $\Sigma = \{a, b, n\}$  défini par  $anbn$

- a. est un langage fini
- b. n'est pas un langage rationnel et ne peut pas être reconnu par un automate à états finis.

Q.14 Soit 3 langages rationnels  $L_1, L_2$  et  $L_3$  tels que  $L_1 \subseteq L_2, L_3 \subseteq L_1$  et  $L_2 \subseteq L_3$ . Si  $L_4 \subseteq L_1$  et  $L_3 \subseteq L_4$ , alors...

- a.  $L_4$  n'est pas rationnel
- b.  $L_4$  est rationnel
- c. on ne peut pas conclure

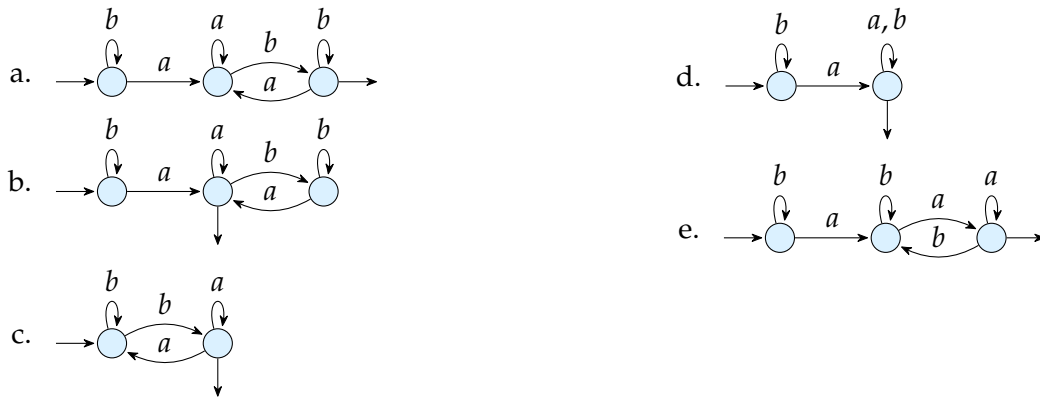
Q.15 Le langage de tous les noms communs du dictionnaire de la langue française

- a. n'est pas reconnaissable par un automate à états fini
- b. peut être décrit par une expression rationnelle
- c. est un langage infini

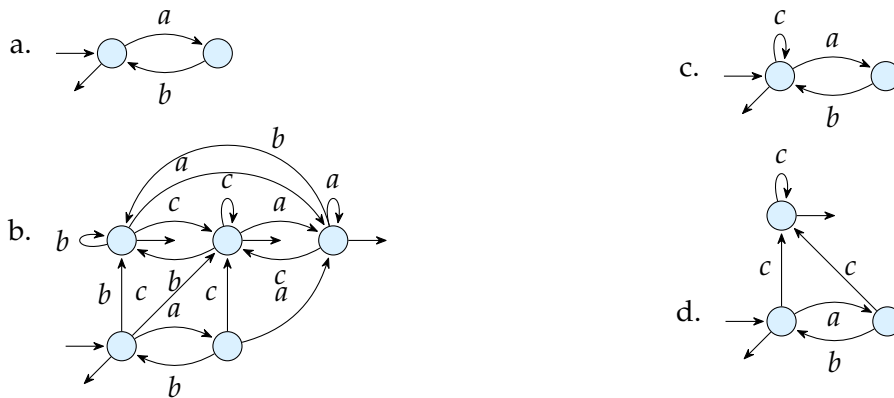
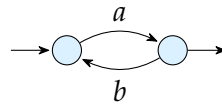
Q.16 L'intersection de deux langages rationnels

- a. n'est pas reconnaissable par un automate à états fini
- b. donne un langage interationnel
- c. peut être décrite par une expression rationnelle
- d. donne un langage rationnel sauf si leur intersection est disjointe

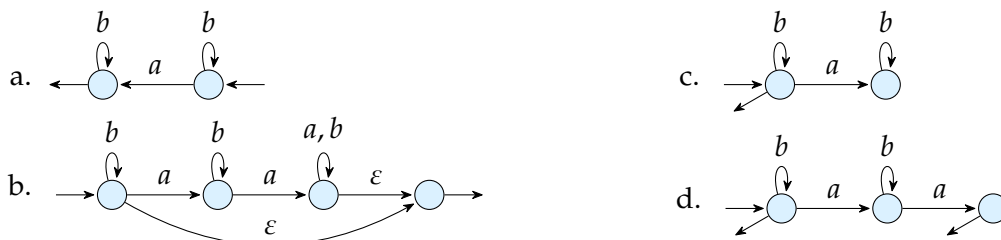
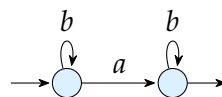
Q.17 Quel automate reconnaît l'intersection des langages définis par  $b^*a(a + b)^*$  et  $b^*a(a + bb^*a)^*$



Q.18 Quel automate reconnaît le langage complémentaire de celui reconnu par l'automate suivant sur l'alphabet  $\Sigma = \{a, b, c\}$  :

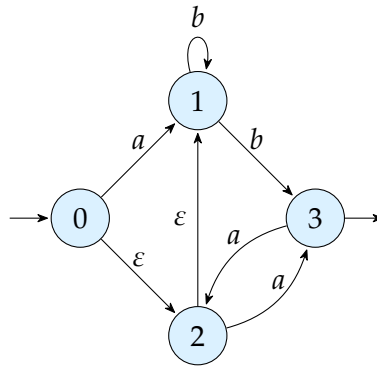


Q.19 Quel automate reconnaît le langage complémentaire (sur  $\Sigma = \{a, b\}$ ) de l'automate suivant.



Q.20 Quel est le résultat de l'application de BMC sur l'automate suivant en éliminant 1, puis 2, puis 3 et enfin 0?



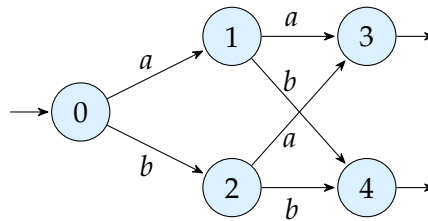


- a.  $(ab^* + (a + b)^*)(a + b)^+$
- b.  $(ab^* + (a + b)^*)a(a + b)^*$
- c.  $(ab^* + a + b^*)a(a + b)^*$
- d.  $(ab^* + a + b^*)a(a + b^*)$
- e.  $(ab^+ + a + b^*)a(a + b^*)$

Q.21 Soit  $\Sigma = \{a, b\}$ . Quelle expression définit le complémentaire du langage engendré par  $(a + b)^*b(a + b)^*$  ?

- a.  $(a + b)^*$
- b.  $(a + b)^*a(a + b)^*$
- c.  $\epsilon$
- d.  $a^*$

Q.22 Quels états peuvent être fusionnés dans l'automate suivant sans changer le langage reconnu.



- a. aucun
- b. 1 avec 2, et 3 avec 4
- c. 1 avec 2
- d. 3 avec 4
- e. 0 avec 1 et avec 2, et 3 avec 4



**Troisième partie**

**Annexes**





## Chapitre 9

# Compléments historiques

Ce chapitre porte sur l'histoire de l'informatique. Le but est de donner un contexte historique aux grandes notions et résultats abordés dans le cours, en expliquant brièvement qui ont été les logiciens et informaticiens qui ont laissé leurs noms aux notions de calculabilité et de langages formels.

