Théorie des langages

Notes de Cours

François Yvon et Akim Demaille
avec la participation
de Pierre Senellart

24 Juin 2021
(rev. 087c43a)

Avertissement au lecteur

Ces notes documentent le cours de théorie des langages enseigné dans le cadre de la BCI d'informatique. 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, akim@lrde.epita.fr, pierre@senellart.com).

Dans ce polycopié, sont au programme du cours (et de l'examen) :

— Généralités sur les langages (chapitre 2) ; les notions d'ordre, de quotient, de distance ne sont pas spécifiquement au programme et seront réintroduites dans l'examen si elles y sont utiles.

— Langages rationnels, expressions rationnelles, automates (chapitres 3 et 4, cf. aussi TD 1, TP 1).

— Introduction aux grammaires génératives et à la hiérarchie de Chomsky (chapitre 5).

— Grammaires et langages hors-contexte : définitions, arbres de dérivation, lemme de pompage pour les grammaires hors-contexte, problématique de l'analyse de grammaire (chapitres 5 et 6, cf. aussi TD 2, TP 2).

— Notions de calculabilité (chapitre 10) ; la définition formelle des modèles de calcul (machines de Turing) n'est pas au programme.

Le chapitre 7 est partiellement couvert en cours mais n'est pas au programme de l'examen. Les chapitres 8 et 9 ne sont pas au programme de l'examen ; certains sujets d'annales font référence à ces notions (analyseurs LL ou LR) car le contenu du cours a changé depuis. Le chapitre 11 (Compléments historiques) n'est pas au programme mais pourra être utilisé pendant le cours.

Chapitre 1

Table des matières

1 Table des matières **3**

I Notes de Cours **9**

2 Mots, Langages **11**

2.1 Quelques langages « réels » 11

2.1.1 La compilation 11

2.1.2 Bio-informatique 12

2.1.3 Les langues « naturelles » 13

2.2 Terminologie 13

2.2.1 Bases 13

2.2.2 Quelques notions de calculabilité 14

2.3 Opérations sur les mots 15

2.3.1 Facteurs et sous-mots 15

2.3.2 Quotient de mots 16

2.3.3 Ordres sur les mots 16

2.3.4 Distances entre mots 17

2.3.5 Quelques résultats combinatoires élémentaires 19

2.4 Opérations sur les langages 20

2.4.1 Opérations ensemblistes 20

2.4.2 Concaténation, Étoile de Kleene 21

2.4.3 Plus d'opérations dans $\mathcal{P}(\Sigma^\star)$ . . . 21
2.4.4 Morphismes . . . 22

3 Langages et expressions rationnels **23**
3.1 Rationalité . . . 24
3.1.1 Langages rationnels . . . 24
3.1.2 Expressions rationnelles . . . 24
3.1.3 Équivalence et réductions . . . 26
3.2 Extensions notationnelles . . . 27

4 Automates finis **31**
4.1 Automates finis . . . 31
4.1.1 Bases . . . 31
4.1.2 Spécification partielle . . . 35
4.1.3 États utiles . . . 37
4.1.4 Automates non-déterministes . . . 37
4.1.5 Transitions spontanées . . . 42
4.2 Reconnaissables . . . 45
4.2.1 Opérations sur les reconnaissables . . . 45
4.2.2 Reconnaissables et rationnels . . . 48
4.3 Quelques propriétés des langages reconnaissables . . . 53
4.3.1 Lemme de pompage . . . 53
4.3.2 Quelques conséquences . . . 54
4.4 L'automate canonique . . . 55
4.4.1 Une nouvelle caractérisation des reconnaissables . . . 55
4.4.2 Automate canonique . . . 57
4.4.3 Minimisation . . . 58

5 Grammaires syntagmatiques **63**
5.1 Grammaires . . . 63
5.2 La hiérarchie de Chomsky . . . 65
5.2.1 Grammaires de type 0 . . . 65
5.2.2 Grammaires contextuelles (type 1) . . . 66

5.2.3 Grammaires hors-contexte (type 2) . . . 69
5.2.4 Grammaires régulières (type 3) . . . 70
5.2.5 Grammaires à choix finis (type 4) . . . 73
5.2.6 Les productions ε . . . 73
5.2.7 Conclusion . . . 74

6 Langages et grammaires hors-contexte **77**
6.1 Quelques exemples . . . 77
6.1.1 La grammaire des déjeuners du dimanche . . . 77
6.1.2 Une grammaire pour le shell . . . 79
6.2 Dérivations . . . 81
6.2.1 Dérivation gauche . . . 82
6.2.2 Arbre de dérivation . . . 83
6.2.3 Ambiguïté . . . 84
6.2.4 Équivalence . . . 86
6.3 Les langages hors-contexte . . . 86
6.3.1 Le lemme de pompage . . . 86
6.3.2 Opérations sur les langages hors-contexte . . . 87
6.3.3 Problèmes décidables et indécidables . . . 88

7 Introduction au parsage de grammaires hors-contexte **91**
7.1 Graphe de recherche . . . 92
7.2 Reconnaissance ascendante . . . 92
7.3 Reconnaissance descendante . . . 95
7.4 Conclusion provisoire . . . 97

8 Introduction aux analyseurs déterministes **99**
8.1 Analyseurs LL . . . 100
8.1.1 Une intuition simple . . . 100
8.1.2 Grammaires LL(1) . . . 103
8.1.3 NULL, FIRST et FOLLOW . . . 103
8.1.4 La table de prédiction . . . 107
8.1.5 Analyseurs LL(1) . . . 109

8.1.6 LL(1)-isation . . . 109
8.1.7 Quelques compléments . . . 111
8.1.8 Un exemple complet commenté . . . 112
8.2 Analyseurs LR . . . 113
8.2.1 Concepts . . . 113
8.2.2 Analyseurs LR(0) . . . 115
8.2.3 Analyseurs SLR(1), LR(1), LR(k)... . . . 122
8.2.4 Compléments . . . 128

9 Normalisation des grammaires CF **129**
9.1 Simplification des grammaires CF . . . 129
9.1.1 Quelques préliminaires . . . 129
9.1.2 Non-terminaux inutiles . . . 130
9.1.3 Cycles et productions non-génératives . . . 132
9.1.4 Productions ε . . . 134
9.1.5 Élimination des récursions gauches directes . . . 135
9.2 Formes normales . . . 136
9.2.1 Forme normale de Chomsky . . . 137
9.2.2 Forme normale de Greibach . . . 138

10 Notions de calculabilité **143**
10.1 Décidabilité et semi-décidabilité . . . 143
10.2 Langages non semi-décidables . . . 144
10.3 Langages non décidables . . . 146
10.4 Modèles de calculs . . . 148

II Annexes **151**

11 Compléments historiques **153**
11.1 Brèves biographies . . . 153
11.2 Pour aller plus loin . . . 158

12 Correction des exercices **161**

12.1 Correction de l'exercice 4.5 161
12.2 Correction de l'exercice 4.10 162
12.3 Correction de l'exercice 4.14 162
12.4 Correction de l'exercice 4.15 165
12.5 Correction de l'exercice 4.19 165
12.6 Correction de l'exercice 4.20 167
12.7 Correction de l'exercice 4.35 167
12.8 Correction de l'exercice 5.21 169

III Références **171**

13 Liste des automates **175**

14 Liste des grammaires **177**

15 Liste des tableaux **179**

16 Table des figures **181**

17 Bibliographie **183**

18 Index **185**

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 $(\{, \}, \ldots)$, 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 langages 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 [1] ;

— 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 Σ, que l'on appelle l'*alphabet*, on appelle *mot* toute suite finie (éventuellement vide) d'éléments de Σ. Par convention, le *mot vide* est noté ε ; certains auteurs le notent 1, voire 1_Σ. 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 Σ (resp. de tous les mots non-vides) est noté $\Sigma^\star$ (resp. Σ^+). Un *langage* sur Σ est un sous-ensemble de $\Sigma^\star$.

L'opération de *concaténation de deux mots* u et v de $\Sigma^\star$ 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 $\Sigma^\star$; elle est associative, mais pas commutative (sauf dans le cas dégénéré où Σ ne contient qu'un seul symbole). ε est l'élément neutre pour la concaténation : $u\varepsilon = \varepsilon u = u$; ceci justifie la notation 1 ou encore 1_Σ. 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 $x = uy^{-1}$.

$\Sigma^\star$, 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 Σ : tout mot u se décompose de manière *unique* comme concaténation de symboles de Σ.

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

— $\Sigma^\star = \{\varepsilon, a, b, c, aa, ab, ac, ba, \ldots\}$, soit l'ensemble de tous les mots composés de lettres de Σ ;

— $\{\varepsilon, a, b, c\}$, soit tous les mots de longueur strictement inférieure à 2 ;

— $\{ab, aab, abb, acb, aaab, aabb, \ldots\}$, soit tous les mots qui commencent par un a et finissent par un b ;

— $\{\varepsilon, ab, aabb, aaabbb, \ldots\}$, soit tous les mots commençant par n a suivis d'autant de b. Ce langage est noté $\{a^n b^n \mid n \geq 0\}$;

— $\{\varepsilon, abc, aabbcc, aaabbbccc, \ldots\}$, 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 $\{a^n b^n c^n \mid n \geq 0\}$.

— $\{aa, aaa, aaaaa, \ldots\}$, tous les mots composés d'un nombre *premier* de a.

Il existe un nombre dénombrable de mots dans $\Sigma^\star$, mais le nombre de langages dans $\Sigma^\star$ 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 $\Sigma^\star$ 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 $\Sigma^\star$ (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 $\Sigma^\star$ 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 $\Sigma^\star$, alors $v = v_1 v_2 \dots v_n$ est *sous-mot*[2] 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 [3] : 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 [4] 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 :*

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

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 ${}_{u\backslash}v$.

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 $\Sigma^\star$: 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 $\Sigma^\star$, à la condition de disposer d'un ordre total $\leq$ sur Σ.

Définition 2.4 (Ordre lexicographique)**.** *L'*ordre lexicographique *sur* $\Sigma^\star$ *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 $\{a,b\}^\star$: $\{\varepsilon, a, aa, ab, aaa, \ldots\}$.

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* $\Sigma^\star$ *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 $wuw' \leq_a wvw'$, 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 dictionaires 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 $\mathrm{plpc}(u,v)$ le plus long préfixe commun à u et à v, alors la fonction $d_p(u,v)$ définie par :

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

définit une distance sur $\Sigma^\star$, 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 plpc(u,v) et à 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 Σ^+ comme une *fonction* de l'intervalle $I = [1 \dots |u|]$ vers Σ, qui à chaque entier i associe le i^{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 $plsmc(u,v)$ 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 :

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

soit encore :

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

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

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

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 $plsmc(u,w)$. On en déduit donc que : $|I \cap J| \leq |plsmc(u,w)|$. Puisque, par ailleurs, on a $|I \cup J| \leq |v|$, on peut conclure que :

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

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 $\Sigma^\star$ 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
chien	supprimer *i*
chen	insérer *a*
chaen	insérer *m*
chamen	supprimer *n*
chame	insérer *a*
chamea	insérer *u*
chameau	

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[5], 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.

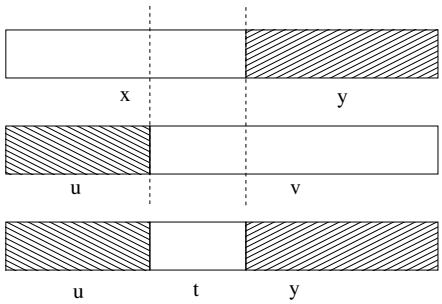
2.3.5 Quelques résultats combinatoires élémentaires

La propriété suivante est trivialement respectée :

Lemme 2.6.

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

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 $\exists u, v \in \Sigma^\star$ 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 $y = uv(uv)^k u = (uv)^{k+1} u$. □

Théorème 2.8. *Si $xy = yx$, avec $x \neq \varepsilon, y \neq \varepsilon$, alors $\exists u \in \Sigma^\star$ 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. □

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 $\Sigma^\star$) se définissent-elles pour L, L_1 et L_2 des langages de $\Sigma^\star$ par :

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

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) :

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

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 $\{u \in \Sigma^\star \mid \exists v \in L, u = v^n\}$.

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

$$L^\star \quad = \quad \bigcup_{i \geq 0} L^i$$

$L^\star$ 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 Σ est un alphabet, $\Sigma^\star$, tel que défini précédemment, représente [6] l'ensemble des séquences finies que l'on peut construire en concaténant des symboles de Σ. Remarquons que, par définition, $\emptyset^\star$ n'est pas vide, puisqu'il (ne) contient (que) ε.

On définit également

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

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

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 : $\{0,1\}^\star\{111\}\{0,1\}^\star$, la notation $\{0,1\}^\star$ 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 $\mathcal{P}(\Sigma^\star)$

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

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

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

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

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 $\Sigma^\star$, on définit le* langage des suffixes *de L, noté* Suff(L) *par :*

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

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

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

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 :*

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

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

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

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

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 $\forall u, L_{/w_1 u} = L_{/w_2 u}$. 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 ϕ 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 $(\Sigma^\star, .)$ dans $(N, +)$: 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 [1], 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 $\mathcal{P}(\Sigma^\star)$ 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 Σ un alphabet. Les* langages rationnels *sur Σ 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 ${L_1}^\star$ 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 : $(\{0\} \cup \{1\})^\star\{1\}\{1\}\{1\}(\{0\} \cup \{1\})^\star$, exprimant que les mots de ce langage sont construits en prenant deux mots quelconques de $\Sigma^\star$ 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 Σ un alphabet. Les* expressions rationnelles *(RE) sur Σ sont définies inductivement par :*

(i) ε *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 $({e_1}^\star)$ 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 :

— $r, e, d, é$, sont des RE (par (ii))

— (re) et $(dé)$ sont des RE (par (iii))

— $((((fa)i)r)e)$ est une RE (par (ii), puis (iii))

— $((re) + (dé))$ est une RE (par (iii))

— $((((re) + (dé)))^\star)$ est une RE (par (iii))

— $(((((re) + (dé)))^\star)((((fa)i)r)e))$ 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) ε 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) $(e_1 e_2)$ dénote la concaténation des langages dénotés par e_1 et par e_2
(iii.3) $(e^\star)$ 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, $aa^\star + b^\star$ s'interprète-t-il comme $((a(a^\star)) + (b^\star))$, et $(((((re) + (dé)))^\star)((((fa)i)r)e))$ peut s'écrire $(re + dé)^\star faire$.