### 9.1 Brèves biographies

#### John Backus (Américain, 1924 – 2007)

est le concepteur du premier langage de programmation de haut niveau compilé, Fortran. Co-auteur avec Peter Naur d'une notation standard pour les grammaires hors-contexte (*BNF* ou *Backus–Naur Form*), il a obtenu le prix Turing en 1977 pour ses travaux sur les langages de programmation.



#### Janusz Brzozowski (Polono-Canadien, né en 1935)

est un chercheur en théorie des automates, qui fut l'étudiant de Edward J. McCluskey. Il a apporté de nombreuses contributions dans ce domaine, et est en particulier connu pour l'*algorithme de minimisation de Brzozowski* et pour l'*algorithme de Brzozowski et McCluskey* ([section 4.2.2](#)).



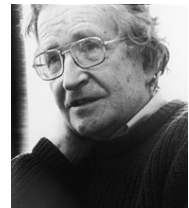
### Georg Cantor (Allemand, 1845 – 1918)

est l'inventeur de la théorie des ensembles et des notions d'ordinal et de cardinal d'un ensemble infini. Le nom de Cantor est donné à un ensemble dont il a imaginé la construction ; cet ensemble, parmi les premiers exemples de fractales, a des caractéristiques topologiques et analytiques très inhabituelles. Le *théorème de Cantor* énonce qu'un ensemble n'est jamais en bijection avec l'ensemble de ses parties, ce qui a pour conséquence que l'ensemble  $\mathbb{R}$  des nombres réels n'est pas en bijection avec l'ensemble  $\mathbb{N}$  des entiers naturels. Ce résultat, et l'argument de *diagonalisation* utilisé dans sa preuve, fut très mal accueilli par certains mathématiciens de l'époque, y compris de très grands comme Henri Poincaré. La fin de la vie de Georg Cantor fut marquée par une longue période de dépression et de récurrents séjours en hôpital psychiatrique.



### Noam Chomsky (Américain, né 1928)

est le fondateur de la linguistique moderne. Pour décrire le langage naturel, il a développé la notion de grammaire générative (en particulier, de grammaire hors-contexte) et a étudié les pouvoirs d'expression de différentes classes de grammaire au sein de la *hiérarchie de Chomsky*. Ces travaux s'appuient sur l'hypothèse d'une *grammaire universelle*, qui suppose que certains mécanismes du langage, universels et indépendants de la langue, sont pré-câblés dans le cerveau. Ses travaux ont eu un impact majeur en linguistique, en traitement du langage naturel, et en informatique théorique. Noam Chomsky est également connu pour son activisme politique très critique de la politique étrangère des États-Unis.



### Alonzo Church (Américain, 1903 – 1995)

était un pionnier de la logique informatique. Stephen C. Kleene et Alan Turing furent ses étudiants. Il proposa le  $\lambda$ -calcul (ancêtre des langages de programmation fonctionnels comme Lisp) comme modèle de calculabilité et démontra, indépendamment de Turing, l'impossibilité de résoudre le problème de décision de Hilbert (*théorème de Church–Turing*). Il est également connu pour la *thèse de Church–Turing*, une hypothèse qui postule que les différents modèles de calculabilité proposés ( $\lambda$ -calcul, machines de Turing, fonctions récursives générales, machines de von Neumann), tous démontrés équivalents, correspondent à la notion intuitive de calculabilité ; dans sa version la plus générale, cette thèse affirme que tout ce qui peut être calculé par la nature, et donc par le cerveau humain, peut l'être par une machine de Turing.



**Augustus De Morgan (Britannique, 1806 – 1871)**

fut l'un des premiers à tenter de baser les mathématiques sur un raisonnement logique formel. Les *lois de De Morgan* expriment que la négation d'une disjonction est la conjonction des négations, et vice-versa.



**Victor Glushkov (Soviétique, 1923 – 1982)**

fut un pionnier de la théorie de l'information, de la théorie des automates et de la conception des premiers ordinateurs en Union soviétique. Il a donné son nom à la construction de Glushkov d'un automate à partir d'une expression rationnelle, alternative à la construction de Thompson.



**Kurt Gödel (Autrichien, 1906 – 1978)**

est connu pour ses travaux sur la théorie des fonctions récursives, un modèle de calculabilité, mais surtout pour deux résultats fondamentaux de logique mathématique : le théorème de *complétude* de Gödel montre que tout résultat vrai dans tout modèle de la logique du premier ordre est démontrable ; le théorème d'*incomplétude* de Gödel montre que dans tout système formel permettant d'exprimer l'arithmétique, il existe des énoncés mathématiques que l'on ne peut ni prouver ni infirmer. Il mourut dans des circonstances tragiques : victime de paranoïa, il était convaincu qu'on cherchait à l'empoisonner, et refusa petit à petit de s'alimenter.



**Sheila Greibach (Américaine, née 1939)**

est une chercheuse en langages formels. Elle est en particulier connue pour la *forme normale de Greibach* d'une grammaire hors-contexte et ses conséquences sur l'équivalence entre grammaires hors-contexte et automates à pile.



### David Hilbert (Allemand, 1862 – 1943)

est reconnu comme l'un des plus grands mathématiciens ayant existé. Il a apporté des contributions majeures dans de nombreux domaines, comme la géométrie, l'analyse (p. ex., *espaces de Hilbert*), la physique mathématique et la logique. En 1900, il énonce 23 problèmes ouverts importants des mathématiques. Cette liste a eu une très grande influence sur la recherche en mathématique au 20<sup>e</sup> siècle. L'un de ces problèmes, le problème de décision (ou *Entscheidungsproblem*), consistait à obtenir une procédure automatique permettant de décider si un énoncé mathématique est vrai ou faux. En 1920, Hilbert énonce un programme pour l'axiomatisation des mathématiques, par lequel tout énoncé mathématique pourrait être prouvé ou infirmé à partir des axiomes. Kurt Gödel a montré que cela était impossible; Alonzo Church et Alan Turing en ont indépendamment déduit l'impossibilité de résoudre le problème de décision. L'épithète de David Hilbert résume sa vision de la science :



Wir müssen wissen.      (*Nous devons savoir.*)  
Wir werden wissen.     (*Nous saurons.*)

### Stephen C. Kleene (Américain, 1909 – 1994)

est l'inventeur des expressions rationnelles et a apporté des contributions majeures à la théorie des langages récursifs, un modèle de calculabilité. L'opérateur d'*étoile*, caractéristique des expressions rationnelles, porte son nom, de même que le théorème d'équivalence entre expressions rationnelles et automates finis ([section 4.2.2](#)).



### Vladimir Levenshtein (Russe, né en 1935)

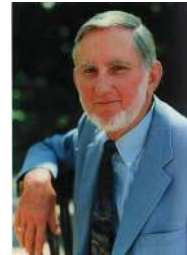
est un chercheur dans les domaines de la théorie de l'information et des codes correcteurs d'erreur. Il a introduit dans ce cadre la distance d'édition qui porte son nom.





### Edward J. McCluskey (Américain, né en 1929)

est connu pour ses travaux sur la conception logique des circuits électroniques. Il est en particulier l'auteur avec W. V. Quine de la *méthode de Quine-McCluskey* de minimisation de fonctions booléennes en utilisant la notion d'impliquants premiers.



### Edward F. Moore (Américain, 1925 – 2003)

est l'un des fondateurs de la théorie des automates. Il introduit l'algorithme de minimisation qui porte son nom, et la notion d'automate avec sortie (introduite également indépendamment par George H. Mealy).

### Peter Naur (Danois, 1928 – 2016)

est un chercheur en informatique qui a travaillé en particulier sur la conception des langages de programmation. Co-auteur avec John Backus d'une notation standard pour les grammaires hors-contexte (*BNF* ou *Backus-Naur Form*), il a obtenu le prix Turing en 2005 pour ses travaux sur le langage Algol.