Revenons à la formule précédente : $(re + dé)^\star faire$ dénote l'ensemble des mots formés en itérant à volonté un des deux préfixes *re* ou *dé*, 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 $\{a, b, \ldots, z\}$ 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 : $a^\star(a^\star ba^\star ba^\star)^\star$ et $a^\star(ba^\star ba^\star)^\star$ 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.

$$
\begin{array}{rcl@{\qquad}rcl}
\emptyset e & \equiv & \emptyset & \varepsilon e & \equiv & e \\
e\emptyset & \equiv & \emptyset & e\varepsilon & \equiv & e \\
\emptyset^\star & \equiv & \varepsilon & \varepsilon^\star & \equiv & \varepsilon \\
e + f & \equiv & f + e & e + \emptyset & \equiv & e \\
e + e & \equiv & e & (e^\star)^\star & \equiv & e^\star \\
e(f + g) & \equiv & ef + eg & (e + f)g & \equiv & eg + fg \\
(ef)^\star e & \equiv & e(fe)^\star & & & \\
(e + f)^\star & \equiv & e^\star(e + f)^\star & (e + f)^\star & \equiv & (e^\star + f)^\star \\
(e + f)^\star & \equiv & (e^\star f^\star)^\star & (e + f)^\star & \equiv & (e^\star f)^\star e^\star
\end{array}
$$

TABLE 3.1 – Identités rationnelles

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

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

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


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 'chaine' mon-texte
```

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

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*ine' mon-texte
```

2. Par *transformation syntaxique* on entend une simple récriture des mots (ici les expressions rationnelles elles-mêmes) sans devoir faire appel à ce qu'ils représentent. Par exemple les règles de récriture $x + 0 \rightsquigarrow x$ et $0 + x \rightsquigarrow x$ expriment *syntaxiquement* 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 $cha^{\star}ine$ 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 $\Sigma^{\star}(e)\Sigma^{\star}$; 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]'`	$a+b+c+\cdots+z$	utilise l'ordre des caractères ASCII
`'[^abc]'`	$\Sigma\backslash\{a,b,c\}$	n'inclut pas le symbole de fin de ligne `\n`
`'.'`	Σ	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}'`	$\varepsilon + e + e^2 + \cdots + e^m$	
`'e{n,m}'`	$e^n + e^{n+1} + \cdots + e^m$	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 [3] :

```
> 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 ε 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-zéèêîôûç]{3,}([îâû]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 `regexp` 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 $fo(o^\star)$.

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

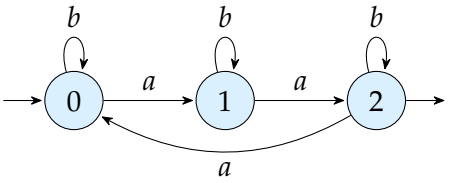
Dans cette section, nous introduisons le modèle le plus simple d'automate fini : l'automate déterministe complet. Ce modèle nous permet de définir les notions de calcul et de langage associé à un automate. Nous terminons cette section en définissant la notion d'équivalence entre automates, ainsi que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

- Σ *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
- δ *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 Σ. 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 :

δ	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 Σ et v dans $\Sigma^\star$: $(q, av) \vdash_A (\delta(q,a), v)$ pour noter une étape de calcul utilisant la transition $(q, a, \delta(q,a))$. La clôture réflexive et transitive de $\vdash_A$ se note $\vdash_A^\star$; $(q, uv) \vdash_A^\star (p, v)$ 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), $(q, u) \vdash^\star (q, u)$. Avec ces notations, on a :

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

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 = u_1 ... u_n est le mot à reconnaître.
// A = (Σ, Q, q_0, δ, F) est le DFA.
q := q_0;
for i := 1 to n do
  | q := δ(q, u_i)
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.*

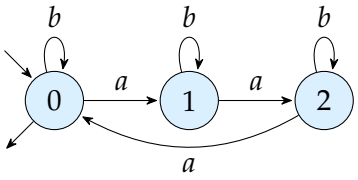
Dans Vcsn, un automate peut-être vu comme une fonction qui calcule un booléen représentant l'appartenance du mot au langage de l'automate :

```
>>> 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 δ en une fonction $\delta^\star$ de $Q \times \Sigma^\star \to Q$ par :

— $\delta^\star(q, \varepsilon) = q$

— $\delta^\star(q, au) = \delta^\star(\delta(q, a), u)$

On notera que, puisque δ est une fonction totale, $\delta^\star$ est également une fonction totale : l'image d'un mot quelconque de $\Sigma^\star$ par $\delta^\star$ 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 :

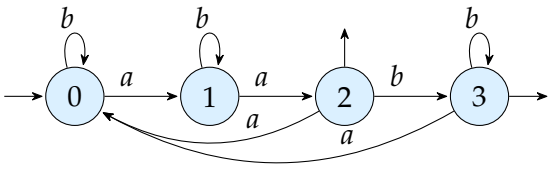
$$L(A) = \{u \in \Sigma^\star \mid \delta^\star(q_0, u) \in F\}$$

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.*

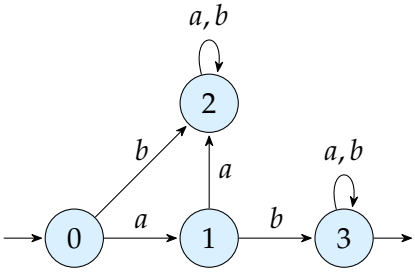
Ainsi, par exemple, l'automate 4.4 est-il équivalent à l'automate 4.1 : tous deux reconnaissent le langage de tous les mots qui contiennent un nombre de a congru à 2 modulo 3.



Automate 4.4 – Un automate fini déterministe équivalent à l'automate 4.1

Nous l'avons noté plus haut, $\delta^\star$ est définie pour tout mot de $\Sigma^\star$. 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 $\{ab\}\{a,b\}^\star$. 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 $ab(a+b)^\star$

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 :*

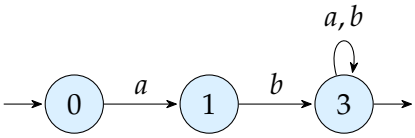
- Σ *est un ensemble fini de symboles (l'alphabet)*
- Q *est un ensemble fini d'états*
- $q_0 \in Q$ *est l'état initial*
- $F \subset Q$ *sont les états finaux*
- δ *est une fonction* partielle *de* $Q \times \Sigma$ *dans* Q

La différence avec la définition 4.1 est que δ 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 $u \in L(A)$, $\delta^\star(q_0, u)$ 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 $\delta^\star(q_0, u)$ 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 $\delta(\delta^\star(q_0, v), a)$ 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 $\Sigma^\star$ est* local *s'il existe I et F deux sous-ensembles de Σ et T un sous-ensemble de Σ^2 (l'ensemble des mots sur Σ de longueur 2) tels que $L = (I\Sigma^\star \cap \Sigma^\star F) \setminus (\Sigma^\star T \Sigma^\star)$. 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 $aa^\star$ est un langage local ; que le langage L_2 dénoté par $ab(ab)^\star$ est un langage local. Dans les deux cas, on prendra $\Sigma = \{a, b\}$.*
2. *Montrer que le langage L_3 dénoté par $aa^\star + ab(ab)^\star$ 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 ϕ de $\Sigma_1^\star$ vers $\Sigma_2^\star$ telle que (i) $\phi(\varepsilon) = \varepsilon$ et (ii) si u et v sont deux mots de $\Sigma_1^\star$, alors $\phi(uv) = \phi(u)\phi(v)$. Un morphisme lettre-à-lettre est induit par une injection ϕ de Σ_1 dans Σ_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 $\Sigma^\star$ tel que $\delta^\star(q_0, u) = q$. 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 $\Sigma^\star$ tel que $\delta^\star(q, u) \in F$. 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 δ' de δ 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'. □

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 :*

- Σ *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 δ comme une fonction de $Q \times \Sigma$ dans 2^Q : l'image par δ 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 $\delta^\star$ de $Q \times \Sigma^\star$ dans 2^Q par :

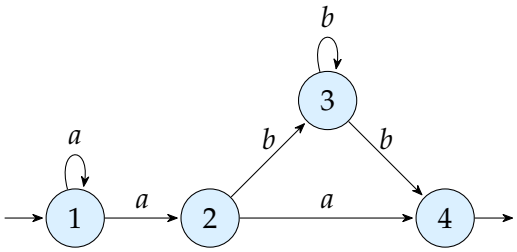
- $\delta^\star(q, \varepsilon) = \{q\}$
- $\delta^\star(q, au) = \bigcup_{r \in \delta(q,a)} \delta^\star(r, u)$

L'automate 4.7 illustre ces notions.

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

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

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.*

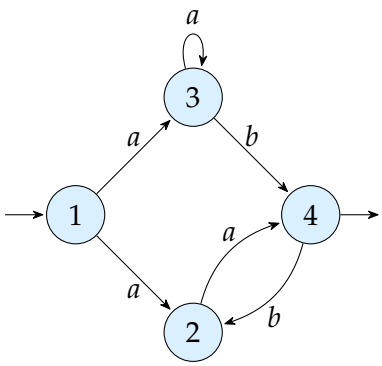
On pose $A = (\Sigma, Q, I, F, \delta)$ et on considère A' défini par $A' = (\Sigma, 2^Q, I, F', \delta')$ avec :

— $F' = \{G \subset Q \mid F \cap G \neq \emptyset\}$.

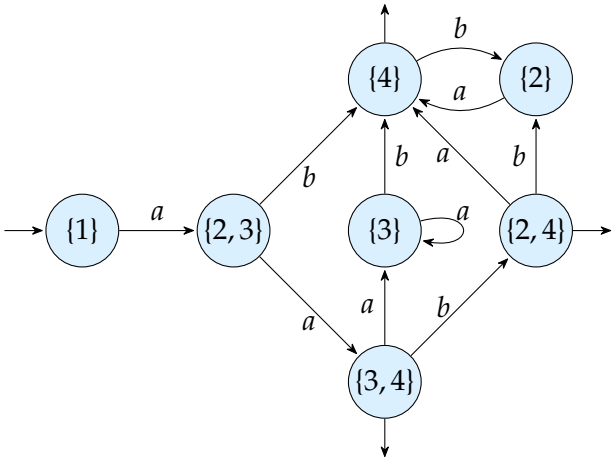
— $\forall G \subset Q, \forall a \in \Sigma, \delta'(G,a) = \bigcup_{q \in G} \delta(q,a)$.

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 déterminiser



Automate 4.9 – Le résultat de la déterminisation 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 δ' d'un couple (H, a) est uniquement définie.

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

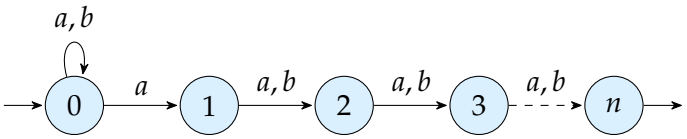
A', soit formellement que :

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

Opérons une récurrence sur la longueur de u : si u est égal à ε 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 $(q_0, va) \vdash_A^\star (p, a) \vdash_A (q, \varepsilon)$ un calcul dans A : par l'hypothèse de récurrence, il existe un calcul dans A' tel que : $(\{I\}, v) \vdash_{A'}^\star (G, \varepsilon)$, avec $p \in G$. Ceci implique, en vertu de la définition même de δ', que q appartient à $H = \delta'(G, a)$, et donc que $(\{I\}, u = va) \vdash_{A'}^\star (H, \varepsilon)$, avec $q \in H$. Inversement, soit $(\{I\}, u = va) \vdash_{A'}^\star (G, a) \vdash_{A'} (H, \varepsilon)$: pour tout p dans G il existe un calcul dans A tel que $(q_0, v) \vdash_A^\star (p, \varepsilon)$. 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 $(q_0, u = va) \vdash_A^\star (q, \varepsilon)$, avec $q \in H$. On déduit alors que l'on a $(q_0, u = va) \vdash_A^\star (q, \varepsilon)$, avec $q \in F$ si et seulement si $(\{I\}, u) \vdash_{A'}^\star (G, \varepsilon)$ avec $q \in G$, donc avec $F \cap G \neq \emptyset$, soit encore $G \in F'$. Il s'ensuit directement que $L(A) = L(A')$. □

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 déterminiser

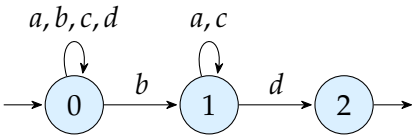
Dans la mesure où ils n'apportent aucun gain en expressivité, il est permis de se demander à quoi servent les automates non-déterministes. Au moins à deux choses : ils sont plus faciles à construire à partir d'autres représentations des langages et sont donc utiles pour certaines preuves (voir plus loin) 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 déterminiser 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 déterminiser

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 *(ε-NFA, nondeterministic finite automaton with ε moves) est défini par un quintuplet $A = (\Sigma, Q, I, F, \delta)$, où :*

— *Σ 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

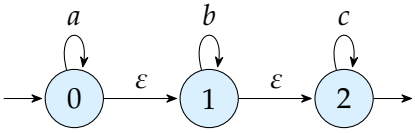
— *$\delta \subset Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ est la relation de transition.*

Encore une fois, on aurait pu ne conserver qu'un seul état initial, et définir δ 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 δ permet d'étiqueter les transitions par ε. 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 ε-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 ε-NFA A est, comme précédemment, défini par

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

L'automate 4.12 est un exemple d'ε-NFA.



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

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

Définition 4.12 (ε-fermeture d'un état)**.** *Soit q un état de Q. On appelle* ε-fermeture *(en anglais closure) de q l'ensemble ε-closure$(q) = \{p \in Q \mid (q, \varepsilon) \vdash_A^\star (p, \varepsilon)\}$. Par construction, $q \in \varepsilon$-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 $\{0, 1, 2\}$.

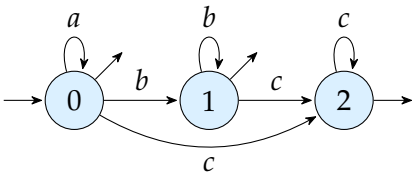
Théorème 4.13. *Pour tout ε-NFA A, il existe un NFA A' tel que $L(A) = L(A')$.*

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

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

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

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'*ε-fermeture arrière d'un automate* (on pourrait dire « amont ») : la fonction δ « 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'*ε-fermeture avant d'un ε-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 ε-NFA, il faut spécifier que le mot vide est une étiquette valide. Le contexte est '`lan_char, b`' au lieu de '`lal_char, b`'. Ainsi l'automate 4.12 s'écrit (notez le '`r`' au début de la chaîne) :

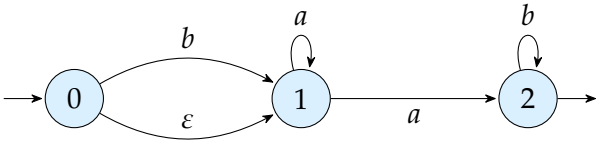
```
>>> a = vcsn.automaton(r'''
context = "lan_char, b"
$ -> 0
0 -> 0 a
0 -> 1 \e
1 -> 1 b
1 -> 2 \e
2 -> 2 c
2 -> $
''')
```

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 : $a^\star n + na^\star$.*

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'ε-fermeture de chacun des états de l'automate construit à la question 1.*
3. *Déduisez-en un automate sans ε-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 spontanées.

Dans cette section, nous montrons dans un premier temps que l'ensemble des langages reconnaissables 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 complémentation.*

Démonstration. Soit L un reconnaissable et A un DFA *complet* reconnaissant L. On construit alors un automate A' pour $\overline{L}$ 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'. □

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 δ 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 $Q^1 \times F^2$). □

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 : $F^1 \times F^2$. □

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 finals les états non-finaux de A, et en rendant non-final les états finals 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 ε 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 ε.

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

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 lieux 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. □

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 δ 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. □

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

Démonstration. La construction de A' reconnaissant $L^\star$ 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'. □

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

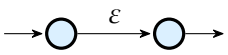
— un unique état final q_F sans transition sortante

— exactement deux fois plus d'états que de symboles dans l'expression rationnelle (sans compter les parenthèses ni la concaténation, implicite).

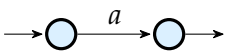
Cette dernière propriété donne un moyen simple de contrôler le résultat en contrepartie d'automates parfois plus lourds que nécessaire.

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), ε (automate 4.16) et les symboles de Σ (automate 4.17).

Automate 4.15 – Automate de Thompson pour $\emptyset$

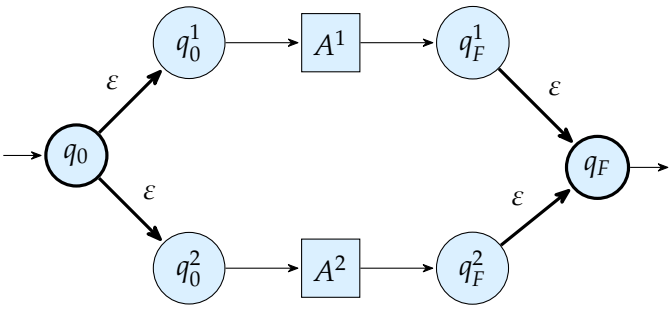


Automate 4.16 – Automate de Thompson pour ε



Automate 4.17 – Automate de Thompson pour a

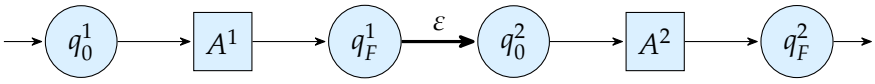
À partir de ces automates élémentaires, nous allons construire de manière itérative des automates pour des expressions rationnelles plus complexes. Si A_1 et A_2 sont les automates de Thompson de e_1 et e_2, alors l'automate 4.18 reconnaît le langage dénoté par l'expression $e_1 + e_2$.



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

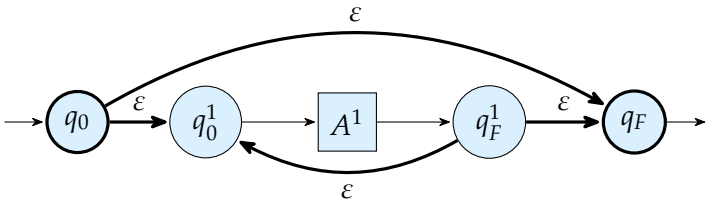
L'union correspond donc à une mise en parallèle de A_1 et de A_2 : un calcul dans cette machine est réussi si et seulement s'il est réussi dans l'une des deux machines A_1 ou A_2.

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 $e_1 e_2$

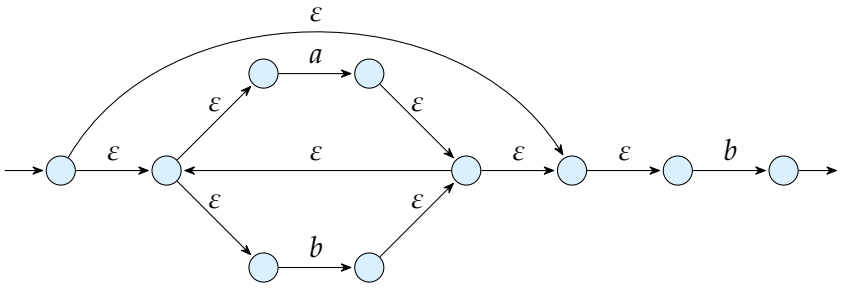
La machine réalisant l'étoile est, comme précédemment, construite en rajoutant une possibilité de boucler depuis l'état final vers l'état initial de la machine, ainsi qu'un arc permettant de reconnaître ε, comme sur l'automate 4.20.



Automate 4.20 – Automate de Thompson pour ${e_1}^\star$

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 $(a + b)^\star b$

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 ε (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 $\Sigma^\star$. 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 originel.

Pour se simplifier la tâche commençons par introduire deux nouveaux états, q_I et q_F, qui joueront le rôle d'uniques 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 $e_{jj} = \varepsilon$.

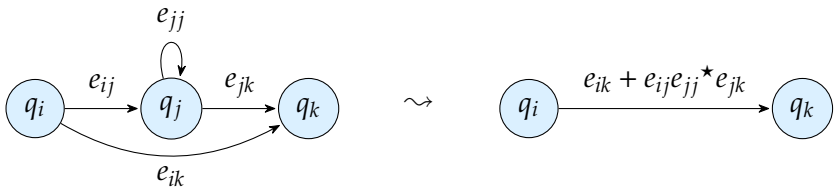
La procédure de suppression de q_j comprend alors les étapes suivantes :

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

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 $q_i \to q_k$, 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^iw 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 : $L(uv^\star w) \subset L$ —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 $\delta^\star(q_0, u) = q$ et v le facteur tel que $\delta^\star(q, v) = q$, 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 $q \dots q$. □

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 $\{a^n b^n \mid n \in \mathbb{N}\} \cup \Sigma^\star\{ba\}\Sigma^\star$.

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

pompage 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^iw 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 $(q_0, u) \vdash_A^\star (q, \varepsilon)$ 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 $2k$. □

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. □

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 $\Sigma^\star$. Deux mots u et v sont dits* indistinguables dans L *si pour tout $w \in \Sigma^\star$, soit uw et vw sont tous deux dans L, soit uw et vw sont tous deux dans $\overline{L}$. 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 $w \in \Sigma^\star$ 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 $L = a(a + b)(bb)^\star$. 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 indistingables d'un automate). *Soit $A = (\Sigma, Q, q_0, F, \delta)$ un automate fini* déterministe. *Deux mots de u et v sont dits* indistinguables dans *A si et seulement si $\delta^\star(q_0, u) = \delta^\star(q_0, v)$. 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 $\delta^\star(q, w)$ 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 fausse, 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 $\Sigma^\star$ 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 Σ, les trois assertions suivantes sont équivalentes :*

(i) *L est un langage rationnel*

(ii) *il existe une relation d'équivalence $\equiv$ sur $\Sigma^\star$, 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 $\{u \in \Sigma^\star \mid \delta^\star(q_0, u) \in F\}$, 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 $\Sigma^\star$ $uw \equiv vw$. 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 ε

— $F = \{[u]_L, u \in L\}$

— $\delta([u]_L, a) = [ua]_L$. Cette définition de δ 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 $(q_0, u) \vdash_A^\star [u]_L$. 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 $(q_0, ua) \vdash_A^\star (p, a) \vdash_A (q, \varepsilon)$. 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. □

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, $L = \{u \in \Sigma^\star \mid \exists a \in \Sigma, i \in \mathbb{N} \text{ tq. } u = a^{2^i}\}$ 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. □

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 $\equiv_\nu$ sur les états de Q. L'ensemble des classes d'équivalence $[q]_\nu$ est notée Q_ν. Pour un automate fini déterministe $A = (\Sigma, Q, q_0, F, \delta)$, on définit l'automate fini A_ν par : $A_\nu = (\Sigma, Q_\nu, [q_0]_\nu, F_\nu, \delta_\nu)$, avec : $\delta_\nu([q]_\nu, a) = [\delta(q, a)]_\nu$; et $F_\nu = [q]_\nu$, avec q dans F. La fonction δ_ν est correctement définie en ce sens que si p et q sont indistinguables, alors nécessairement $\delta(q, a) \equiv_\nu \delta(p, a)$.

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

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

Notons d'abord que ϕ est une application : si u et v de $\Sigma^\star$ 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 ϕ ne dépend pas d'un choix particulier de u.

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

$$\begin{aligned} \phi([q]_\nu) = \phi([p]_\nu) &\Leftrightarrow \exists u, v \in \Sigma^\star, \delta^\star(q_0, u) = q, \delta^\star(q_0, u) = p, \text{ et } u \equiv_L v && (4.1)\\ &\Leftrightarrow \delta^\star(q_0, u) \equiv_\nu \delta^\star(q_0, u) && (4.2)\\ &\Leftrightarrow [q]_\nu = [p]_\nu && (4.3) \end{aligned}$$

Montrons enfin que les calculs dans A_ν sont en bijection par ϕ avec les calculs de A_L. On a en effet :

- $\phi([q_0]_\nu) = [\varepsilon]_L$, car $\delta^\star(q_0, \varepsilon) = q_0 \in [q_0]_\nu$
- $\phi(\delta_\nu([q]_\nu, a) = \delta_L(\phi([q]_\nu), a)$ car soit u tel que $\delta^\star(q_0, u) \in [q]_\nu$, alors : (i) $\delta(\delta^\star(q_0, u), a) \in \delta_\nu([q]_\nu, a)$ (cf. la définition de δ_ν) et $[ua]_L = \phi(\delta_\nu([q]_\nu, a)) = \delta_L([u], a)$ (cf. la définition de δ_L), ce qu'il fallait prouver.
- si $[q]_\nu$ est final dans A_ν, alors il existe u tel que $\delta^\star(q_0, u)$ 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_ν est isomorphe (par ϕ) 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 $\equiv_\nu$, de manière à dériver l'automate A_ν (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 Π_0 qui distingue simplement états finaux et non-finaux. Cet algorithme se glose comme suit :

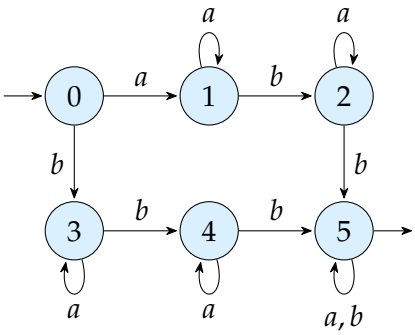
— Initialiser avec deux classes d'équivalence : F et $Q \setminus F$

— Itérer jusqu'à stabilisation :

  — pour toute paire d'états q et p dans la même classe de la partition Π_k, s'il existe $a \in \Sigma$ tel que $\delta(q,a)$ et $\delta(p,a)$ ne sont pas dans la même classe pour Π_k, alors ils sont dans deux classes différentes de Π_{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*. Implantée 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 $\equiv_\nu$ se déroulent alors comme suit :

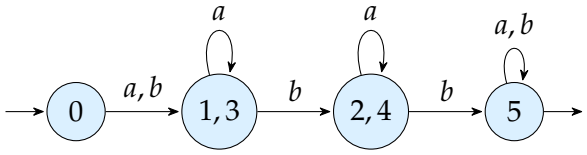
— $\Pi_0 = \{\{0,1,2,3,4\},\{5\}\}$ (car 5 est le seul état final)

— $\Pi_1 = \{\{0,1,3\},\{2,4\},\{5\}\}$ (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.

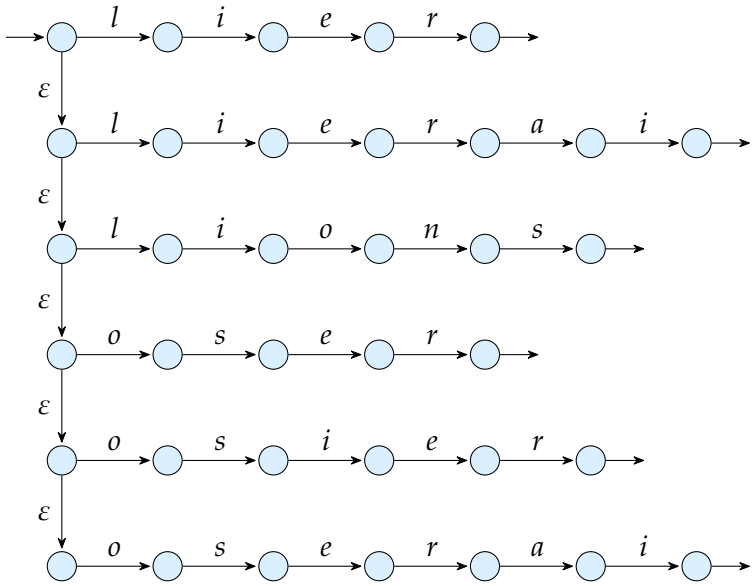


Automate 4.24 – L'automate minimal de $(a+b)a^\star ba^\star b(a+b)^\star$

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 $L = \{lier, lierai, lions, oser, osier, oserai\}$*, 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* ε*, puis déterminisation de l'automate sans transition* ε*.*
2. *On dit que deux états p et q d'un automate déterministe sont* indistinguables *si pour tout mot u de* Σ^**,* $\delta^*(p,u) \in F \Leftrightarrow \delta^*(q,u) \in F$*.*
   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 utilisation 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.*

Chapitre 5

Grammaires syntagmatiques

Dans cette partie, nous présentons de manière générale les grammaires syntagmatiques, ainsi que les principaux concepts afférents. Nous montrons en particulier qu'au-delà de la classe des langages rationnels, il existe bien d'autres familles de langages, qui valent également la peine d'être explorées.

5.1 Grammaires

Définition 5.1 (Grammaire)**.** *Une* grammaire syntagmatique *G est définie par un quadruplet (N, Σ, P, S), où N, Σ et P désignent respectivement des ensembles de* non-terminaux *(ou* variables*), de* terminaux*, et de* productions *(ou règles de production). N et Σ sont des alphabets disjoints. Les productions sont des éléments de $(N \cup \Sigma)^\star \times (N \cup \Sigma)^\star$, que l'on note sous la forme $\alpha \to \beta$. S est un élément distingué de N, appelé le* symbole initial *ou encore l'*axiome *de la grammaire.*

La terminologie de langue anglaise équivalente à grammaire syntagmatique est *Phrase Structure Grammar*, plutôt utilisée par les linguistes, qui connaissent bien d'autres sortes de grammaires. Les informaticiens disent plus simplement *rules* (pour eux, il n'y a pas d'ambiguïté sur le terme !).

Il est fréquent d'utiliser T pour désigner Σ, l'ensemble des terminaux. Il est également utile d'introduire $V = T \cup N$, le *vocabulaire* de la grammaire. Dans la suite, nous utiliserons les conventions suivantes pour noter les éléments de la grammaire : les non-terminaux seront notées par des symboles en majuscule latine ; les terminaux par des symboles en minuscules ; les mots de $V^\star = (T \cup N)^\star$ par des lettres minuscules grecques.

Illustrons ces premières définitions en examinant la grammaire G_1 (grammaire 5.1). Cette grammaire contient une seule variable, S, qui est également l'axiome ; deux éléments terminaux a et b, et deux règles de production, p_1 et p_2.

Si $\alpha \to \beta$ est une production d'une grammaire G, on dit que α est la *partie gauche* et β la *partie droite* de la production.

L'unique opération autorisée, dans les grammaires syntagmatiques, est la récriture d'une séquence de symboles par application d'une production. Formellement,

$$p_1 = S \rightarrow aSb$$
$$p_2 = S \rightarrow ab$$

Grammaire 5.1 – G_1, une grammaire pour $a^n b^n$

Définition 5.2 (Dérivation immédiate)**.** *On définit la relation* $\underset{G}{\Rightarrow}$ *(lire :* dérive immédiatement*) sur l'ensemble* $V^\star \times V^\star$ *par* $\gamma\alpha\delta \underset{G}{\Rightarrow} \gamma\beta\delta$ *si et seulement si* $\alpha \rightarrow \beta$ *est une production de G.*

La notion de dérivation immédiate se généralise à la dérivation en un nombre quelconque d'étapes. Il suffit pour cela de considérer la fermeture transitive et réflexive de la relation $\underset{G}{\Rightarrow}$, que l'on note $\overset{\star}{\underset{G}{\Rightarrow}}$:

Définition 5.3 (Dérivation)**.** *On définit la relation* $\overset{\star}{\underset{G}{\Rightarrow}}$ *sur l'ensemble* $V^\star \times V^\star$ *par* $\alpha_1 \overset{\star}{\underset{G}{\Rightarrow}} \alpha_m$ *si et seulement si il existe* $\alpha_2, ..., \alpha_{m-1}$ *dans* $V^\star$ *tels que* $\alpha_1 \underset{G}{\Rightarrow} \alpha_2 \underset{G}{\Rightarrow} ... \underset{G}{\Rightarrow} \alpha_{m-1} \underset{G}{\Rightarrow} \alpha_m$.

Ainsi, par exemple, la dérivation suivante est une dérivation pour la grammaire 5.1 :

$$S \underset{G_1}{\Rightarrow} aSb \underset{G_1}{\Rightarrow} aaSbb \underset{G_1}{\Rightarrow} aaaSbbb \underset{G_1}{\Rightarrow} aaaaSbbbb$$

permettant de déduire que $S \overset{\star}{\underset{G_1}{\Rightarrow}} aaaaSbbbb$.

Ces définitions préliminaires étant posées, il est maintenant possible d'exprimer formellement le lien entre grammaires et langages.

Définition 5.4 (Langage engendré par une grammaire)**.** *On appelle* langage engendré *par G, noté* $L(G)$*, le sous-ensemble de* $\Sigma^\star$ *défini par* $L(G) = \{w \in \Sigma^\star \mid S \overset{\star}{\underset{G}{\Rightarrow}} w\}$.

$L(G)$ est donc un sous-ensemble de $\Sigma^\star$, contenant précisément les mots qui se dérivent par $\overset{\star}{\underset{G}{\Rightarrow}}$ depuis S. Lorsque α, contenant des non-terminaux, se dérive de S, on dit qu'α est un *proto-mot* (en anglais *sentential form*). Pour produire un mot[1] du langage, il faudra alors habilement utiliser les productions, de manière à se débarrasser, par récritures successives depuis l'axiome, de tous les non-terminaux du proto-mot courant. Vous vérifierez ainsi que le langage engendré par la grammaire 5.1 est l'ensemble des mots formés en concaténant n fois le symbole a, suivi de n fois le symbole b (soit le langage $\{a^n b^n, n \geq 1\}$).

Si toute grammaire engendre un langage unique, la réciproque n'est pas vraie. Un langage L peut être engendré par de multiples grammaires, comme on peut s'en persuader en rajoutant

1. La terminologie est ainsi faite que lorsque l'on parle de grammaires, il est d'usage d'appeler *phrases* les séquences d'éléments terminaux, que nous appelions précédemment *mots*. Il s'ensuit parfois (et c'est fort fâcheux), que le terme de *mot* est parfois employé pour dénoter les éléments de l'alphabet Σ (et non plus les éléments de $\Sigma^\star$). Nous essaierons d'éviter cette confusion, et nous continuerons de parler de mots pour désigner les éléments d'un langage, même lorsque ces mots correspondront à ce qu'il est commun d'appeler phrase dans le langage courant.

à volonté des non-terminaux ou des terminaux inutiles. Il existe plusieurs manières pour un non-terminal d'être inutile : par exemple en n'apparaissant dans aucune partie droite (ce qui fait qu'il n'apparaîtra dans aucun proto-mot) ; ou bien en n'apparaissant dans aucune partie gauche (il ne pourra jamais être éliminé d'un proto-mot, ni donc être utilisé dans la dérivation d'une phrase du langage...). Nous reviendrons sur cette notion d'utilité des éléments de la grammaire à la section 9.1.2.

Comme nous l'avons fait pour les expressions rationnelles (à la section 3.1.3) et pour les automates finis (voir la section 4.1), il est en revanche possible de définir des classes d'équivalence de grammaires qui engendrent le même langage.

Définition 5.5 (Équivalence entre grammaires)**.** *Deux grammaires G_1 et G_2 sont* équivalentes *si et seulement si elles engendrent le même langage.*

Les grammaires syntagmatiques décrivent donc des systèmes permettant d'engendrer, par récritures successives, les mots d'un langage : pour cette raison, elles sont parfois également appelées *grammaires génératives*. De ces grammaires se déduisent (de manière plus ou moins directe) des algorithmes permettant de *reconnaître* les mots d'un langage, voire, pour les sous-classes les plus simples, de *décider* si un mot appartient ou non à un langage.