### John von Neumann (Hongro-Américain, 1903 – 1957)

, en plus d'être un des pères fondateurs de l'informatique, a fourni des contributions majeures dans de nombreux domaines scientifiques (en particulier, théorie des ensembles, mécanique quantique, théorie des jeux). En informatique, il est connu pour les automates cellulaires, le tri fusion, et bien sûr l'architecture de von Neumann, le modèle formel de calcul qui se rapproche le plus des ordinateurs actuels. Ce modèle a été élaboré lors de ses travaux sur la construction des premiers ordinateurs (ENIAC, EDVAC). Durant la deuxième guerre mondiale, il a travaillé dans le projet américain Manhattan de conception de la bombe atomique, et a fait partie de la commission américaine à l'énergie atomique pendant la guerre froide. L'organisation IEEE décerne chaque année une très prestigieuse médaille John von Neumann à un chercheur en informatique.



### Ken Thompson (Américain, né en 1943)

est un informaticien notable pour ses contributions à la fois théoriques et appliquées. Il a popularisé les expressions rationnelles en créant notamment l'utilitaire Unix `grep`, basé sur la construction (*de Thompson*, [section 4.2.2](#)) d'un automate décrivant l'expression rationnelle. Il a travaillé sur la conception du langage B avec Dennis Ritchie, ancêtre du langage C. Toujours avec Ritchie, il est à l'origine du système d'exploitation Unix. Il a également élaboré avec Rob Pike l'encodage de caractères UTF-8. Il obtient le prix Turing en 1983 avec Dennis Ritchie pour leurs recherches sur les systèmes d'exploitation et l'implémentation d'Unix.



### Alan Turing (Britannique, 1912 – 1954)

est souvent considéré comme le premier informaticien. Ses travaux sur le problème de décision le conduisent à introduire le modèle de calculabilité connu aujourd'hui sous le nom de machine de Turing. En se basant sur les travaux de Kurt Gödel, il prouve l'indécidabilité de l'arrêt d'une machine de Turing, ce qui montre l'impossibilité d'obtenir une solution générale au problème de décision (*théorème de Church–Turing*). Durant la seconde guerre mondiale, il fut l'un des principaux cryptanalystes qui déchiffrèrent les codes de communication de l'armée allemande. Après la guerre, il travailla sur la conception des premiers ordinateurs. Alan Turing est également connu pour la notion de *test de Turing*, une expérience de pensée permettant de définir la notion d'intelligence artificielle. En 1952, Alan Turing fut condamné pour homosexualité à une castration chimique. Il meurt mystérieusement en 1954 (empoisonnement par une pomme enrobée de cyanure, à l'image du film de Walt Disney *Blanche-Neige*, qu'il affectionnait particulièrement), probablement un suicide. La société savante ACM décerne chaque année un prix Turing, considéré comme la plus haute distinction que peut recevoir un chercheur en informatique.



## 9.2 Pour aller plus loin

Pour approfondir l'histoire de la théorie des langages, voir les auteurs suivants : [Hodges \(1992\)](#), [Doxiadis et Papadimitriou \(2010\)](#), [Dowek \(2007\)](#), [Turing \(1936\)](#), [Barsky \(1997\)](#), [Perrin \(1995\)](#).

### **Attributions**

Certaines photographies sont reproduites en vertu de leur licence d'utilisation Creative Commons, et attribuées à :

- John Backus : Pierre Lescanne
- Stephen C. Kleene : Konrad Jacobs (Mathematisches Forschungsinstitut Oberwolfach)
- Peter Naur : Erik tj (Wikipedia EN)
- Kenneth Thompson : Bojars (Wikimedia)

Les photographies suivantes sont reproduites en vertu du droit de citation :

- Janusz Brzozowski
- Alonzo Church
- Sheila Greibach
- Vladimir Levenshtein
- Edward F. McCluskey

Les autres photos sont dans le domaine public.



# Chapitre 10

## Correction des exercices

### 10.1 Correction de l'exercice 4.5

1. Pour  $L_1$ , on vérifie que le choix de  $I_1 = \{a\}$ ,  $F_1 = \{a\}$  et  $T_1 = \{ab, ba, bb\}$  garantit  $L_1 = (I_1\Sigma^* \cap \Sigma^*F_1) \setminus \Sigma^*T_1\Sigma^*$ .

Pour  $L_2$ , il en va de même pour  $I_2 = \{a\}$ ,  $F_2 = \{b\}$  et  $T_2 = \{aa, bb\}$ .

**Note :** Pour  $L_1$ , il suffit de prendre  $T_1 = \{ab\}$  pour interdire toute occurrence de  $b$  dans  $L_1$ .

2. Si l'on veut accepter les mots de  $L_1$  et de  $L_2$ , il est nécessaire d'avoir  $a \in I$ ,  $a, b \in F$ , et  $T$  contient au plus  $bb$ . Deux possibilités : soit  $T$  est vide, mais alors le langage correspondant ne contient que  $a$ ; soit  $T = \{bb\}$ , auquel cas un mot comme  $aab$  est dans le langage local défini par  $I, F$  et  $T$ , mais pas dans le langage dénoté par  $aa^* + ab(ab)^*$ .
3. Pour l'intersection, il suffit d'utiliser le fait que  $A \setminus B = A \cap \bar{B}$ .  $L_1 \cap L_2$  s'écrit alors comme une intersection de six termes; en utilisant que le fait que l'intersection est associative et commutative, et que la concaténation est distributive par rapport à l'intersection, il vient :

$$L_1 \cap L_2 = ((I_1 \cap I_2)\Sigma^* \cap \Sigma^*(F_1 \cap F_2)) \setminus (\Sigma^*(T_1 \cup T_2)\Sigma^*)$$

Ce qui prouve que l'intersection de deux langages locaux est un langage local.

Pour l'union, le contre-exemple de la question 2 permet de conclure directement.

4.  $I, F$  et  $T$  sont finis donc rationnels,  $\Sigma^*$  également; les rationnels sont stables par concaténation, intersection, complément. Ceci suffit pour conclure que tout langage de la forme  $(I\Sigma^* \cap \Sigma^*F) \setminus \Sigma^*T\Sigma^*$  est rationnel.
5. Un langage local reconnaissant les mots de  $C$  doit nécessairement avoir  $a, b \in I, a, b, c \in F$ . On cherche le plus petit langage possible, ce qui conduit à interdire tous les facteurs de longueur 2 qui ne sont pas dans au moins un mot de  $c$  :  $T = \{ac, bb, ba, cb, cc\}$ .

L'automate correspondant a un état initial  $q_0$ , qui a des transitions sortantes sur les symboles de  $I$ , et un état par lettre :  $q_a, q_b, q_c$ . Ces états portent la « mémoire » du dernier symbole lu (après un  $a$ , on va dans  $q_a$ ...). Les transitions sortantes de ces états sont définies de façon à interdire les facteurs de  $T$  : après un  $a$ , on peut avoir  $a$  ou  $b$  mais pas

---

$c$ , d'où les transitions  $\delta(q_a, a) = q_a$ ,  $\delta(q_a, b) = q_b$ . De même, on a :  $\delta(q_b, c) = q_c$ ,  $\delta(q_c, a) = q_a$ .  
Finalement, un état  $q_x$  est final si et seulement  $x \in F$ . Ici  $q_a, q_b$  et  $q_c$  sont dans  $F$ .

**Note :** De nombreux élèves ont correctement identifié  $I, F$  et  $T$  mais ont construit un automate qui ne reconnaissait qu'un ensemble fini de mots, ce qui n'est pas possible ici.

6. L'intuition de ce beau résultat est qu'un automate fini  $A$  est défini par son état initial, ses états finaux, et ses transitions. Transposés dans l'ensemble des chemins dans  $A$ , l'état initial est caractérisé par les préfixes de longueur 1 des chemins; les états finaux par les suffixes de longueur 1, et les transitions par des facteurs de longueur 2.

Formellement, soit  $L$  rationnel et  $A = (\Sigma, Q, q_0, F, \delta)$  un automate fini déterministe reconnaissant  $L$ . On définit  $L'$  sur l'alphabet  $\Sigma' = Q \times \Sigma \times Q$  par les trois ensembles :

- $I = \{(q_0, a, \delta(q_0, a)), a \in \Sigma\}$
- $F = \{(q, a, r), q \in Q, a \in \Sigma, r \in \delta(a, q) \cap F\}$
- $\Sigma' \times \Sigma' \setminus \{(q, a, r)(r, b, p), q \in Q, a, b \in \Sigma, r \in \delta(a, q), b \in \delta(r, p)\}$ .

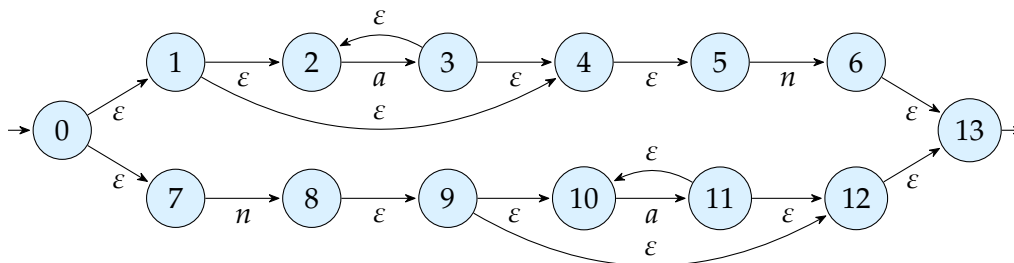
Soit maintenant un mot  $u$  de  $L$ . Son calcul dans  $A$  est par construction un mot  $w$  de  $L'$  qui vérifie de plus  $\phi(w) = u$ , donc  $L \subset \phi(L')$ . Réciproquement, si  $u = \phi(w)$ , avec  $w$  dans  $L'$ , alors on déduit immédiatement un calcul pour  $u$  dans  $A$ , et donc  $u \in L$ .

## 10.2 Correction de l'exercice 4.10

1.  $A$  n'est pas déterministe. L'état 0 a deux transitions sortantes étiquetées par le symbole  $b$ , vers 0 et vers 1.
2. Le langage reconnu par  $A$  est l'ensemble des mots se terminant par le suffixe  $b(c + a)^*d$ , soit  $\Sigma^*b(c + a)^*d$ .
3. À faire...

## 10.3 Correction de l'exercice 4.14

1. La construction de Thompson conduit à l'automate 10.1.



Automate 10.1 – Automate d'origine pour  $(a^*n \mid na^*)$

État	Fermeture	État	Fermeture
0	0, 1, 2, 4, 5, 7	7	7
1	1, 2, 4, 5	8	8, 9, 10, 12, 13
2	2	9	9, 10, 12, 13
3	2, 3, 4, 5	10	10
4	4, 5	11	10, 11, 12, 13
5	5	12	12, 13
6	6, 13	13	13

TABLE 10.2 –  $\varepsilon$ -fermetures (avant) des états de l'automate 10.1

Pour chaque état, l'ensemble des états accessibles par transitions spontanées.

- Le calcul de l' $\varepsilon$ -fermeture conduit aux résultats du tableau 10.2.
- La suppression des transitions  $\varepsilon$  s'effectue en court-circuitant ces transitions. La suppression *arrière* des transitions spontanées consiste à agréger les transitions spontanées avec les transitions non spontanées qui sont à leur aval. En d'autres termes, pour toute transition non spontanée  $\delta(q, a) = q'$ , on ajoute une transition partant de chaque état en amont de  $q$ , sur l'étiquette  $a$ , arrivant en  $q'$ . De même, tout état en amont d'un état final devient final.

En utilisant le tableau des clôtures avant (tableau 10.2), ajouter une transition  $\delta(p, a) = q'$  pour tout état  $p$  tel que  $q$  appartienne à l' $\varepsilon$ -fermeture avant de  $p$ . Marquer ensuite comme final tous les états dont la fermeture avant contient un état final.

État	Fermeture	État	Fermeture
0	0	7	0, 7
1	0, 1	8	8
2	0, 1, 2, 3	9	8, 9
3	3	10	8, 9, 10, 11
4	0, 1, 3, 4	11	11
5	0, 1, 3, 4, 5	12	8, 9, 11, 12
6	6	13	6, 8, 9, 11, 12, 13

TABLE 10.3 –  $\varepsilon$ -fermetures arrières des états de l'automate 10.1

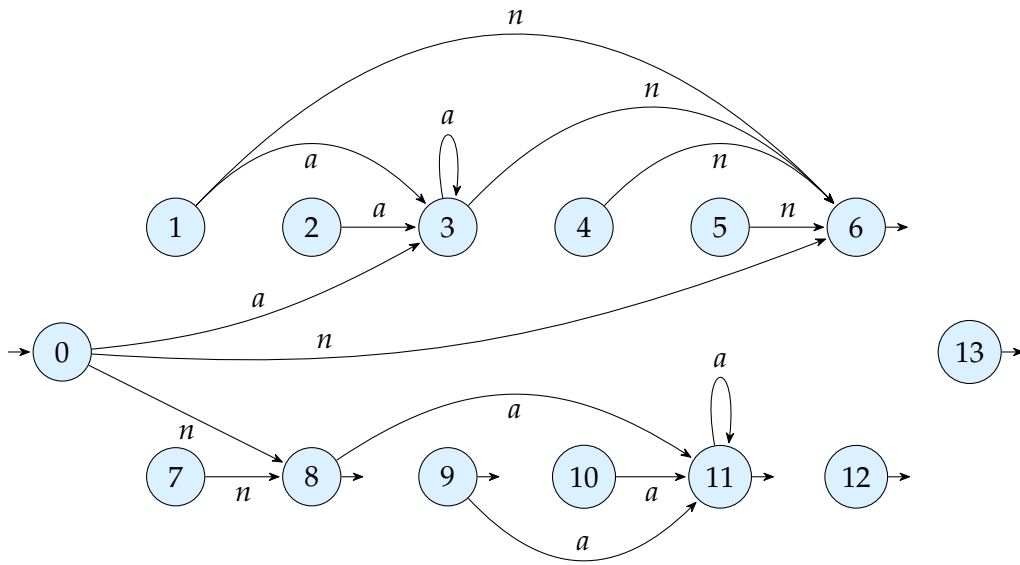
Pour chaque état, l'ensemble des états co-accessibles par transitions spontanées.

Alternativement, en utilisant le tableau des clôtures arrières (tableau 10.3), il faut donc ajouter une transition  $\delta(p, a) = q'$  pour tout état  $p$  dans l' $\varepsilon$ -fermeture arrière de  $q$ . On marque ensuite comme final tous les états dans la fermeture arrière des états final.