Le processus de construction itérative d'un mot par récriture d'une suite de proto-mots permet de tracer les étapes de la construction et apporte une information indispensable pour *interpréter* le mot. Les grammaires syntagmatiques permettent donc également d'associer des structures aux mots d'un langage, à partir desquelles il est possible de calculer leur signification. Ces notions sont formalisées à la section 6.2.

5.2 La hiérarchie de Chomsky

Chomsky identifie une hiérarchie de familles de grammaires de complexité croissante, chaque famille correspondant à une contrainte particulière sur la forme des règles de récriture. Cette hiérarchie, à laquelle correspond une hiérarchie [2] de langages, est présentée dans les sections suivantes.

5.2.1 Grammaires de type 0

Définition 5.6 (Type 0)**.** *On appelle grammaire de type 0 une grammaire syntagmatique dans laquelle la forme des productions est non-contrainte.*

Le type 0 est le type le plus général. Toute grammaire syntagmatique est donc de type 0 ; toutefois, au fur et à mesure que les autres types seront introduits, on prendra pour convention d'appeler *type d'une grammaire* le type le plus spécifique auquel elle appartient.

Le principal résultat à retenir pour les grammaires de type 0 est le suivant, que nous ne démontrerons pas.

2. Les développements des travaux sur les grammaires formelles ont conduit à largement raffiner cette hiérarchie. Il existe ainsi, par exemple, de multiples sous-classes des langages algébriques, dont certaines seront présentées dans la suite.

Théorème 5.7. *Les langages récursivement énumérables sont les langages engendrés par une grammaire de type 0.*

L'ensemble des langages récursivement énumérables est noté $\mathcal{RE}$. Rappelons qu'il a été introduit à la section 2.2.2.

Ce qui signifie qu'en dépit de leur apparente simplicité, les grammaires syntagmatiques permettent de décrire (pas toujours avec élégance, mais c'est une autre question) exactement les langages que l'on sait reconnaître (ou énumérer) avec une machine de Turing. Les grammaires syntagmatiques de type 0 ont donc une expressivité maximale.

Ce résultat est théoriquement satisfaisant, mais ne nous en dit guère sur la véritable nature de ces langages. Dans la pratique, il est extrêmement rare de rencontrer un langage de type 0 qui ne se ramène pas à un type plus simple.

5.2.2 Grammaires contextuelles (type 1)

Les grammaires *monotones* introduisent une première restriction sur la forme des règles, en imposant que la partie droite de chaque production soit nécessairement plus longue que la partie gauche. Formellement :

Définition 5.8 (Grammaire monotone)**.** *On appelle* grammaire monotone *une grammaire syntagmatique dans laquelle toute production $\alpha \to \beta$ est telle que $|\alpha| \leq |\beta|$.*

Cette définition impose qu'au cours d'une dérivation, les proto-mots d'une grammaire monotone s'étendent de manière monotone. Cette contrainte interdit en particulier d'engendrer le mot vide ε. Nous reviendrons en détail sur cette question à la section 5.2.6.

Les langages engendrés par les grammaires monotones sont également obtenus en posant une contrainte alternative sur la forme des règles, conduisant à la notion de grammaire contextuelle :

Définition 5.9 (Grammaire contextuelle)**.** *On appelle* grammaire contextuelle*, ou* grammaire sensible au contexte*, en abrégé* grammaire CS *(en anglais Context-Sensitive) une grammaire telle que toute production de G est de la forme $\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$, avec $\alpha_1, \alpha_2, \beta$ dans $V^\star$, $\beta \neq \varepsilon$ et A est dans N.*

Les grammaires contextuelles sont donc telles que chaque production ne récrit qu'un symbole non-terminal à la fois, le contexte (c'est-à-dire les symboles encadrant ce non-terminal) restant inchangé : d'où la qualification de *contextuelles* pour ces grammaires. Par construction, toute grammaire contextuelle est monotone (car $\beta \neq \varepsilon$). Le théorème suivant nous dit plus : grammaires contextuelles et monotones engendrent exactement les mêmes langages.

Théorème 5.10. *Pour tout langage L engendré par une grammaire monotone G, il existe une grammaire contextuelle qui engendre L.*

Démonstration. Soit L engendré par la grammaire monotone $G = (N, \Sigma, S, P)$. Sans perte de généralité, nous supposons que les terminaux n'apparaissent que dans des productions de

la forme $A \to a$. Si cela n'est pas le cas, il suffit d'introduire un nouveau non-terminal X_a pour chaque terminal, de remplacer a par X_a dans toutes les productions, et d'ajouter à P la production $X_a \to a$. Nous allons construire une grammaire contextuelle équivalente à G et pour cela nous allons démontrer deux résultats intermédiaires. □

Lemme 5.11. *Si G est une grammaire monotone, il existe une grammaire monotone équivalente dans laquelle toutes les parties droites ont une longueur inférieure ou égale à* 2.

Démonstration. En procédant par récurrence sur le nombre de productions de G qui ne satisfont pas la condition, il suffit d'illustrer comment récrire une seule telle production. Ainsi, soit $\alpha = \alpha_1 \dots \alpha_m \to \beta = \beta_1 \dots \beta_n$ une production de G. G étant monotone, $m \leqslant n$: complétons alors α avec des ε de manière à écrire $\alpha = \alpha_1 \dots \alpha_n$, où les α_i sont dans $\Sigma \cup N \cup \{\varepsilon\}$. Construisons ensuite G', déduite de G en introduisant les $n-1$ nouveaux non-terminaux $X_1 \dots X_{n-1}$, et en remplaçant $\alpha \to \beta$ par les n les productions suivantes :

$$\begin{array}{rcll} \alpha_1\alpha_2 & \to & \beta_1 X_1 & \\ X_{i-1}\alpha_{i+1} & \to & \beta_i X_i & \forall i, 1 < i < n \\ X_{n-1} & \to & \beta_n & \end{array}$$

Il est clair que G' est monotone et que l'application successive de ces nouvelles règles a bien pour effet global de récrire de proche en proche α en β par :

$$\begin{array}{rcl} \alpha_1\alpha_2 \dots \alpha_m & \underset{G'}{\Rightarrow} & \beta_1 X_1 \alpha_3 \dots \alpha_m \\ & \underset{G'}{\Rightarrow} & \beta_1\beta_2 X_2 \alpha_4 \dots \alpha_n \\ & \underset{G'}{\Rightarrow} & \dots \\ & \underset{G'}{\Rightarrow} & \beta_1\beta_2 \dots \beta_{n-1} X_{n-1} \\ & \underset{G'}{\Rightarrow} & \beta_1\beta_2 \dots \beta_{n-1}\beta_n \end{array}$$

Soit alors u dans $L(G)$: si u se dérive de S sans utiliser $\alpha \to \beta$, alors u se dérive pareillement de S dans G'. Si u se dérive dans G par $S \overset{\star}{\underset{G}{\Rightarrow}} v\alpha w \underset{G}{\Rightarrow} v\beta w \overset{\star}{\underset{G}{\Rightarrow}} u$, alors u se dérive dans G' en remplaçant l'étape de dérivation $v\alpha w \underset{G}{\Rightarrow} v\beta w$ par les étapes de dérivation détaillées ci-dessus. Inversement, si u se dérive dans G' sans utiliser aucune variable X_i, elle se dérive à l'identique dans G. Si X_1 apparaît dans une étape de la dérivation, son élimination par application successive des n règles précédentes implique la présence du facteur α dans le proto-mot dérivant u : u se dérive alors dans G en utilisant $\alpha \to \beta$. □

Par itération de cette transformation, il est possible de transformer toute grammaire monotone en une grammaire monotone équivalente dont toutes les productions sont telles que leur partie droite (et, par conséquent, leur partie gauche, par monotonicité de G) ont une longueur inférieure ou égale à 2. Le lemme suivant nous permet d'arriver à la forme désirée pour les parties gauches (les productions de la forme $A \to a$ pouvant être gardées telles quelles).

Lemme 5.12. *Si G est une grammaire monotone ne contenant que des productions de type $AB \to CD$, avec A, B, C et D dans $N \cup \{\varepsilon\}$, alors il existe une grammaire équivalente satisfaisant la propriété du théorème 5.10.*

Démonstration. Cette démonstration utilise le même principe que la démonstration précédente, en introduisant pour chaque production $R = AB \rightarrow CD$ le nouveau non-terminal X_{AB} et en remplaçant R par les trois productions suivantes :

— $AB \rightarrow X_{AB}B$

— $X_{AB}B \rightarrow X_{AB}D$

— $X_{AB}D \rightarrow CD$

□

On déduit directement le résultat qui nous intéresse : toute grammaire monotone est équivalente à une grammaire dans laquelle chaque production ne récrit qu'un seul et unique symbole.

Ces résultats permettent finalement d'introduire la notion de langage contextuel.

Définition 5.13 (Langage contextuel)**.** *On appelle* langage contextuel *(ou langage* sensible au contexte*, en abrégé* langage CS*) un langage engendré par une grammaire contextuelle (ou par une grammaire monotone).*

Les langages contextuels constituent une classe importante de langages, que l'on rencontre effectivement (en particulier en traitement automatique des langues). Un représentant notable de ces langages est le langage $\{a^n b^n c^n, n \geq 1\}$, qui est engendré par la grammaire 5.2 d'axiome S.

$$\begin{array}{lll} p_1 = S & \rightarrow aSQ \\ p_2 = S & \rightarrow abc \\ p_3 = cQ & \rightarrow Qc \\ p_4 = bQc & \rightarrow bbcc \end{array}$$

Grammaire 5.2 – Une grammaire pour $a^n b^n c^n$

La grammaire 5.2 est monotone. Dans cette grammaire, on observe par exemple les dérivations listées dans le tableau 5.3.

Il est possible de montrer qu'en fait les seules dérivations qui réussissent dans cette grammaire produisent les mots (et tous les mots) du langage $\{a^n b^n c^n, n \geq 1\}$, ce qui est d'ailleurs loin d'être évident lorsque l'on examine les productions de la grammaire.

Un dernier résultat important concernant les langages contextuels est le suivant :

Théorème 5.14. *Tout langage contextuel est récursif, soit en notant $\mathcal{RC}$ l'ensemble des langages récursifs, et $\mathcal{CS}$ l'ensemble des langages contextuels : $\mathcal{CS} \subset \mathcal{RC}$.*

Ce que dit ce résultat, c'est qu'il existe un algorithme capable de décider si un mot appartient ou pas au langage engendré par une grammaire contextuelle. Pour s'en convaincre, esquissons le raisonnement suivant : soit u le mot à décider, de taille $|u|$. Tout proto-mot impliqué dans la dérivation de u est au plus aussi long que u, à cause de la propriété de monotonie.

S	$\underset{G}{\overset{p_2}{\Rightarrow}}$	abc
S	$\underset{G}{\overset{p_1}{\Rightarrow}}$	aSQ
	$\underset{G}{\overset{p_2}{\Rightarrow}}$	$aabcQ$
	$\underset{G}{\overset{p_3}{\Rightarrow}}$	$aabQc$
	$\underset{G}{\overset{p_4}{\Rightarrow}}$	$aabbcc$
S	$\underset{G}{\overset{p_1}{\Rightarrow}}$	aSQ
	$\underset{G}{\overset{p_1}{\Rightarrow}}$	$aaSQQ$
	$\underset{G}{\overset{p_2}{\Rightarrow}}$	$aaabcQQ$
	$\underset{G}{\overset{p_3}{\Rightarrow}}$	$aaabQcQ$
	$\underset{G}{\overset{p_4}{\Rightarrow}}$	$aaabbccQ$
	$\underset{G}{\overset{p_3}{\Rightarrow}}$	$aaabbcQc$
	$\underset{G}{\overset{p_3}{\Rightarrow}}$	$aaabbQcc$
	$\underset{G}{\overset{p_4}{\Rightarrow}}$	$aaabbbccc$

TABLE 5.3 – Des dérivations pour $a^n b^n c^n$

Il « suffit » donc, pour décider u, de construire de proche en proche l'ensemble D de tous les proto-mots qu'il est possible d'obtenir en « inversant les productions » de la grammaire. D contient un nombre fini de proto-mots ; il est possible de construire de proche en proche les éléments de cet ensemble. Au terme de processus, si S est trouvé dans D, alors u appartient à $L(G)$, sinon, u n'appartient pas à $L(G)$.

Ce bref raisonnement ne dit rien de la complexité de cet algorithme, c'est-à-dire du temps qu'il mettra à se terminer. Dans la pratique, tous les algorithmes généraux pour les grammaires CS sont exponentiels. Quelques sous-classes particulières admettent toutefois des algorithmes de décision polynomiaux.

Pour finir, notons qu'il est possible de montrer que la réciproque n'est pas vraie, donc qu'il existe des langages récursifs *qui ne peuvent être décrits par aucune grammaire contextuelle*.

5.2.3 Grammaires hors-contexte (type 2)

Une contrainte supplémentaire par rapport à celle imposée pour les grammaires contextuelles consiste à exiger que toutes les productions contextuelles aient un contexte vide. Ceci conduit à la définition suivante.

Définition 5.15 (Grammaire hors-contexte)**.** *On appelle grammaire de type 2 (on dit également* grammaire hors-contexte*, en anglais* Context-free*, ou* grammaire algébrique*, en abrégé* CFG*) une grammaire syntagmatique dans laquelle toute production $\alpha \to \beta$ est de la forme : $A \to \beta$, avec A*

dans N et β dans V^+.

Le sens de cette nouvelle contrainte est le suivant : chaque fois qu'une variable A figure dans un proto-mot, elle peut être récrite *indépendamment du contexte* dans lequel elle apparaît. Par définition, toute grammaire hors-contexte est un cas particulier de grammaire contextuelle.

Cette nouvelle restriction permet d'introduire la notion de A-production :

Définition 5.16 (A-production)**.** *On appelle A-production une production dont la partie gauche est réduite au symbole A.*

Les langages hors-contexte se définissent alors par :

Définition 5.17 (Langage hors-contexte)**.** *On appelle* langage hors-contexte *(ou langage* algébrique*, en abrégé* langage CF*) un langage engendré par une grammaire hors-contexte.*

Attention : il est aisé de faire des définitions complexes d'entités simples. Une grammaire compliquée, par exemple sensible au contexte, peut définir un langage simple, par exemple hors-contexte, rationnel, voire fini.

$$\begin{aligned} p_1 &= S \quad \rightarrow aSb \\ p_2 &= aSb \rightarrow aaSbb \\ p_3 &= S \quad \rightarrow ab \end{aligned}$$

Grammaire 5.4 – Une grammaire contextuelle pour $a^n b^n$

Par exemple, la grammaire 5.4, contextuelle, engendre le langage hors-contexte $a^n b^n$. Ce langage étant un langage CF, on peut trouver une grammaire CF pour ce langage (par exemple la grammaire 5.1).

On notera $\mathcal{CF}$ l'ensemble des langages hors-contexte. Ces langages constituent probablement la classe de langages la plus étudiée et la plus utilisée dans les applications pratiques. Nous aurons l'occasion de revenir en détail sur ces langages dans les chapitres qui suivent et d'en étudier de nombreux exemples, en particulier dans le chapitre 6.

Notons simplement, pour l'instant, que cette classe contient des langages non-triviaux, par exemple $a^n b^n$, ou encore le langage des palindromes (dont l'écriture d'une grammaire est laissée en exercice).

5.2.4 Grammaires régulières (type 3)

Régularité

Les grammaires de type 3 réduisent un peu plus la forme des règles autorisées, définissant une classe de langages encore plus simple que la classe des langages hors-contexte.

Définition 5.18 (Grammaire de type 3)**.** *On appelle grammaire de type 3 (on dit également* grammaire régulière*, en abrégé* grammaire RG*) (en anglais* regular*) une grammaire syntagmatique*

dans laquelle toute production $\alpha \to \beta$ est soit de la forme $A \to aB$, avec a dans Σ et A, B dans N, soit de la forme $A \to a$.

Par définition, toute grammaire régulière est hors-contexte. Le trait spécifique des grammaires régulières est que chaque règle produit un symbole terminal et au plus un non-terminal à sa droite. Les dérivations d'une grammaire régulière ont donc une forme très simple, puisque tout proto-mot produit par n étapes successives de dérivation contient en tête n symboles terminaux, suivis d'au plus un non-terminal. La dérivation se termine par la production d'un mot du langage de la grammaire, lorsque le dernier terminal est éliminé par une récriture de type $A \to a$.