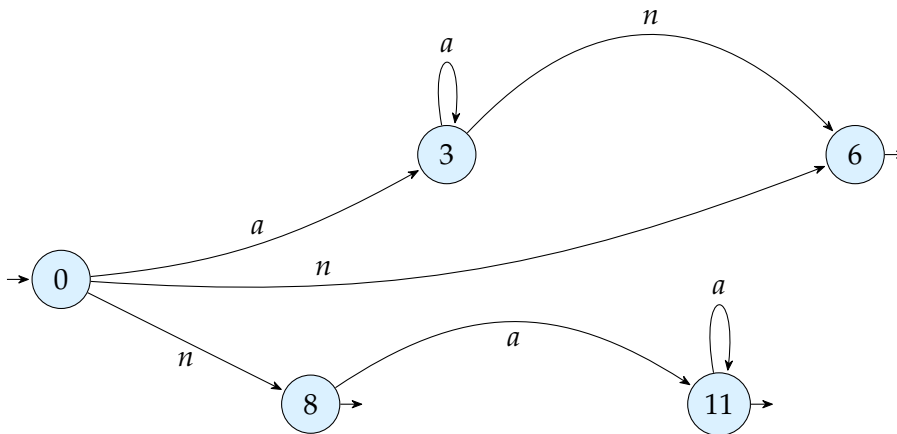
Cette procédure aboutit à l'automate 10.4.

Les états 1, 2, 4, 5, 7, 9, 10, 12 et 13 sont devenus inutiles dans l'automate 10.4, si on les élimine, on obtient l'automate 10.5. Cet automate n'est pas déterministe, l'état 0 ayant deux transitions sortantes pour le symbole  $n$ .

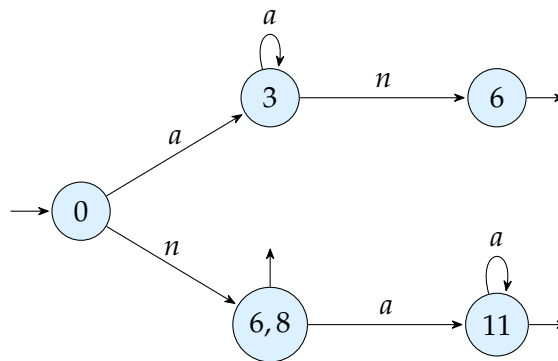
- Après déterminisation par la méthode des sous-ensembles, on aboutit finalement à l'automate 10.6.



Automate 10.4 – Automate sans transition spontanée pour  $(a^*n \mid na^*)$



Automate 10.5 – L'automate 10.4, émondé

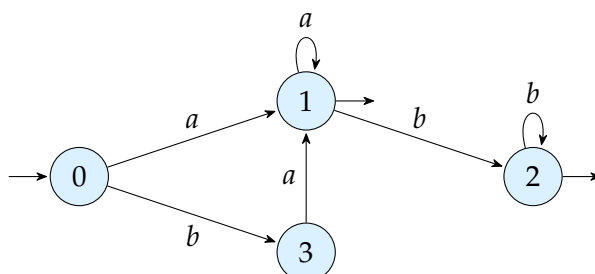


Automate 10.6 – Automate déterministe pour  $(a^*n \mid na^*)$



## 10.4 Correction de l'exercice 4.15

- Après élimination de la transition spontanée, la méthode de construction des sous-ensembles permet de construire l'automate 10.7.



Automate 10.7 – Déterminisation de l'automate 4.14

## 10.5 Correction de l'exercice 4.19

- Les automates  $A_1$  et  $A_2$  sont représentés respectivement verticalement et horizontalement dans la représentation de l'automate 10.8.
- L'application de l'algorithme construisant l'intersection conduit à l'automate 10.8.
- L'algorithme d'élimination des transitions  $\varepsilon$  demande de calculer dans un premier temps l' $\varepsilon$ -fermeture de chaque état. Puisque les seules transitions  $\varepsilon$  sont celles ajoutées par la construction de l'union, on a :

$$- \varepsilon\text{-closure}(q_0) = \{q_0^1, q_0^2\}$$

$$- \forall q \in Q^1 \cup Q^2, \varepsilon\text{-closure}(q) = \{q\}$$

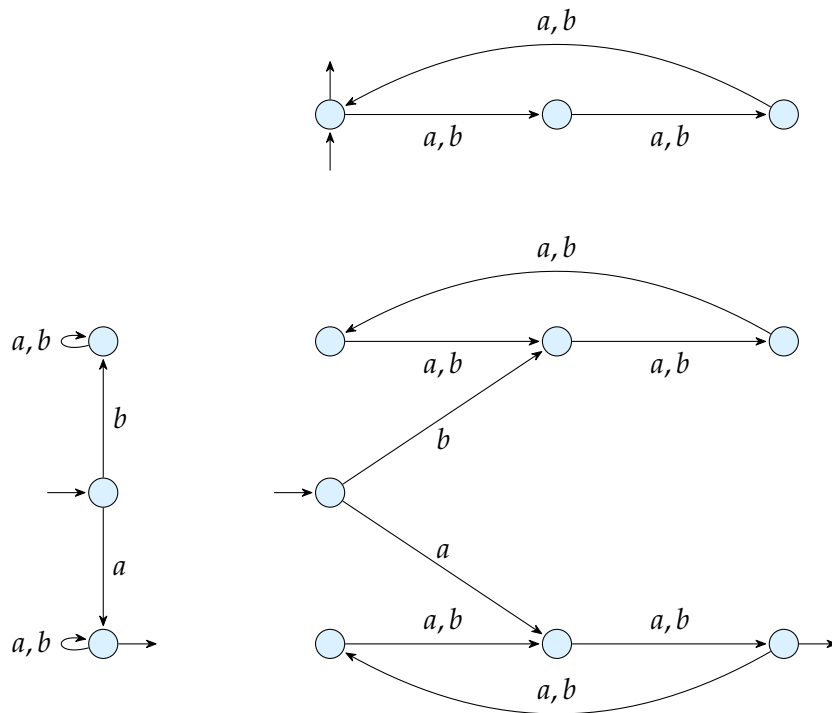
L'élimination des deux transitions  $\varepsilon$  conduit donc à ajouter les transitions suivantes :  $\forall a \in \Sigma, \delta(q_0, a) = \{\delta(q_0^1, a), \delta(q_0^2, a)\}$ .  $q_0$  est bien non-déterministe et c'est le seul état dans ce cas, puisqu'aucune autre transition n'est rajoutée.

- L'algorithme de construction de la partie utile du déterminisé construit de proche en proche les transitions des états accessibles depuis  $q_0$ . On l'a vu, le traitement de  $q_0$  ne construit que des états utiles de la forme  $\{\delta(q_0^1, a), \delta(q_0^2, a)\}$ , qui correspondent à des paires de  $Q^1 \times Q^2$ , car  $A_1$  et  $A_2$  sont déterministes. Supposons qu'à l'étape  $n$  de l'algorithme, on traite un état  $q = \{q^1, q^2\}$ , avec  $q^1 \in Q^1$  et  $q^2 \in Q^2$ . Par définition du déterminisé on a :

$$\forall a \in \Sigma, \delta(q, a) = \delta^1(q^1, a) \cup \delta^2(q^2, a)$$

Les  $A_i$  étant déterministes, chacun des  $\delta^i(q^i, a)$  est un singleton de  $Q^i$  (pour  $i = 1, 2$ ), les nouveaux états utiles créés lors de cette étape correspondent bien des doubletons de  $Q^1 \times Q^2$ .

On note également que, par construction, les transitions de  $\bar{A}$  sont identiques aux transitions de la construction directe de l'intersection.



Automate 10.8 – Les automates pour  $A_1$  (à gauche),  $A_2$  (en haut) et leur intersection (au milieu)

- c. Les états finaux du déterminisé sont ceux qui contiennent un état final de  $\overline{A^1}$  ou de  $\overline{A^2}$ . Les seuls non-finaux sont donc de la forme :  $\{q^1, q^2\}$ , avec à la fois  $q^1$  non-final dans  $\overline{A^1}$  et  $q^2$  non-final dans  $\overline{A^2}$ . Puisque les états non-finaux de  $\overline{A^1}$  sont précisément les états finals de  $A^1$  (et idem pour  $A_2$ ), les états finals de  $A$  correspondent à des doubletons  $\{q^1, q^2\}$ , avec  $q^1$  dans  $F^1$  et  $q^2$  dans  $F^2$ .

On retrouve alors le même ensemble d'états finaux que dans la construction directe.

La dernière vérification est un peu plus fastidieuse et concerne l'état initial  $q_0$ . Notons tout d'abord que  $q_0$  n'a aucune transition entrante (cf. le point [b]). Notons également que, suite à l'élimination des transitions  $\varepsilon$ ,  $q_0$  possède les mêmes transitions sortantes que celles qu'aurait l'état  $\{q_0^1, q_0^2\}$  du déterminisé. Deux cas sont à envisager :

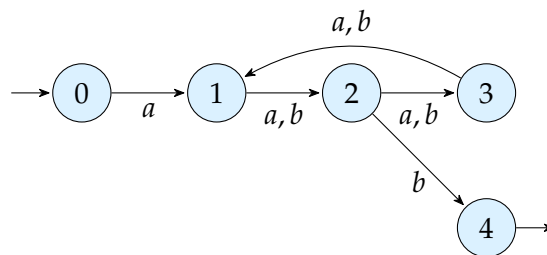
- soit la partie  $\{q_0^1, q_0^2\}$  n'est pas construite par l'algorithme de déterminisation et on peut assimiler  $q_0$  à cette partie ;
- soit elle est construite et on peut alors rendre cette partie comme état initial et supprimer  $q_0$  : tous les calculs réussis depuis  $q_0$  seront des calculs réussis depuis  $\{q_0^1, q_0^2\}$ , et réciproquement, puisque ces deux états ont les mêmes transitions sortantes.

## 10.6 Correction de l'exercice 4.20

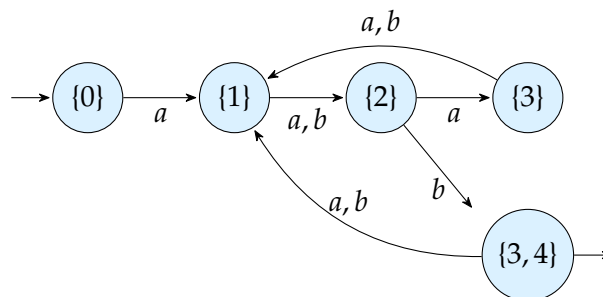
Le langage contenant les mots dont la longueur est divisible par 3 est reconnu par un automate à trois états, chaque état correspondant à un reste de la division par 3. De 0 on transite vers 1 (indépendamment de l'entrée); de 1 on transite vers 2, et de 2 vers 0, qui est à la fois initial et final.

Le langage  $a\Sigma^*b$  est reconnu par un automate à 3 états. L'état initial a une unique transition sortante sur le symbole d'entrée  $a$  vers l'état 1, dans lequel on peut soit boucler (sur  $a$  ou  $b$ ) ou bien transiter dans 2 (sur  $b$ ). 2 est le seul état final.

L'intersection de ces deux machines conduit à  $A_1$ , l'automate 10.9 non-déterministe, qui correspond formellement au produit des automates intersectés. L'application de la méthode des sous-ensembles conduit à  $A_2$ , l'automate 10.10, déterministe.



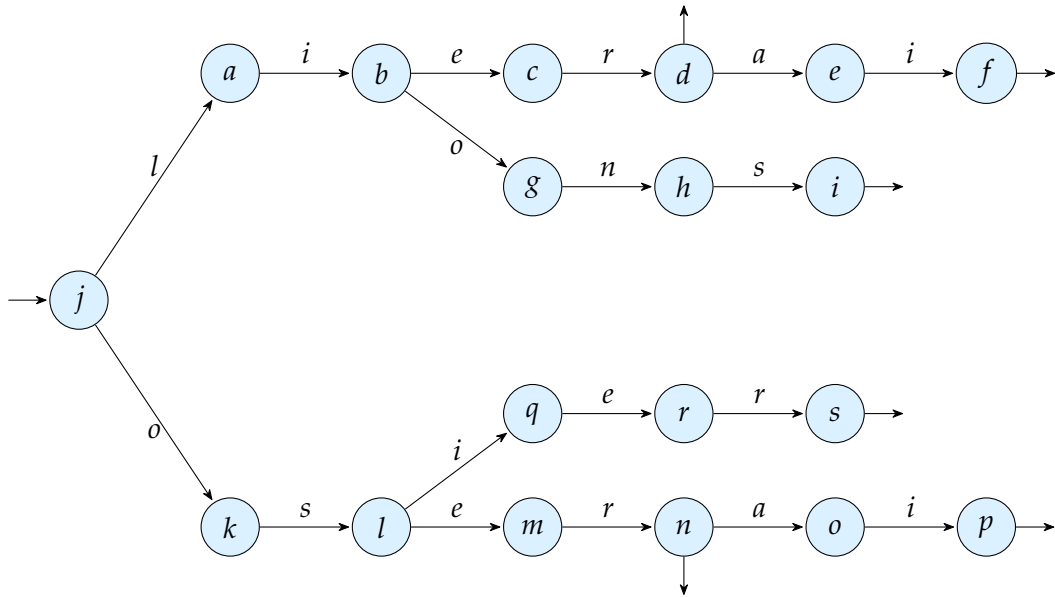
Automate 10.9 – L'automate  $A_1$  de la question 1



Automate 10.10 – L'automate  $A_2$  de la question 1

## 10.7 Correction de l'exercice 4.35

1. En utilisant les méthodes du cours, on obtient l'automate 10.11, déterminisé. Cet automate est appelé *l'arbre accepteur des préfixes* (il comprend exactement un état par préfixe dans le langage).
2. (a) Les états ( $f$ ) et ( $p$ ) sont deux états finaux sans transition sortante : ils sont indistinguables ; l'état ( $f$ ) est final quand ( $e$ ) est non-final : le mot  $\varepsilon$  permet de les distinguer (i.e. qu'il conduit à un calcul réussi depuis ( $f$ ) mais pas depuis ( $e$ )).