La notion de langage régulier se définit comme précédemment par :

Définition 5.19 (Langage régulier)**.** *Un langage est* régulier *si et seulement si il existe une grammaire régulière qui l'engendre.*

On notera $\mathcal{RG}$ l'ensemble des langages réguliers.

Les réguliers sont rationnels

Le résultat principal concernant les langages réguliers est énoncé dans le théorème suivant :

Théorème 5.20. *Si L est un langage régulier, il existe un automate fini A qui reconnaît L. Réciproquement, si L est un langage reconnaissable, avec $\varepsilon \notin L$, alors il existe une grammaire régulière qui engendre L.*

Ainsi les langages réguliers, à un détail près (que nous discutons à la section 5.2.6), ne sont rien d'autre que les langages rationnels, aussi connus sous le nom de reconnaissables. Leurs propriétés principales ont été détaillées par ailleurs (notamment à la section 4.2) ; nous y renvoyons le lecteur.

Pour avoir l'intuition de ce théorème, considérons la grammaire engendrant le langage $aa(a+b)^\star a$ et l'automate reconnaissant ce langage, tous deux reproduits dans le tableau 5.5.

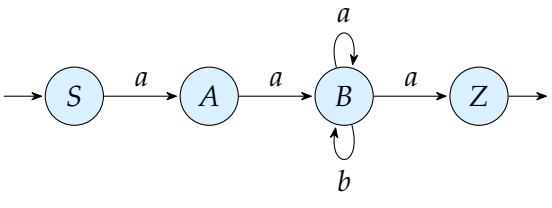
G	A
$S \to aA$ $A \to aB$ $B \to aB \mid bB$ $B \to a$	

TABLE 5.5 – Grammaire et automate pour $aa(a+b)^\star a$

Dans G, la dérivation de l'entrée $aaba$ est $S \underset{G}{\Rightarrow} aA \underset{G}{\Rightarrow} aaB \underset{G}{\Rightarrow} aabB \underset{G}{\Rightarrow} aaba$, correspondant au calcul : $(S, aabb) \vdash_A (A, abb) \vdash_A (B, bb) \vdash_A (B, a) \vdash_A (F, \varepsilon)$. Cet exemple illustre la symétrie du rôle joué par les variables de la grammaire et les états de l'automate. Cette symétrie est formalisée

dans la construction suivante, qui vise à fournir une démonstration de l'équivalence entre rationnels et réguliers.

Démonstration du théorème 5.20. Soient $G = (N, \Sigma, P, S)$ une grammaire régulière et A l'automate dérivé de G suivant $A = (\Sigma, N \cup \{Z\}, S, \{Z\}, \delta)$, avec la fonction δ définie selon :

— $\delta(A, a) = B \Leftrightarrow (A \to aB) \in P$

— $\delta(A, a) = Z \Leftrightarrow (A \to a) \in P$

Montrons maintenant que $L(G) = L(A)$ et pour cela, montrons par induction l'équivalence suivante : $S \overset{\star}{\underset{G}{\Rightarrow}} uB$ si et seulement si $(S, u) \vdash_A^\star (B, \varepsilon)$. Cette équivalence est vraie par définition si u est de longueur 1. Supposons-la vraie jusqu'à une longueur n, et considérons : $u = avb$ tel que $S \overset{\star}{\underset{G}{\Rightarrow}} avA \underset{G}{\Rightarrow} avbB$. Par l'hypothèse de récurrence il s'ensuit que $(S, av) \vdash_A^\star (A, \varepsilon)$, et donc que $(S, avb) \vdash_A^\star (B, \varepsilon)$ par construction de δ. Pour conclure, reste à examiner comment une dérivation se termine : il faut nécessairement éliminer la dernière variable par une production de type $A \to a$; ceci correspond à une transition dans l'état Z, unique état final de A. Inversement, un calcul réussi dans A correspond à une dérivation « éliminant » toutes les variables et donc à un mot du langage $L(G)$.

La construction réciproque, permettant de construire une grammaire équivalente à un automate fini quelconque, est exactement inverse : il suffit d'associer une variable à chaque état de l'automate (en prenant l'état initial pour axiome) et de rajouter une production par transition. Cette construction est achevée en rajoutant une nouvelle production $A \to a$ pour toute transition $\delta(A, a)$ aboutissant dans un état final. □

Une conséquence de cette équivalence est que si tout langage régulier est par définition hors contexte, le contraire n'est pas vrai. Nous avons rencontré plus haut (à la section 5.2.3) une grammaire engendrant $\{a^n b^n, n \geq 1\}$, langage dont on montre aisément qu'en vertu du lemme de pompage pour les langages rationnels (cf. la section 4.3.1), il ne peut être reconnu par un automate fini. La distinction introduite entre langages réguliers et langages hors-contexte n'est donc pas de pure forme : il existe des langages CF qui ne sont pas rationnels.

Exercice 5.21 (BCI-0405-1, suite (exercice 4.15))**.** *2 Construisez, par une méthode du cours, une grammaire régulière G telle que $L(G) = L(A)$, où A est l'automate 4.14.*

Variantes

Il est possible de définir de manière un peu plus libérale les grammaires de type 3 : la limitation essentielle concerne en fait le nombre de non-terminaux en partie droite et leur positionnement systématique à droite de tous les terminaux.

Définition 5.22 (Grammaire de type 3)**.** *On appelle grammaire de type 3 (ici* grammaire linéaire à droite*) (en anglais* right linear*) une grammaire syntagmatique dans laquelle toute production $\alpha \to \beta$ est soit de la forme : $A \to uB$, avec u dans $\Sigma^\star$ et A, B dans N, soit de la forme $A \to u$, avec u dans Σ^+.*

Cette définition est donc plus générale que la définition des grammaires régulières. Il est pourtant simple de montrer que les langages engendrés sont les mêmes : chaque grammaire linéaire à droite admet une grammaire régulière équivalente. La démonstration est laissée en exercice. Les grammaires linéaires à droite engendrent donc exactement la classe des langages rationnels.

On montre, de même, que les *grammaires linéaires à gauche* (au plus un non-terminal dans chaque partie droite, toujours positionné en première position) engendrent les langages rationnels. Ces grammaires sont donc aussi des grammaires de type 3.

Attention : ces résultats ne s'appliquent plus si l'on considère les *grammaires linéaires* quelconques (définies comme étant les grammaires telles que toute partie droite contient au plus un non-terminal) : *les grammaires linéaires peuvent engendrer des langages hors-contexte non rationnels*. Pour preuve, il suffit de considérer de nouveau la grammaire 5.1.

On ne peut pas non plus « généraliser » en acceptant les grammaires dont les règles panachent linéarité à gauche *et* à droite. Il est facile de reconnaître la grammaire 5.1 dans le déguisement linéaire suivant.

$$S \to aX \mid ab$$
$$X \to Sb$$

5.2.5 Grammaires à choix finis (type 4)

Il existe des grammaires encore plus simples que les grammaires régulières. En particulier les grammaires à choix finis sont telles que tout non-terminal ne récrit que des terminaux. On appelle de telles grammaires les grammaires de type 4 :

Définition 5.23 (Grammaire de type 4)**.** *On appelle grammaire de type 4 (on dit également* grammaire à choix finis*, en abrégé* grammaire FC*) une grammaire syntagmatique dans laquelle toute production est de la forme : $A \to u$, avec A dans N et u dans Σ^+.*

Les grammaires de type 4, on s'en convaincra aisément, n'engendrent que des langages finis, mais engendrent tous les langages finis qui ne contiennent pas le mot vide.

5.2.6 Les productions ε

La définition que nous avons donnée des grammaires de type 1 implique un accroissement monotone de la longueur des proto-mots au cours d'une dérivation, ce qui interdit la production du mot vide. Cette « limitation » théorique de l'expressivité des grammaires contextuelles a conduit à énoncer une version « faible » (c'est-à-dire ne portant que sur les reconnaissables ne contenant pas le mot vide) de l'équivalence entre langages réguliers et langages rationnels.

Pour résoudre cette limitation, on admettra qu'une grammaire contextuelle (ou hors-contexte, ou régulière) puisse contenir des règles de type $A \to \varepsilon$ et on acceptera la possibilité qu'un langage contextuel contienne le mot vide. Pour ce qui concerne les grammaires CF, nous reviendrons sur ce point en étudiant les algorithmes de normalisation de grammaires (à la section 9.1.4).

Pour ce qui concerne les grammaires régulières, notons simplement que si l'on accepte des productions de type $A \rightarrow \varepsilon$, alors on peut montrer que les langages réguliers sont exactement les langages reconnaissables. Il suffit pour cela de compléter la construction décrite à la section 5.2.4 en marquant comme états finaux de A tous les états correspondant à une variable pour laquelle il existe une règle $A \rightarrow \varepsilon$. La construction inverse en est également simplifiée : pour chaque état final A de l'automate, on rajoute la production $A \rightarrow \varepsilon$.

Pour différentier les résultats obtenus avec et sans ε, on utilisera le qualificatif de *strict* pour faire référence aux classes de grammaires et de langages présentées dans les sections précédentes : ainsi on dira qu'une grammaire est *strictement hors-contexte* si elle ne contient pas de production de type $A \rightarrow \varepsilon$; qu'elle est simplement hors-contexte sinon. De même on parlera de langage strictement hors-contexte pour un langage CF ne contenant pas ε et de langage hors-contexte sinon.

5.2.7 Conclusion

Le titre de la section introduisait le terme de hiérarchie : quelle est-elle finalement ? Nous avons en fait montré dans cette section la série d'inclusions suivante :

$$\mathcal{FC} \subset \mathcal{RG} \subset \mathcal{CF} \subset \mathcal{CS} \subset \mathcal{RC} \subset \mathcal{RE}$$

Il est important de bien comprendre le sens de cette hiérarchie : les grammaires les plus complexes ne décrivent pas des langages plus grands, mais permettent d'exprimer des distinctions plus subtiles entre les mots du langage de la grammaire et ceux qui sont ne sont pas grammaticaux.

Un autre point de vue, sans doute plus intéressant, consiste à dire que les grammaires plus complexes permettent de décrire des « lois » plus générales que celles exprimables par des grammaires plus simples. Prenons un exemple en traitement des langues naturelles : en français, le bon usage impose que le sujet d'une phrase affirmative s'accorde avec son verbe en nombre et personne. Cela signifie, par exemple, que si le sujet est un pronom singulier à la première personne, le verbe sera également à la première personne du singulier. Si l'on appelle L l'ensemble des phrases respectant cette règle, on a :

— la phrase *l'enfant mange* est dans L

— la phrase *l'enfant manges* n'est pas dans L

Une description de la syntaxe du français par une grammaire syntagmatique qui voudrait exprimer cette règle avec une grammaire régulière est peut-être possible, mais serait certainement très fastidieuse, et sans aucune vertu pour les linguistes, ou pour les enfants qui apprennent cette règle. Pourquoi ? Parce qu'en français, le sujet peut être séparé du verbe par un nombre arbitraire de mots, comme dans *l'enfant de Jean mange*, *l'enfant du fils de Jean mange*... Il faut donc implanter dans les règles de la grammaire un dispositif de mémorisation du nombre et de la personne du sujet qui « retienne » ces valeurs jusqu'à ce que le verbe soit rencontré, c'est-à-dire pendant un nombre arbitraire d'étapes de dérivation. La seule solution, avec une grammaire régulière, consiste à envisager à l'avance toutes les configurations possibles, et de les encoder dans la topologie de l'automate correspondant, puisque la mémorisation « fournie » par la grammaire est de taille 1 (le passé de la dérivation

est immédiatement oublié). On peut montrer qu'une telle contrainte s'exprime simplement et sous une forme bien plus naturelle pour les linguistes, lorsque l'on utilise une grammaire CF.

La prix à payer pour utiliser une grammaire plus fine est que le problème algorithmique de la reconnaissance d'une phrase par une grammaire devient de plus en plus complexe. Vous savez déjà que ce problème est solvable en temps linéaire pour les langages rationnels (réguliers), et non-solvable (indécidable) pour les langages de type 0. On peut montrer qu'il existe des automates généralisant le modèle d'automate fini pour les langages CF et CS, d'où se déduisent des algorithmes (polynomiaux pour les langages CF, exponentiels en général pour les langages CS) permettant également de décider ces langages. En particulier, le problème de la reconnaissance pour les grammaires CF sera étudié en détail dans les chapitres 7 et 8.

Chapitre 6

Langages et grammaires hors-contexte

Dans ce chapitre, nous nous intéressons plus particulièrement aux grammaires et aux langages hors-contexte. Rappelons que nous avons caractérisé ces grammaires à la section 5.2.3 comme étant des grammaires syntagmatiques dont toutes les productions sont de la forme $A \to u$, avec A un non-terminal et u une séquence quelconque de terminaux et non-terminaux. Autrement dit, une grammaire algébrique est une grammaire pour laquelle il est toujours possible de récrire un non-terminal en cours de dérivation, quel que soit le contexte (les symboles adjacents dans le proto-mot) dans lequel il apparaît.

Ce chapitre débute par la présentation, à la section 6.1, de quelques grammaires CF exemplaires. Cette section nous permettra également d'introduire les systèmes de notation classiques pour ces grammaires. Nous définissons ensuite la notion de dérivation gauche et d'arbre de dérivation, puis discutons le problème de l'équivalence entre grammaires et de l'ambiguïté (section 6.2). La section 6.3 introduit enfin un certain nombre de propriétés élémentaires des langages CF, qui nous permettent de mieux cerner la richesse (et la complexité) de cette classe de langages. L'étude des grammaires CF, en particulier des algorithmes permettant de traiter le problème de la reconnaissance d'un mot par une grammaire, sera poursuivie dans les chapitres suivants, en particulier au chapitre 7.

6.1 Quelques exemples

6.1.1 La grammaire des déjeuners du dimanche

La grammaire 6.1 est un exemple de grammaire hors-contexte correspondant à une illustration très simplifiée de l'utilisation de ces grammaires pour des applications de traitement du langage naturel. Cette grammaire engendre un certain nombre d'énoncés du français. Dans cette grammaire, les non-terminaux sont en majuscules, les terminaux sont des mots du vocabulaire usuel. On utilise également dans cette grammaire le symbole "|" pour exprimer une alternative : $A \to u \mid v$ vaut pour les deux règles $A \to u$ et $A \to v$.

Première remarque sur la grammaire 6.1 : elle contient de nombreuses règles de type $A \to a$, qui servent simplement à introduire les symboles terminaux de la grammaire : les symboles

p_1	S	$\to GN\ GV$	p_{15}	V	$\to mange \mid sert$
p_2	GN	$\to DET\ N$	p_{16}	V	$\to donne$
p_3	GN	$\to GN\ GNP$	p_{17}	V	$\to boude \mid s'ennuie$
p_4	GN	$\to NP$	p_{18}	V	$\to parle$
p_5	GV	$\to V$	p_{19}	V	$\to coupe \mid avale$
p_6	GV	$\to V\ GN$	p_{20}	V	$\to discute \mid gronde$
p_7	GV	$\to V\ GNP$	p_{21}	NP	$\to Louis \mid Paul$
p_8	GV	$\to V\ GN\ GNP$	p_{22}	NP	$\to Marie \mid Sophie$
p_9	GV	$\to V\ GNP\ GNP$	p_{23}	N	$\to fille \mid maman$
p_{10}	GNP	$\to PP\ GN$	p_{24}	N	$\to paternel \mid fils$
p_{11}	PP	$\to de \mid à$	p_{25}	N	$\to viande \mid soupe \mid salade$
p_{12}	DET	$\to la \mid le$	p_{26}	N	$\to dessert \mid fromage \mid pain$
p_{13}	DET	$\to sa \mid son$	p_{27}	ADJ	$\to petit \mid gentil$
p_{14}	DET	$\to un \mid une$	p_{28}	ADJ	$\to petite \mid gentille$

Grammaire 6.1 – La grammaire G_D des repas dominicaux

apparaissant en partie gauche de ces productions sont appelés *pré-terminaux*. En changeant le vocabulaire utilisé dans ces productions, on pourrait facilement obtenir une grammaire permettant d'engendrer des énoncés décrivant d'autres aspects de la vie quotidienne. Il est important de réaliser que, du point de vue de leur construction (de leur structure interne), les énoncés ainsi obtenus resteraient identiques à ceux de $L(G_D)$.