Automate 10.11 – Le dictionnaire déterminisé

- (b) Notons  $\mathcal{I}$  la relation d'indistinguabilité : cette relation est trivialement réflexive et symétrique. Soient  $p$  et  $q$  indistinguables et  $q$  et  $r$  indistinguables : supposons que  $p$  et  $r$  soient distingués par le mot  $v$ . On a alors un calcul réussi depuis  $p$  qui échoue depuis  $r$ ; ceci est absurde : s'il réussit depuis  $p$  il réussira depuis  $q$ , et réussira donc également depuis  $r$ .
3. La combinaison de (f) et (p) conduit à un automate dans lequel ces deux états sont remplacés par un unique état  $(fp)$ , qui est final.  $(fp)$  possède deux transitions entrantes, toutes deux étiquetées par  $i$ ;  $(fp)$  n'a aucune transition sortante.
  4. Supposons que  $A'$  résulte de la fusion de  $p$  et  $q$  indistinguables dans  $A$ , qui sont remplacés par  $r$ . Il est tout d'abord clair que  $L(A) \subset L(A')$  : soit en effet  $u$  dans  $L(A)$ , il existe un calcul réussi pour  $u$  dans  $A$  et :
    - soit ce calcul évite les états  $p$  et  $q$  et le même calcul existe dans  $A'$  ; avec les notations de l'énoncé, il en va de même si ce calcul évite l'état  $q$ .
    - soit ce calcul utilise au moins une fois  $q$  :  $(q_0, u) \vdash_A (q, v) \vdash_A (s, \varepsilon)$ . Par construction de  $A'$  on a  $(q_0, u) \vdash_{A'} (r, v)$  ; par ailleurs  $p$  et  $q$  étant indistinguable, il existe un calcul  $(p, v) \vdash_A (s', \varepsilon)$  avec  $s' \in F$  dans  $A$  qui continue d'exister dans  $A'$  (au renommage de  $p$  en  $r$  près). En combinant ces deux résultats, on exhibe un calcul réussi pour  $u$  dans  $A'$ .

Supposons maintenant que  $L(A)$  soit strictement inclus dans  $L(A')$  et que le mot  $v$  de  $L(A')$  ne soit pas dans  $L(A)$ . Un calcul réussi de  $v$  dans  $A'$  utilise nécessairement le nouvel état  $r$ , sinon ce calcul existerait aussi dans  $A$ ; on peut même supposer qu'il utilise une fois exactement  $r$  (s'il utilise plusieurs fois  $r$ , on peut court-circuiter les boucles).

On a donc :  $(q'_0, v = v_1 v_2) \vdash_{A'} (r, v_2) \vdash_{A'} (s, \varepsilon)$ , avec  $s \in F'$ . Dans  $A$ , ce calcul devient soit  $(q_0, v_1) \vdash_A p$  suivi de  $(q, v_2) \vdash_A (s, \varepsilon)$ ; soit  $(q_0, v_1) \vdash_A q$  suivi de  $(p, v_2) \vdash_A (s, \varepsilon)$ ; dans la

première de ces alternatives, il est de plus assuré que, comme  $v$  n'est pas dans  $A$ , il n'existe pas de calcul  $(p, v_2)$  aboutissant dans un état final. Ceci contredit toutefois le fait que  $p$  et  $q$  sont indistinguables.

5. Le calcul de la relation d'indistinguabilité demande de la méthode : dans la mesure où l'automate est sans cycle, ce calcul peut s'effectuer en considérant la longueur du plus court chemin aboutissant dans un état final.

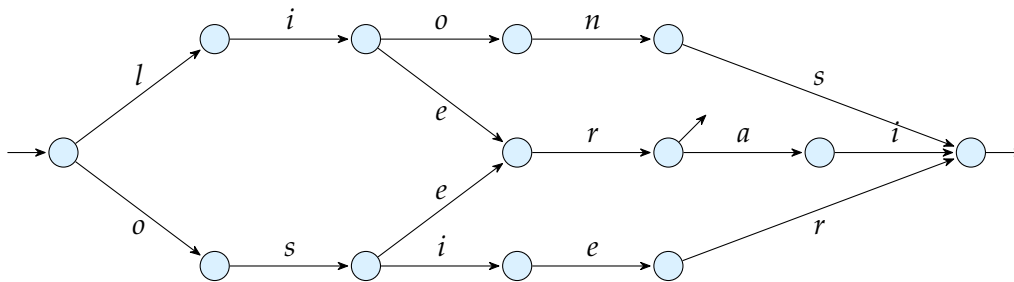
Commençons par l'initialisation :  $\{d, f, i, n, s, p\}$  sont distingués de tous les autres états par leur « finalité ». On distingue deux sous-ensembles :  $\{f, i, s, p\}$  sont distingués de  $d$  et de  $n$  par le mot  $ai$ , mais sont indistinguables entre eux. À ce stade, on ne peut conclure sur  $d$  et  $n$ .

Les états desquels débute un calcul réussi de longueur 1 sont :  $c, e, h, r, m, o$  :  $c$  et  $m$  sont distingués des autres par le mot  $rai$ ,  $r$  et  $h$  sont distingués du couple  $(e, o)$  par respectivement  $s$  et  $r$ , et sont aussi distingués entre eux ;  $e$  et  $o$  sont indistinguables.

Les états depuis lesquels débutent un calcul réussi de longueur 2 sont  $d, g, q, n$ .  $g$  et  $q$  sont distingués des deux autres, et distingués entre eux. En revanche,  $d$  et  $n$  sont indistinguables.

En poursuivant ce raisonnement, on aboutit finalement aux classes d'équivalence suivantes :  $\{f, i, p, s\}, \{e, o\}, \{d, n\}, \{m, c\}, \{a\}, \{b\}, \{g\}, \{h\}, \{j\}, \{k\}, \{l\}, \{q\}, \{r\}$ .

6. Après réalisation des fusions, on obtient l'automate 10.12.



On conçoit aisément que pour un dictionnaire du français, ce principe de « factorisation » des suffixes conduise à des réductions très sensibles du nombre d'état de l'automate : chaque verbe du français (environ 10 000) possède une cinquantaine de formes différentes ; parmi les verbes, les verbes du premier groupe sont de loin les plus nombreux et ont des suffixes très réguliers qui vont « naturellement » se factoriser.

Automate 10.12 – Le dictionnaire déterminisé et minimisé



**Quatrième partie**

**Références**







# Liste des Algorithmes

4.2	Reconnaissance par un DFA . . . . .	31
-----	-------------------------------------	----





# Chapitre 11

## Liste des automates

4.1	Un automate fini (déterministe) . . . . .	30
4.3	Un automate fini déterministe comptant les $a$ (modulo 3) . . . . .	32
4.4	Un automate fini déterministe équivalent à l'automate 4.1 . . . . .	32
4.5	Un automate fini déterministe pour $ab(a + b)^*$ . . . . .	33
4.6	Un automate partiellement spécifié . . . . .	34
4.7	Un automate non-déterministe . . . . .	37
4.8	Un automate à déterminer . . . . .	38
4.9	Le résultat de la détermination de l'automate 4.8 . . . . .	38
4.10	Un automate difficile à déterminer . . . . .	39
4.11	Un automate potentiellement à déterminer . . . . .	40
4.12	Un automate avec transitions spontanées correspondant à $a^*b^*c^*$ . . . . .	41
4.13	L'automate 4.12 débarrassé de ses transitions spontanées . . . . .	41
4.14	Automate non-déterministe $A$ . . . . .	42
4.15	Automate de Thompson pour $\emptyset$ . . . . .	47
4.16	Automate de Thompson pour $\varepsilon$ . . . . .	47
4.17	Automate de Thompson pour $a$ . . . . .	47
4.18	Automate de Thompson pour $e_1 + e_2$ . . . . .	47
4.19	Automate de Thompson pour $e_1e_2$ . . . . .	48
4.20	Automate de Thompson pour $e_1^*$ . . . . .	48
4.21	Automate de Thompson pour $(a + b)^*b$ . . . . .	48
4.22	Illustration de BMC : élimination de l'état $q_j$ . . . . .	50