En utilisant la grammaire 6.1, on construit une première dérivation pour l'énoncé *Louis boude*, représentée dans le tableau 6.2 :

$S \underset{G_D}{\Rightarrow} GN\ GV$	(par p_1)
$S \underset{G_D}{\Rightarrow} NP\ GV$	(par p_4)
$\underset{G_D}{\Rightarrow} Louis\ GV$	(par p_{21})
$\underset{G_D}{\Rightarrow} Louis\ V$	(par p_5)
$\underset{G_D}{\Rightarrow} Louis\ boude$	(par p_{17})

TABLE 6.2 – Louis boude

Il existe d'autres dérivations de *Louis boude*, consistant à utiliser les productions dans un ordre différent : par exemple p_5 avant p_4, selon : $S \underset{G_D}{\Rightarrow} GN\ GV \underset{G_D}{\Rightarrow} GN\ V \underset{G_D}{\Rightarrow} GN\ boude \underset{G_D}{\Rightarrow} NP\ boude \underset{G_D}{\Rightarrow} Louis\ boude$. Ceci illustre une première propriété importante des grammaires hors contexte : lors d'une dérivation, il est possible d'appliquer les productions dans un ordre arbitraire. Notons également qu'à la place de *Louis*, on aurait pu utiliser *Marie* ou *Paul*, ou même *le fils boude* et obtenir un nouveau mot du langage : à nouveau, c'est la propriété d'indépendance au contexte qui s'exprime. Ces exemples éclairent un peu la signification des noms de variables : *GN* désigne les groupes nominaux, *GV* les groupes verbaux... La première

production de G_D dit simplement qu'un énoncé bien formé est composé d'un groupe nominal (qui, le plus souvent, est le sujet) et d'un groupe verbal (le verbe principal) de la phrase.

Cette grammaire permet également d'engendrer des énoncés plus complexes, comme par exemple :

le paternel sert le fromage

qui utilise une autre production (p_6) pour dériver un groupe verbal (GV) contenant un verbe et son complément d'objet direct.

La production p_3 est particulière, puisqu'elle contient le même symbole dans sa partie gauche et dans sa partie droite. On dit d'une production possédant cette propriété qu'elle est *récursive*. Pour être plus précis p_3 est récursive à gauche : le non-terminal en partie gauche de la production figure également en tête de la partie droite. Comme c'est l'élément le plus à gauche de la partie droite, on parle parfois de *coin gauche* de la production.

Cette propriété de p_3 implique immédiatement que le langage engendré par G_D est infini, puisqu'on peut créer des énoncés de longueur arbitraire par application itérée de cette règle, engendrant par exemple :

— *le fils de Paul mange*

— *le fils de la fille de Paul mange*

— *le fils de la fille de la fille de Paul mange*

— *le fils de la fille de la fille ... de Paul mange*

Il n'est pas immédiatement évident que cette propriété, à savoir qu'une grammaire contenant une règle récursive engendre un langage infini, soit toujours vraie... Pensez, par exemple, à ce qui se passerait si l'on avait une production récursive de type $A \rightarrow A$. En revanche, il est clair qu'une telle production risque de poser problème pour engendrer de manière systématique les mots de la grammaire. Le problème est d'éviter de tomber dans des dérivations interminables telles que : $GN \underset{G_D}{\Rightarrow} GN\ GNP \underset{G_D}{\Rightarrow} GN\ GNP\ GNP \underset{G_D}{\Rightarrow} GN\ GNP\ GNP\ GNP \ldots$.

Une autre remarque sur G_D : cette grammaire engendre des énoncés qui ne sont pas corrects en français. Par exemple, *la fils avale à le petite dessert*.

Dernière remarque concernant G_D : cette grammaire engendre des énoncés qui sont (du point de vue du sens) ambigus. Ainsi *Louis parle à la fille de Paul*, qui peut exprimer soit une conversation entre *Louis* et *la fille de Paul*, soit un échange concernant *Paul* entre *Louis* et *la fille*. En écrivant les diverses dérivations de ces deux énoncés, vous pourrez constater que les deux sens correspondent à deux dérivations employant des productions différentes pour construire un groupe verbal : la première utilise p_7, la seconde p_9.

6.1.2 Une grammaire pour le shell

Dans cette section, nous présentons des fragments [1] d'une autre grammaire, celle qui décrit la syntaxe des programmes pour l'interpréteur `bash`. Nous ferons, dans la suite, référence à cette grammaire sous le nom de G_B.

1. Ces fragments sont extraits de *Learning the Bash Shell*, de C. Newham et B. Rosenblatt, O'Reilly & associates, 1998, consultable en ligne à l'adresse `http://safari.oreilly.com`.

$$
\begin{aligned}
\langle \texttt{number} \rangle &\underset{G_B}{\Rightarrow} \langle \texttt{number} \rangle\ \texttt{0} \\
&\underset{G_B}{\Rightarrow} \langle \texttt{number} \rangle\ \texttt{10} \\
&\underset{G_B}{\Rightarrow} \langle \texttt{number} \rangle\ \texttt{510} \\
&\underset{G_B}{\Rightarrow} \texttt{3510}
\end{aligned}
$$

TABLE 6.4 – Dérivation du nombre `3510`

Un mot tout d'abord sur les notations : comme il est d'usage pour les langages informatiques, cette grammaire est exprimée en respectant les conventions initialement proposées par Backus et Naur pour décrire le langage ALGOL 60. Ce système de notation des grammaires est connu sous le nom de *Backus-Naur Form* ou en abrégé BNF. Dans les règles de la grammaire 6.3, les non-terminaux figurent entre chevrons (⟨⟩) ; les terminaux sont les autres chaînes de caractères, certains caractères spéciaux apparaissant entre apostrophes ; les productions sont marquées par l'opérateur ::= ; enfin les productions alternatives ayant même partie gauche sont séparées par le symbole |, les alternatives pouvant figurer sur des lignes distinctes.

Les éléments de base de la syntaxe sont les mots et les chiffres, définis dans la grammaire 6.3.

```
⟨letter⟩ ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
           | A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

⟨digit⟩ ::= 0|1|2|3|4|5|6|7|8|9

⟨number⟩ ::= ⟨number⟩ ⟨digit⟩
           | ⟨digit⟩

⟨word⟩ ::= ⟨word⟩ ⟨letter⟩
         | ⟨word⟩ '_'
         | ⟨letter⟩

⟨word_list⟩ ::= ⟨word_list⟩ ⟨word⟩
              |  ⟨word⟩
```

Grammaire 6.3 – Constructions élémentaires de Bash

Les deux premières définitions sont des simples énumérations de terminaux définissant les lettres (⟨`letter`⟩), puis les chiffres (⟨`digit`⟩). Le troisième non-terminal, ⟨`number`⟩, introduit une figure récurrente dans les grammaires informatiques : celle de la liste, ici une liste de chiffres. Une telle construction nécessite deux productions :

— une production récursive (ici récursive à gauche) spécifie une liste comme une liste suivie d'un nouvel élément ;

— la seconde alternative achève la récursion, en définissant la liste composée d'un unique chiffre.

Ce couple de règles permet de dériver des séquences de longueur arbitraire, selon des dérivations similaires à celle du tableau 6.4. On reconnaît dans celle-ci une dérivation régulière : le fragment de G_B réduit aux trois productions relatives aux nombres et d'axiome ⟨`number`⟩ définit une grammaire régulière et pourrait être représenté sous la forme d'un automate fini. Le même principe est à l'œuvre pour décrire les mots, ⟨`word`⟩, par des listes de lettres ; ou encore les listes de mots (⟨`word_list`⟩).

Une seconde figure remarquable et typique des langages de programmation apparaît dans les productions du tableau 6.5.

```
⟨group_command⟩ ::=  '{' ⟨list⟩ '}'

⟨if_command⟩ ::= if ⟨compound_list⟩ then ⟨compound_list⟩ fi
               | if ⟨compound_list⟩ then ⟨compound_list⟩ else ⟨compound_list⟩ fi
               | if ⟨compound_list⟩ then ⟨compound_list⟩ ⟨elif_clause⟩ fi

⟨elif_clause⟩ ::= elif ⟨compound_list⟩ then ⟨compound_list⟩
                | elif ⟨compound_list⟩ then ⟨compound_list⟩ else ⟨compound_list⟩
                | elif ⟨compound_list⟩ then ⟨compound_list⟩ ⟨elif_clause⟩
```

TABLE 6.5 – Constructions parenthésées

La définition du non-terminal ⟨`group_command`⟩ fait apparaître deux caractères spéciaux *appariés* { (ouvrant) et } (fermant) : dans la mesure où cette production est la seule qui introduit ces symboles, il est garanti qu'à chaque ouverture de { correspondra une fermeture de }, et ceci indépendamment du nombre et du degré d'imbrication de ces symboles. Inversement, si cette condition n'est pas satisfaite dans un programme, une erreur de syntaxe sera détectée.

Le cas des constructions conditionnelles est similaire : trois constructions sont autorisées, correspondant respectivement à la construction sans alternative (simple `if-then`), construction alternative unique (`if-then-else`), ou construction avec alternatives multiples, (`if-then-elif`...). Dans tous les cas toutefois, chaque mot-clé `if` (ouvrant) doit s'apparier avec une occurrence mot-clé `fi` (fermant), quel que soit le contenu délimité par le non-terminal ⟨`compound_list`⟩. Les constructions parenthésées apparaissent, sous de multiples formes, dans tous les langages de programmation. Sous leur forme la plus épurée, elles correspondent à des langages de la forme $a^n b^n$, qui sont des langages hors-contexte, mais pas réguliers (cf. la discussion de la section 5.2.4).

6.2 Dérivations

Nous l'avons vu, une première caractéristique des grammaires CF est la multiplicité des dérivations possibles pour un même mot. Pour « neutraliser » cette source de variabilité, nous allons poser une convention permettant d'ordonner l'application des productions. Cette convention posée, nous introduisons une représentation graphique des dérivations et définissons les notions d'ambiguïté et d'équivalence entre grammaires.

6.2.1 Dérivation gauche

Définition 6.1 (Dérivation gauche)**.** *On appelle* dérivation gauche *d'une grammaire hors-contexte G une dérivation dans laquelle chaque étape de dérivation récrit le non-terminal le plus à gauche du proto-mot courant.*

En guise d'illustration, considérons de nouveau la dérivation de *Louis boude* présentée dans le tableau 6.2 : chaque étape récrivant le non-terminal le plus à gauche, cette dérivation est bien une dérivation gauche.

Un effort supplémentaire est toutefois requis pour rendre la notion de dérivation gauche complètement opératoire. Imaginez, par exemple, que la grammaire contienne une règle de type $A \to A$. Cette règle engendre une infinité de dérivations gauches équivalentes pour toute dérivation gauche contenant A : chaque occurrence de cette production peut être répétée à volonté sans modifier le mot engendré. Pour achever de spécifier la notion de dérivation gauche, on ne considérera dans la suite que des dérivations gauches *minimales*, c'est-à-dire qui sont telles que chaque étape de la dérivation produit un proto-mot différent de ceux produits aux étapes précédentes.

On notera $A \underset{G}{\overset{L}{\Rightarrow}} u$ lorsque u dérive de A par une dérivation gauche.

De manière duale, on définit la notion de dérivation droite :

Définition 6.2 (Dérivation droite)**.** *On appelle* dérivation droite *d'une grammaire hors-contexte G une dérivation dans laquelle chaque étape de la dérivation récrit le terminal le plus à droite du proto-mot courant.*

Un premier résultat est alors énoncé dans le théorème suivant, qui nous assure qu'il est justifié de ne s'intéresser qu'aux seules dérivations gauches d'une grammaire.

Théorème 6.3. *Soit G une grammaire CF d'axiome S, et u dans* $\Sigma^\star$ *:* $S \underset{G}{\overset{\star}{\Rightarrow}} u$ *si et seulement si* $S \underset{G}{\overset{L}{\Rightarrow}} u$ *(idem pour* $S \underset{G}{\overset{R}{\Rightarrow}} u$*).*

Démonstration. Un sens de l'implication est immédiat : si un mot u se dérive par une dérivation gauche depuis S, alors $u \in L(G)$. Réciproquement, soit $u = u_1 \dots u_n$ se dérivant de S par une dérivation non-gauche, soit B le premier non-terminal non-gauche récrit dans la dérivation de u (par $B \to \beta$) et soit A le non-terminal le plus à gauche de ce proto-mot. La dérivation de u s'écrit :

$$S \underset{G}{\overset{\star}{\Rightarrow}} u_1 \dots u_i A u_j \dots u_k B \gamma \underset{G}{\Rightarrow} u_1 \dots u_i A u_j \dots u_k \beta \gamma \underset{G}{\overset{\star}{\Rightarrow}} u$$

La génération de u implique qu'à une étape ultérieure de la dérivation, le non-terminal A se récrive (éventuellement en plusieurs étapes) en : $u_{i+1} \dots u_{j-1}$. Soit alors $A \to \alpha$ la première production impliquée dans cette dérivation : il apparaît qu'en appliquant cette production juste avant $B \to \beta$, on obtient une nouvelle dérivation de u depuis S, dans laquelle A, qui est plus à gauche que B, est récrit avant lui. En itérant ce procédé, on montre que toute dérivation non-gauche de u peut se transformer de proche en proche en dérivation gauche de u, précisément ce qu'il fallait démontrer. □

Attention : si l'on peut dériver par dérivation gauche tous les mots d'un langage, on ne peut pas en dire autant des proto-mots. Il peut exister des proto-mots qui ne sont pas accessibles par une dérivation gauche : ce n'est pas gênant, dans la mesure où ces proto-mots ne permettent pas de dériver davantage de mots de la grammaire.

En utilisant ce résultat, il est possible de définir une relation d'équivalence sur l'ensemble des dérivations de G : deux dérivations sont équivalentes si elles se transforment en une même dérivation gauche. Il apparaît alors que ce qui caractérise une dérivation, ou plutôt une classe de dérivations, est partiellement [2] indépendant de l'ordre d'application des productions, et peut donc simplement être résumé par l'ensemble [3] des productions utilisées. Cet ensemble partiellement ordonné admet une expression mathématique (et visuelle) : *l'arbre de dérivation*.

6.2.2 Arbre de dérivation

Définition 6.4 (Arbre de dérivation)**.** *Un* arbre de dérivation *dans G est un arbre A tel que :*

- *tous les nœuds de A sont étiquetés par un symbole de $V = T \cup N$,*
- *la racine est étiquetée par S,*
- *si un nœud n n'est pas une feuille et porte l'étiquette X, alors $X \in N$,*
- *si $n_1, n_2, \ldots, n_k$ sont les fils de n dans A, d'étiquettes respectives $X_1, X_2, \ldots, X_k$, alors $X \to X_1 X_2 \cdots X_k$ est une production de G.*

Notons que cette définition n'impose pas que toutes les feuilles de l'arbre soient étiquetées par des terminaux : un arbre peut très bien décrire un proto-mot en cours de dérivation.

Pour passer de l'arbre de dérivation à la dérivation proprement dite, il suffit de parcourir l'arbre de dérivation en lisant les productions appliquées. La dérivation gauche s'obtient en effectuant un parcours *préfixe* de l'arbre, c'est-à-dire en visitant d'abord un nœud père, puis tous ses fils de gauche à droite. D'autres parcours fourniront d'autres dérivations équivalentes.

Un arbre de dérivation (on parle aussi d'*arbre d'analyse* d'un mot) correspondant à la production de l'énoncé *Paul mange son fromage* dans la grammaire des « dimanches » est représenté à la figure 6.6. Les numéros des nœuds sont portés en indice des étiquettes correspondantes ; la numérotation adoptée correspond à un parcours préfixe de l'arbre. Sur cette figure, on voit en particulier qu'il existe deux nœuds étiquetés *GN*, portant les indices 2 et 8.

L'arbre de dérivation représente graphiquement la *structure* associée à un mot du langage, de laquelle se déduira *l'interprétation* à lui donner.

On appelle *parsage* (en anglais *parsing*) d'un mot dans la grammaire le processus de construction du ou des arbres de dérivation pour ce mot, lorsqu'ils existent.

L'interprétation correspond à la signification d'un énoncé pour la grammaire des dimanches, à la valeur du calcul dénoté par une expression arithmétique pour une grammaire de calculs,

2. Partiellement seulement : pour qu'un non-terminal A soit dérivé, il faut qu'il est été préalablement produit par une production qui le contient dans sa partie droite.

3. En réalité un multi-ensemble, pas un ensemble au sens traditionnel : le nombre d'applications de chaque production importe.