---

4.23 Un DFA à minimiser . . . . .	57
4.24 L'automate minimal de $(a + b)a^*ba^*b(a + b)^*$ . . . . .	57
4.25 Un petit dictionnaire . . . . .	58
10.1 Automate d'origine pour $(a^*n \mid na^*)$ . . . . .	126
10.4 Automate sans transition spontanée pour $(a^*n \mid na^*)$ . . . . .	128
10.5 L'automate 10.4, émondé . . . . .	128
10.6 Automate déterministe pour $(a^*n \mid na^*)$ . . . . .	128
10.7 Détermination de l'automate 4.14 . . . . .	129
10.8 Les automates pour $A_1$ (à gauche), $A_2$ (en haut) et leur intersection (au milieu)	130
10.9 L'automate $A_1$ de la question 1 . . . . .	131
10.10 L'automate $A_2$ de la question 1 . . . . .	131
10.11 Le dictionnaire déterminisé . . . . .	132
10.12 Le dictionnaire déterminisé et minimisé . . . . .	133

# Chapitre 12

## Liste des tableaux

2.1	Métamorphose de chien en chameau . . . . .	17
3.1	Identités rationnelles . . . . .	24
3.2	Définition des motifs pour <code>grep</code> . . . . .	26
10.2	$\epsilon$ -fermetures (avant) des états de l'automate 10.1 . . . . .	127
10.3	$\epsilon$ -fermetures arrières des états de l'automate 10.1 . . . . .	127





## Chapitre 13

# Bibliographie

- BARSKY, R. F. (1997). *Noam Chomsky: A Life of Dissent*. MIT Press. Biographie de Noam Chomsky, également disponible en ligne sur <http://cognet.mit.edu/library/books/chomsky/chomsky/>. 9.2
- DOWEK, G. (2007). *Les métamorphoses du calcul. Une étonnante histoire des mathématiques*. Le Pommier. L'histoire du calcul et du raisonnement, des mathématiciens grecs aux progrès récents en preuve automatique. Grand prix de philosophie 2007 de l'Académie française. 9.2
- DOXIADIS, A. et PAPADIMITRIOU, C. H. (2010). *Logicomix. La folle quête de la vérité scientifique absolue*. Vuibert. Bande dessinée romançant les découvertes en logique mathématique du début du 20<sup>e</sup> siècle. 9.2
- HODGES, A. (1992). *Alan Turing. The Enigma*. Random House. Biographie d'Alan Turing, traduit en français sous le nom de *Alan Turing ou l'énigme de l'intelligence*. 9.2
- HOPCROFT, J. E. et ULLMAN, J. D. (1979). *Introduction to automata theory, languages and computation*. Addison-Wesley. 4
- PERRIN, D. (1995). Les débuts de la théorie des automates. *Technique et science informatique*, 14(4). Un historique de l'émergence de la théorie des automates. 9.2
- SAKAROVITCH, J. (2003). *Éléments de théorie des automates*. Vuibert, Paris. 4
- SUDKAMP, T. A. (1997). *Languages and Machines*. Addison-Wesley. 4
- TURING, A. M. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42). L'article introduisant la notion de machine de Turing, un des articles fondateurs de la science informatique. Intéressant à la fois par son contenu et par son aspect historique. 9.2
-





# Chapitre 14

## Index

### Symbols

$\Sigma$ , 11

$\Sigma^+$ , 12

$\Sigma^*$ , 12

$|u|$ , 11

$|u|_a$ , 12

$\varepsilon$ , 11

$\varepsilon$ -NFA, 40

$\varepsilon$ -fermeture

- arrière

- d'un automate, 41

- avant

- d'un automate, 41

- d'un état, 41

### A

Alan Turing, 122

algorithme

- d'élimination des états, 50

- de Brzozowski et McCluskey, 50

- de Glushkov, 48

- de Moore, 57

- de Thompson, 48

Alonzo Church, 118

alphabet, 11

analyse

- lexicale, 9

- syntaxique, 9

Augustus De Morgan, 119

automate

- canonique, 55

- complet, 34

- déterminisé, 37

- fini déterministe, 33

- à transitions spontanées, 40

- émondé, 35

- équivalent, 32

- fini déterministe (complet), 29

- généralisé, 49

- non-déterministe, 36

### B

Backus, John, 117

Brzozowski, Janusz, 117

### C

calcul

- dans un automate, 30

- réussi, 30

Cantor, Georg, 118

Chomsky, Noam, 118

Church, Alonzo, 118

complémentation

- d'automate, 43

concaténation

- d'automates, 45

- de langages, 19

- de mots, 12

congruence

- droite, 20, 54

construction

- des sous-ensembles, 39

### D

David Hilbert, 120

De Morgan, Augustus, 119

destination

- d'une transition, 29

DFA, 29

---

distance

- d'édition, 16
- de Levenshtein, 16
- préfixe, 15

## E

- Edward F. Moore, 121
- Edward J. McCluskey, 121
- état, 29
  - accessible, 35
  - co-accessible, 35
  - s distinguables, 55
  - final, 29
  - s indistinguables, 55
  - initial, 29
  - puits, 33
  - utile, 35
- étiquette
  - d'un calcul, 30
  - d'une transition, 29
- étoile
  - d'automate, 46
- expression rationnelle, 22
  - équivalente, 24

## F

- facteur
  - propre, 14
- fonction
  - de transition, 29

## G

- Georg Cantor, 118
- Glushkov, Victor, 119
- Greibach, Sheila, 119
- Gödel, Kurt, 119

## H

- Hilbert, David, 120

## I

- intersection
  - d'automates, 44

## J

- Janusz Brzozowski, 117
- John Backus, 117
- John von Neumann, 121

## K

- Ken Thompson, 122
- Kleene, Stephen C., 120
- Kurt Gödel, 119

## L

- langage, 12
  - des facteurs, 20
  - des préfixes, 19
  - des suffixes, 20
  - préfixe, 20
  - rationnel, 22
  - reconnaissable, 31
  - reconnu, 30
  - récursif, 13
  - récursivement énumérable, 12
- lemme
  - de l'étoile, 51
  - de pompage
    - rationnel, 51
- Levenshtein, Vladimir, 120
- longueur
  - d'un mot, 11

## M

- McCluskey, Edward J., 121
- monoïde, 12
  - libre, 12
- Moore, Edward F., 121
- morphisme, 20
- mot, 11
  - conjugué, 14
  - s distinguables d'un langage, 53
  - facteur, 13
  - s indistinguables d'un automate, 54
  - s indistinguables d'un langage, 53
  - miroir, 14
  - préfixe, 13
  - primitif, 14
  - suffixe, 13
  - transposé, 14
  - vide, 11

## N

- Naur, Peter, 121
- NFA, 36
- Noam Chomsky, 118

---

## O

ordre

- alphabétique, [15](#)
- bien fondé, [15](#)
- lexicographique, [15](#)
- militaire, [15](#)
- partiel, [14](#)
- radiciel, [15](#)
- total, [15](#)

origine

- d'une transition, [29](#)

## P

palindrome, [14](#)

Peter Naur, [121](#)

produit

- d'automates, [44](#)
- de langages, [19](#)

pumping lemma

- rational, [51](#)

## Q

quotient

- droit
- d'un langage, [20](#)
- d'un mot, [14](#)

## R

relation

- d'ordre, [14](#)
- d'équivalence invariante à droite, [54](#)

## S

semi-groupe, [12](#)

Sheila Greibach, [119](#)

sous-mot, [13](#)

star lemma, [51](#)

Stephen C. Kleene, [120](#)

## T

théorème

- de Kleene, [50](#)

Thompson, Ken, [122](#)

transformation syntaxique, [25](#)

transition, [29](#)

- spontanée, [40](#)

transposition

- d'automate, [45](#)

Turing, Alan, [122](#)

## U

union

- d'automates, [43](#)

## V

Victor Glushkov, [119](#)

Vladimir Levenshtein, [120](#)

von Neumann, John, [121](#)