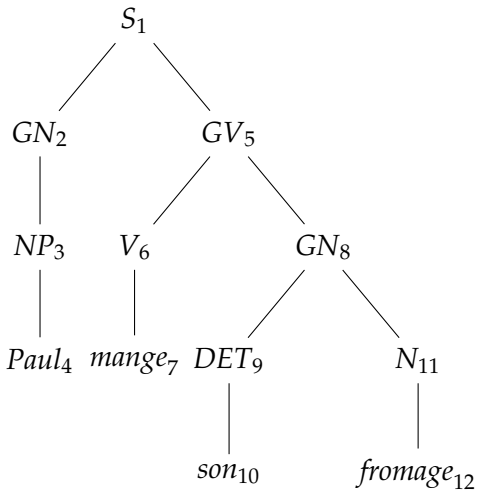


FIGURE 6.6 – L'arbre de dérivation de *Paul mange son fromage*

ou encore à la séquence d'instructions élémentaires à effectuer dans le cadre d'une grammaire représentant un langage informatique. On appelle *sémantique* d'un mot le résultat de son interprétation, et par extension *la sémantique* le domaine (formel) traitant les problèmes d'interprétation. *Sémantique* s'oppose ainsi à *syntaxique* aussi bien lorsque l'on parle de langage informatique ou de langage humain. On notera que dans les deux cas, il existe des phrases syntaxiques qui n'ont pas de sens, comme $0 = 1$, ou encore *le dessert s'ennuie Marie*.

Notons que la structure associée à un mot n'est pas nécessairement unique, comme le montre l'exemple suivant. Soit G_E une grammaire d'axiome *Sum* définissant des expressions arithmétiques simples en utilisant les productions de la grammaire 6.7.

$$\begin{aligned} \mathit{Sum} &\rightarrow \mathit{Sum} + \mathit{Sum} \mid \mathit{Number} \\ \mathit{Number} &\rightarrow \mathit{Number}\ \mathit{Digit} \mid \mathit{Digit} \\ \mathit{Digit} &\rightarrow 0 \mid 1 \mid 2 \ldots \mid 9 \end{aligned}$$

Grammaire 6.7 – Une grammaire pour les sommes

Pour cette grammaire, l'expression $3 + 5 + 1$ correspond à deux arbres d'analyse, représentés à la figure 6.8.

En remplaçant l'opérateur $+$, qui est associatif, par $-$, qui ne l'est pas, on obtiendrait non seulement deux analyses syntaxiques différentes du mot, mais également deux interprétations (résultats) différents : -3 dans un cas, -1 dans l'autre. Ceci est fâcheux.

6.2.3 Ambiguïté

Définition 6.5 (Ambiguïté d'une grammaire)**.** *Une grammaire est* ambiguë *s'il existe un mot admettant plusieurs dérivations gauches dans la grammaire.*

De manière équivalente, une grammaire est ambiguë s'il existe un mot qui admet plusieurs arbres de dérivation.

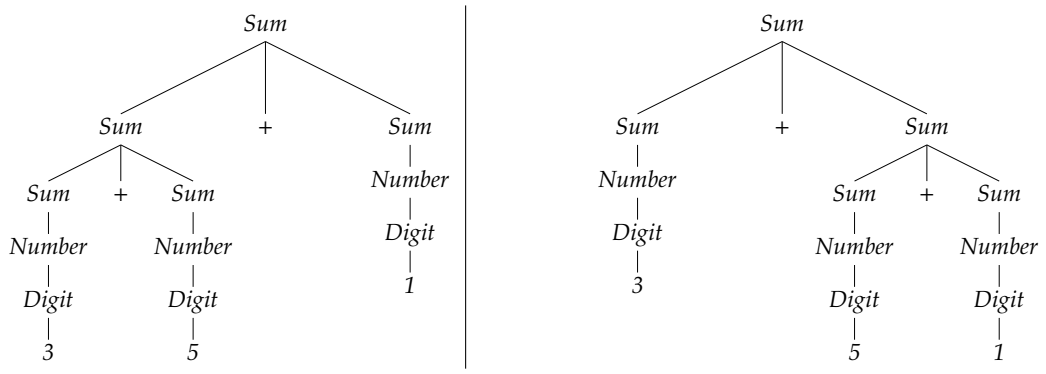


FIGURE 6.8 – Deux arbres de dérivation d'un même calcul

La grammaire 6.9 est un exemple de grammaire ambiguë.

$$\begin{aligned} S &\rightarrow ASB \mid AB \\ A &\rightarrow aA \mid a \\ B &\rightarrow bB \mid b \end{aligned}$$

Grammaire 6.9 – Une grammaire ambiguë

Le mot *aabb* admet ainsi plusieurs analyses, selon que l'on utilise ou pas la première production de S. Cette grammaire engendrant un langage régulier (lequel ?), il est toutefois élémentaire, en utilisant le procédé de transformation d'un automate en grammaire régulière, de construire une grammaire non-ambiguë qui reconnaît ce langage. L'ambiguïté est une propriété des grammaires.

Cette propriété contamine pourtant parfois même les langages :

Définition 6.6 (Ambiguïté d'un langage)**.** *Un langage hors-contexte est (intrinsèquement)* ambigu *si toutes les grammaires hors-contexte qui l'engendrent sont ambiguës.*

Un langage intrinsèquement ambigu ne peut donc être décrit par une grammaire non-ambiguë. Un exemple fameux est $L = L_1 \cup L_2 = \{a^n b^n c^m\} \cup \{a^m b^n c^n\}$, soit $\{a^n b^m c^p$, avec $m = n$ ou $m = p\}$. Décrire L_1 demande d'introduire une récursion centrale, pour apparier les a et les b ; L_2 demande également une règle exhibant une telle récursion, pour apparier les b et les c. On peut montrer que pour toute grammaire hors-contexte engendrant ce langage, tout mot de la forme $a^n b^n c^n$ aura une double interprétation, selon que l'on utilise le mécanisme d'appariement (de comptage) des a et des b ou celui qui contrôle l'appariement des b et des c.

Fort heureusement, les langages (informatiques) intrinsèquement ambigus ne courent pas les rues : lorsque l'on conçoit un nouveau langage informatique, il suffit de se prémunir contre les ambiguïtés qui peuvent conduire à des conflits d'interprétation. Ainsi, les grammaires formelles utilisées pour les langages de programmation sont-elles explicitement conçues pour limiter l'ambiguïté d'analyse.

En revanche, lorsqu'on s'intéresse au langage naturel, l'ambiguïté lexicale (un mot ayant

plusieurs catégories ou plusieurs sens) et syntaxique (un énoncé avec plusieurs interprétations) sont des phénomènes massifs et incontournables, qu'il faut donc savoir affronter avec les outils adéquats.

6.2.4 Équivalence

À travers la notion de dérivation gauche et d'arbre de dérivation, nous sommes en mesure de préciser les différentes notions d'équivalence pouvant exister entre grammaires.

Définition 6.7 (Équivalence)**.** *Deux grammaires G_1 et G_2 sont* équivalentes *si et seulement si elles engendrent le même langage (définition 5.5).*

Si, de plus, pour tout mot du langage, les arbres de dérivation dans G_1 et dans G_2 sont identiques, on dit que G_1 et G_2 sont fortement équivalentes*. Dans le cas contraire, on dit que G_1 et G_2 sont* faiblement équivalentes*.*

Ces notions sont importantes puisque, on l'a vu, c'est l'arbre de dérivation qui permet d'interpréter le sens d'un énoncé : transformer une grammaire G_1 en une autre grammaire G_2 qui reconnaît le même langage est utile, mais il est encore plus utile de pouvoir le faire *sans avoir à changer les interprétations*. À défaut, il faudra se résoudre à utiliser des mécanismes permettant de reconstruire les arbres de dérivation de la grammaire initiale à partir de ceux de la grammaire transformée.

6.3 Les langages hors-contexte

6.3.1 Le lemme de pompage

Bien que se prêtant à de multiples applications pratiques, les grammaires algébriques sont intrinsèquement limitées dans la structure des mots qu'elles engendrent. Pour mieux appréhender la nature de cette limitation, nous allons introduire quelques notions nouvelles. Si a est un symbole terminal d'un arbre de dérivation d'une grammaire G, on appelle *lignée* de a la séquence de règles utilisée pour produire a à partir de S. Chaque élément de la lignée est une paire (P, i), où P est une production, et i l'indice dans la partie droite de P de l'ancêtre de a. Considérant de nouveau la grammaire 5.1 pour le langage $a^n b^n$ et une dérivation de *aaabbb* dans cette grammaire, la lignée du second a de *aaabbb* correspond à la séquence $(S \to aSb, 2)$, $(S \to aSb, 1)$.

On dit alors qu'un symbole est *original* si tous les couples (P, i) qui constituent sa lignée sont différents. Contrairement au premier et au second a de *aaabbb*, le troisième a n'est pas original, puisque sa lignée est $(S \to aSb, 2)$, $(S \to aSb, 2)$, $(S \to ab, 1)$. Par extension, un mot est dit *original* si tous les symboles qui le composent sont originaux.

Le résultat intéressant est alors qu'une grammaire algébrique, même lorsqu'elle engendre un nombre infini de mots, ne peut produire *qu'un nombre fini de mots originaux*. En effet, puisqu'il n'y a qu'un nombre fini de productions, chacune contenant un nombre fini de symboles dans sa partie droite, chaque symbole terminal ne peut avoir qu'un nombre fini de lignées

différentes. Les symboles étant en nombre fini, il existe donc une longueur maximale pour un mot original et donc un nombre fini de mots originaux.

À quoi ressemblent alors les mots non-originaux ? Soit s un tel mot, il contient nécessairement un symbole non-original a, dont la lignée contient donc deux fois le même ancêtre A. La dérivation complète de s pourra donc s'écrire :

$$S \overset{\star}{\underset{G}{\Rightarrow}} uAy \overset{\star}{\underset{G}{\Rightarrow}} uvAxy \overset{\star}{\underset{G}{\Rightarrow}} uvwxy$$

où u, v, w, x, y sont des séquences de terminaux, la séquence w contenant le symbole a. Il est alors facile de déduire de nouveaux mots engendrés par la grammaire, en remplaçant w (qui est une dérivation possible de A) par vwx qui est une autre dérivation possible de A. Ce processus peut même être itéré, permettant de construire un nombre infini de nouveaux mots, tous non-originaux, et qui sont de la forme : uv^nwx^nz. Ces considérations nous amènent à un théorème caractérisant de manière précise cette limitation des langages algébriques.

Théorème 6.8 (Lemme de pompage pour les CFL)**.** *Si L est un langage CF, alors il existe un entier k tel que tout mot de L de longueur supérieure à k se décompose en cinq facteurs u, v, w, x, y, avec $vx \neq \varepsilon$, $|vwx| < k$ et tels que pour tout n, uv^nwx^ny est également dans L.*

Démonstration. Une partie de la démonstration de ce résultat découle des observations précédentes. En effet, si L est vide ou fini, on peut prendre pour k un majorant de la longueur d'un mot de L. Comme aucun mot de L n'est plus long que k, il est vrai que tout mot de L plus long que k satisfait le théorème 6.8.

Supposons que L est effectivement infini, alors il existe nécessairement un mot z dans $L(G)$ plus long que le plus long mot original. Ce mot étant non-original, la décomposition précédente en cinq facteurs, dont deux sont simultanément itérables, s'applique immédiatement. La première condition supplémentaire, $vx \neq \varepsilon$, dérive de la possibilité de choisir G telle qu'aucun terminal autre que S ne dérive ε (voir chapitre 9) : en considérant une dérivation minimale, on s'assure ainsi qu'au moins un terminal est produit durant la dérivation $A \overset{\star}{\Rightarrow} vAx$. La seconde condition, $|vwx| < k$, dérive de la possibilité de choisir pour vwx un facteur original et donc de taille strictement inférieure à k. □

Ce résultat est utile pour prouver qu'un langage n'est pas hors-contexte. Montrons, à titre d'illustration, que $\{a^nb^nc^n, n \geq 1\}$ n'est pas hors-contexte. Supposons qu'il le soit et considérons un mot z suffisamment long de ce langage. Décomposons z en $uvwxy$, et notons $z_n = uv^nwx^ny$. Les mots v et x ne peuvent chacun contenir qu'un seul des trois symboles de l'alphabet, sans quoi leur répétition aboutirait à des mots ne respectant pas le séquencement imposé : tous les a avant tous les b avant tous les c. Pourtant, en faisant croître n, on augmente simultanément, dans z_n, le nombre de a, de b et de c dans des proportions identiques : ceci n'est pas possible puisque seuls deux des trois symboles sont concernés par l'exponentiation de v et x. Cette contradiction prouve que ce langage n'est pas hors-contexte.

6.3.2 Opérations sur les langages hors-contexte

Dans cette section, nous étudions les propriétés de clôture pour les langages hors-contexte, d'une manière similaire à celle conduite pour les reconnaissables à la section 4.2.1.

Une première série de résultats est établie par le théorème suivant :

Théorème 6.9 (Clôture)**.** *Les langages hors-contexte sont clos pour les opérations rationnelles.*

Démonstration. Pour les trois opérations, une construction simple permet d'établir ce résultat. Si, en effet, G_1 et G_2 sont définies par : $G_1 = (N_1, \Sigma_1, P_1, S_1)$ et $G_2 = (N_2, \Sigma_2, P_2, S_2)$, on vérifie simplement que :

— $G = (N_1 \cup N_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}, S)$ engendre exactement $L_1 \cup L_2$.

— $G = (N_1 \cup N_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, P_1 \cup P_2 \cup \{S \to S_1 S_2\}, S)$ engendre exactement $L_1 L_2$.

— $G = (N_1 \cup \{S\}, \Sigma_1, P_1 \cup \{S \to SS_1, S \to \varepsilon\}, S)$ engendre exactement ${L_1}^\star$.

□

En revanche, contrairement aux langages rationnels/reconnaissables, les langages algébriques ne sont pas clos pour l'intersection. Soient en effet $L_1 = \{a^n b^n c^m\}$ et $L_2 = \{a^m b^n c^n\}$: ce sont clairement deux langages hors-contexte dont nous avons montré plus haut que l'intersection, $L = L_1 \cap L_2 = \{a^n b^n c^n\}$, n'est pas un langage hors contexte. Un corollaire (déduit par application directe de la loi de De Morgan [4]) est que les langages hors-contexte ne sont pas clos par complémentation.

Pour conclure cette section, ajoutons un résultat que nous ne démontrons pas ici :

Théorème 6.10. *L'intersection d'un langage régulier et d'un langage hors-contexte est un langage hors-contexte.*

6.3.3 Problèmes décidables et indécidables

Dans cette section, nous présentons sommairement les principaux résultats de décidabilité concernant les grammaires et langages hors-contexte. Cette panoplie de nouveaux résultats vient enrichir le seul dont nous disposons pour l'instant, à savoir que les langages hors-contexte sont récursifs et qu'en conséquence, il existe des algorithmes permettant de décider si un mot u de $\Sigma^\star$ est, ou non, engendré par une grammaire G. Nous aurons l'occasion de revenir longuement sur ces algorithmes, notamment au chapitre 7.

Nous commençons par deux résultats positifs, qui s'énonce comme :

Théorème 6.11. *Il existe un algorithme permettant de déterminer si le langage engendré par une grammaire hors-contexte est vide.*

Démonstration. La preuve exploite en fait un argument analogue à celui utilisé dans notre démonstration du lemme de pompage (théorème 6.8) : on considère un arbre de dérivation hypothétique quelconque de la grammaire, engendrant w. Supposons qu'un chemin contienne plusieurs fois le même non-terminal A (aux nœuds n_1 et n_2, le premier dominant le second). n_1 domine le facteur w_1, n_2 domine w_2 ; on peut remplacer dans w la partie correspondant à w_1 par celle correspondant à w_2. Donc, s'il existe un mot dans le langage engendré, il en existera également un qui soit tel que le même non-terminal n'apparaît jamais deux fois

4. Rappelons : $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.

dans un chemin. Dans la mesure où les non-terminaux sont en nombre fini, il existe un nombre fini de tels arbres. Il suffit alors de les énumérer et de vérifier s'il en existe un dont la frontière ne contient que des terminaux : si la réponse est oui, alors $L(G)$ est non-vide, sinon, $L(G)$ est vide. □

Théorème 6.12. *Il existe un algorithme permettant de déterminer si le langage engendré par une grammaire hors-contexte est infini.*

Démonstration. L'idée de la démonstration repose sur l'observation suivante : après élimination des productions inutiles, des productions epsilon et des cycles (voir la section 9.1), il est possible d'énumérer tous les arbres de profondeur bornée : si on ne trouve jamais deux fois le même non terminal sur une branche, le langage est fini ; sinon il est infini. □

Ces résultats sont les seuls résultats positifs pour les grammaires CF, puisqu'il est possible de prouver les résultats négatifs suivants :

- il n'existe pas d'algorithme pour décider si deux grammaires sont équivalentes (rappelons : la preuve du résultat (positif) obtenu pour les langages rationnels utilisait la clôture par intersection de ces langages) ;
- il n'existe pas d'algorithme pour décider si le langage engendré par une grammaire CF est inclus dans le langage engendré par une autre grammaire CF ;
- il n'existe pas d'algorithme pour décider si une grammaire CF engendre en fait un langage régulier ;
- il n'existe pas d'algorithme pour décider si une grammaire est ambiguë.

Munis de ces résultats, nous pouvons maintenant aborder le problème de la reconnaissance des mots engendrés par une grammaire CF, qui fait l'objet des chapitres suivants.

Chapitre 7

Introduction au parsage de grammaires hors-contexte

Dans ce chapitre, nous présentons les principales difficultés algorithmiques que pose l'analyse de grammaires hors-contexte. Il existe, en fait, trois tâches distinctes que l'on souhaiterait effectuer à l'aide d'une grammaire : la *reconnaissance*, qui correspond au calcul de l'appartenance d'un mot à un langage ; l'analyse (ou le *parsage*), qui correspond au calcul de tous les arbres d'analyse possibles pour un énoncé ; la *génération*, qui correspond à la production de tous les mots du langage décrit par une grammaire. Dans la suite de ce chapitre, nous ne nous intéresserons qu'aux deux premières de ces tâches.

Les langages hors-contexte étant une sous classe des langages récursifs, nous savons qu'il existe des algorithmes permettant d'accomplir la tâche de reconnaissance. En fait, il en existe de multiples, qui, essentiellement se ramènent tous à deux grands types : les analyseurs *ascendants* et les analyseurs *descendants*. Comme expliqué dans la suite, toute la difficulté pour les analyseurs consiste à affronter le non-déterminisme inhérent aux langages hors-contexte de manière à :

— éviter de perdre son temps dans des impasses ;

— éviter de refaire deux fois les mêmes calculs.

Ce chapitre est organisé comme suit : dans la section 7.1, nous présentons l'espace de recherche du parsage ; nous présentons ensuite les stratégies ascendantes dans la section 7.2 et descendantes en section 7.3, ainsi que les problèmes que pose la mise en œuvre de telles stratégies. Cette étude des analyseurs se poursuit au chapitre 8, où nous présentons des analyseurs déterministes, utilisables pour certains types de grammaires ; ainsi qu'au chapitre 9, où nous étudions des techniques de normalisation des grammaires, visant à simplifier l'analyse, ou à se prémunir contre des configurations indésirables.

Une référence extrêmement riche et complète pour aborder les questions de parsage est Grune et Jacob (1990).

7.1 Graphe de recherche

Un point de vue général sur la question de la reconnaissance est donné par la considération suivante. Comme toute relation binaire, la relation binaire $\underset{G}{\Rightarrow}$ sur les séquences de $(N \cup \Sigma)^\star$ se représente par un graphe dont les sommets sont les séquences de $(N \cup \Sigma)^\star$ et dans lequel un arc de α vers β indique que $\alpha \underset{G}{\Rightarrow} \beta$. On l'appelle le *graphe de la grammaire* G. Ce graphe est bien sûr infini dès lors que $L(G)$ est infini ; il est en revanche *localement fini*, signifiant que tout sommet a un nombre fini de voisins.

La question à laquelle un algorithme de reconnaissance doit répondre est alors la suivante : existe-t-il dans ce graphe un chemin du nœud S vers le nœud u ? Un premier indice : s'il en existe un, il en existe nécessairement plusieurs, correspondant à plusieurs dérivations différentes réductibles à une même dérivation gauche. Il n'est donc pas nécessaire d'explorer tout le graphe.

Pour répondre plus complètement à cette question, il existe deux grandes manières de procéder : construire et explorer de proche en proche les voisins de S en espérant rencontrer u : ce sont les approches dites *descendantes* ; ou bien inversement construire et explorer les voisins [1] de u, en essayant de « remonter » vers S : on appelle ces approches *ascendantes*. Dans les deux cas, si l'on prend soin, au cours de l'exploration, de se rappeler des différentes étapes du chemin, on sera alors à même de reconstruire un arbre de dérivation de u.

Ces deux groupes d'approches, ascendantes et descendantes, sont considérés dans les sections qui suivent, dans lesquelles on essayera également de suggérer que ce problème admet une implantation algorithmique polynomiale : dans tous les cas, la reconnaissance de u demande un nombre d'étapes borné par $k|u|^p$ pour un certain p fixé qui est indépendant de l'entrée.

Le second problème intéressant est celui de l'analyse, c'est-à-dire de la construction de tous les arbres de dérivation de u dans G. Pour le résoudre, il importe non seulement de déterminer non pas un chemin, mais tous les chemins [2] et les arbres de dérivation correspondants. Nous le verrons, cette recherche exhaustive peut demander, dans les cas où la grammaire est ambiguë, un temps de traitement exponentiel par rapport à la longueur de u. En effet, dans une grammaire ambiguë, il peut exister un nombre exponentiel d'arbres de dérivation, dont l'énumération (explicite) demandera nécessairement un temps de traitement exponentiel.

7.2 Reconnaissance ascendante

Supposons qu'étant donnée la grammaire 6.1 des repas dominicaux, nous soyons confrontés à l'énoncé : *le fils mange sa soupe*. Comment faire alors pour :

— vérifier que ce « mot » appartient au langage engendré par la grammaire ?

— lui assigner, dans le cas où la réponse est positive, une (ou plusieurs) structure(s) arborée(s) ?

1. En renversant l'orientation des arcs.
2. En fait, seulement ceux qui correspondent à une dérivation gauche.

Une première idée d'exploration nous est donnée par l'algorithme suivant, qui, partant des terminaux du mot d'entrée, va chercher à « inverser » les règles de production pour parvenir à récrire le symbole initial de la grammaire : chaque étape de l'algorithme vise à réduire la taille du proto-mot courant, en substituant la partie droite d'une production par sa partie gauche. Dans l'algorithme présenté ci-dessous, cette réduction se fait par la gauche : on cherche à récrire le proto-mot courant en commençant par les symboles qui se trouvent le plus à gauche : à chaque étape, il s'agit donc d'identifier, en commençant par la gauche, une partie droite de règle. Lorsqu'une telle partie droite est trouvée, on la remplace par la partie gauche correspondante et on continue. Si on ne peut trouver une telle partie droite, il faut remettre en cause une règle précédemment appliquée (par retour en arrière) et reprendre la recherche en explorant un autre chemin. Cette recherche s'arrête lorsque le proto-mot courante ne contient plus, comme unique symbole, que l'axiome S.

Une trace de l'exécution de cet algorithme est donnée ci-dessous, après que le proto-mot courant α a été initialisé avec *le fils mange sa soupe* :

1. on remplace *le* par la partie gauche de la règle $le \rightarrow DET$. α = *DET fils mange sa soupe*.
2. on essaie de remplacer *DET* en le trouvant comme partie droite. Échec. Idem pour *DET fils*, *DET fils mange*...
3. on récrit *fils* : *N*. α = *DET N mange sa soupe*.
4. on essaie de remplacer *DET* en le trouvant comme partie droite. Échec.
5. on récrit *DET N* : *GN*. α = *GN mange sa soupe*.
6. on essaie de remplacer *GN* en le trouvant comme partie droite. Échec. Idem pour *GN mange*, *GN mange sa*...
7. on récrit *mange* : *V*. α = *GN V sa soupe*.
8. on essaie de remplacer *GN* en le trouvant comme partie droite. Échec. Idem pour *GN V*, *GN V sa*...
9. on récrit *V* : *GV*. α = *GN GV sa soupe*.
10. on essaie de remplacer *GN* en le trouvant comme partie droite. Échec.
11. on récrit *GN GV* : *S*. α = *S sa soupe*.
12. on essaie de remplacer *S* en le trouvant comme partie droite. Échec. Idem pour *S sa*, *S sa soupe*.
13. on récrit *sa* : *DET*. Résultat : *S DET soupe*.
14. on essaie de remplacer *S* en le trouvant comme partie droite. Échec. Idem pour *S DET*, *S DET soupe*.
15. on essaie de remplacer *DET* en le trouvant comme partie droite. Échec. Idem pour *DET soupe*.
16. on remplace *soupe* par *N*. α = *S DET N*.
17. on essaie de remplacer *S* en le trouvant comme partie droite. Échec. Idem pour *S DET*, *S DET GN*.
18. on essaie de remplacer *DET* en le trouvant comme partie droite. Échec.
19. on remplace *DET N* par *GN*. α = *S GN*.
20. on essaie de remplacer *S* en le trouvant comme partie droite. Échec.

21. on essaie de remplacer *S GN* en le trouvant comme partie droite. Échec. On est bloqué : on fait un retour arrière jusqu'en 11. α = *GN GV sa soupe*.

... on va d'abord parcourir entièrement le choix après *GN* jusqu'à *GN GV GN*. Nouvelle impasse. Donc on revient en arrière en 8. Et on recommence...

... tôt ou tard, on devrait finalement parvenir à une bonne solution.

Cette stratégie est mise en œuvre par l'algorithme 7.1.

```
// La fonction principale est bulrp: bottom-up left-right parsing.
// α contient le proto-mot courant.
Function bulrp(α) : bool is
    if α = S then return true ;
    for i := 1 to |α| do
        for j := i to |α| do
            if ∃A → α_i ... α_j then
                if bulrp(α_1 ... α_{i-1} A α_{j+1} ... α_n) = true then
                    return true
    return false ;
```

Algorithme 7.1 – Parsage ascendant en profondeur d'abord

La procédure détaillée dans l'algorithme 7.1 correspond à la mise en œuvre d'une stratégie *ascendante* (en anglais *bottom-up*), signifiant que l'on cherche à construire l'arbre de dérivation depuis le bas (les feuilles) vers le haut (la racine de l'arbre), donc depuis les symboles de Σ vers S. La stratégie mise en œuvre par l'algorithme 7.1 consiste à explorer le graphe de la grammaire *en profondeur d'abord*. Chaque branche est explorée de manière successive ; à chaque échec (lorsque toutes les parties droites possibles ont été successivement envisagées), les choix précédents sont remis en cause et des chemins alternatifs sont visités. Seules quelques branches de cet arbre mèneront finalement à une solution. Notez qu'il est possible d'améliorer un peu `bulrp` pour ne considérer que des dérivations droites. Comment faudrait-il modifier cette fonction pour opérer une telle optimisation ?

Comme en témoigne l'aspect un peu répétitif de la simulation précédente, cette stratégie conduit à répéter de nombreuses fois les mêmes tests, et à reconstruire en de multiples occasions les mêmes constituants (voir la figure 7.2).

L'activité principale de ce type d'algorithme consiste à chercher, dans le proto-mot courant, une séquence pouvant correspondre à la partie droite d'une règle. L'alternative que cette stratégie d'analyse conduit à évaluer consiste à choisir entre remplacer une partie droite identifiée par le non-terminal correspondant (étape de *réduction*, en anglais *reduce*), ou bien différer la réduction et essayer d'étendre la partie droite en considérant des symboles supplémentaires (étape *d'extension*, on dit aussi *décalage*, en anglais *shift*). Ainsi les étapes de réduction 9 et 11 dans l'exemple précédent conduisent-elles à une impasse, amenant à les remplacer par des étapes d'extension, qui vont permettre dans notre cas de construire l'objet direct du verbe avant d'entreprendre les bonnes réductions. Comme on peut le voir dans cet exemple, l'efficacité de l'approche repose sur la capacité de l'analyseur à prendre les bonnes décisions (*shift* ou *reduce*) au bon moment. À cet effet, il est possible de pré-calculer un certain nombre de tables auxiliaires, qui permettront de mémoriser le fait que par exemple, *mange* attend un complément, et que donc il ne faut pas réduire V avant d'avoir trouvé ce complément. Cette intuition sera formalisée dans le chapitre suivant, à la section 8.2.

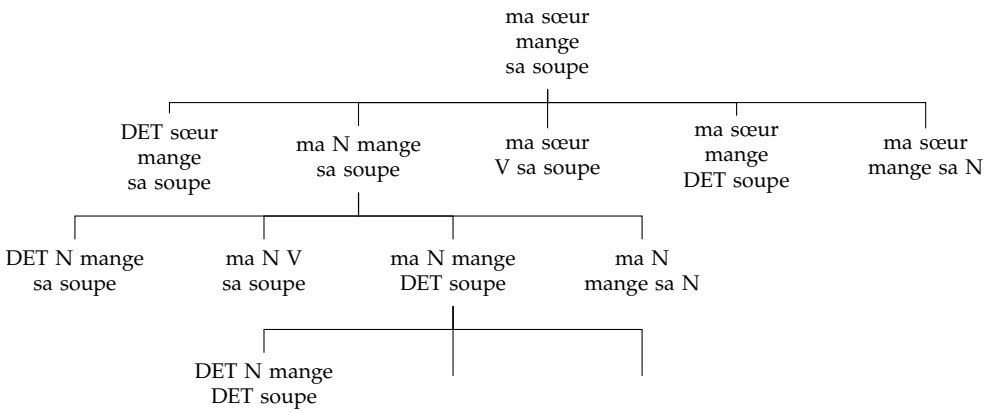


FIGURE 7.2 – Recherche ascendante

La stratégie naïve conduit à une analyse exponentielle en fonction de la longueur de l'entrée. Fort heureusement, il existe des techniques éprouvées, consistant à stocker les résultats intermédiaires dans des tableaux, qui permettent d'aboutir à une complexité polynomiale.

Notez, pour conclure, que dans notre exemple, la stratégie mise en œuvre termine, ce qui ne serait pas le cas si notre grammaire était moins « propre ». L'analyse ascendante est prise en défaut par des règles du type $X \to X$, ou par des cycles de production de type $X \to Y, Y \to X$, qui conduisent à effectuer (indéfiniment !) des séries de réductions « stériles », c'est-à-dire des réductions qui laissent inchangée la longueur du proto-mot courant. Des problèmes sérieux se posent également avec les productions ε, qui sont susceptibles de s'appliquer à toutes les étapes de l'algorithme. La « grammaire des dimanches » n'en comporte pas, mais de telles règles ne sont pas rares dans les grammaires réelles.

Avant de montrer qu'il est, en théorie, possible de se prémunir contre de telles horreurs (voir la section 9.1), il est important de noter que cette stratégie s'accompagne de multiples variantes, consistant par exemple à effectuer l'exploration *en largeur d'abord*, ou de droite à gauche, ou simultanément dans les deux sens...

Donnons l'intuition de ce que donnerait une exploration en largeur d'abord : d'une manière générale, ce type de recherche correspond à la poursuite en parallèle d'un ensemble de chemins possibles. L'algorithme est initialisé avec un seul proto-mot, qui est l'énoncé à analyser. À chaque étape, un nouvel ensemble de proto-mots est considéré, auxquels sont appliquées toutes les réductions possibles, donnant lieu à un nouvel ensemble de proto-mots. Si cet ensemble contient S, on arrête ; sinon la procédure est répétée. En guise d'application, écrivez sur le modèle de l'algorithme précédent une procédure mettant en application cette idée.

7.3 Reconnaissance descendante

Une seconde stratégie de recherche consiste à procéder de manière *descendante*, c'est-à-dire à partir de l'axiome de la grammaire pour tenter d'engendrer l'énoncé à analyser. Les impasses

de cette recherche sont détectées en confrontant des préfixes terminaux du proto-mot courant avec les terminaux de l'énoncé à analyser. Comme précédemment, dans une recherche en profondeur d'abord, les situations d'échecs conduisent à remettre en cause les choix précédemment effectués ; une recherche en largeur d'abord conduit à développer simultanément plusieurs proto-mots. La recherche s'arrête lorsque le proto-mot dérivé depuis S est identique à la séquence qu'il fallait analyser.

Simulons, par exemple, le fonctionnement d'un analyseur en profondeur d'abord :

1. S
2. *GN GV*
3. *DET N GV*
4. *la N GV*. Échec : l'entrée commence par *le*, pas par *la*. On revient à *DET N GV*
5. *le N GV*.
6. *le fille GV*. Nouvel échec α = *le N GV*

...

7. *le fils GV*.

... *le fils V*.

... *le fils boude*. Échec α = *le fils V*

... *le fils s'ennuie*. Échec α = *le fils V*

... *le fils mange*. Il reste des terminaux non-analysés dans l'entrée, mais le proto-mot ne contient plus de non-terminaux. Échec et retour en 7.

...

Algorithmiquement, cette stratégie d'analyse correspond à une boucle dans laquelle l'analyseur examine le non-terminal le plus à gauche du proto-mot courant et essaie de le dériver tout en restant compatible avec l'énoncé d'entrée. Cet analyseur construit donc des dérivations gauches. Son activité alterne des étapes de *prédiction* (en anglais *prediction*) d'un symbole terminal ou non-terminal et des étapes *d'appariement* (en anglais *matching*) des terminaux prédits avec les mots de l'entrée. Pour cette raison, le parsage descendant est souvent qualifié de *prédictif*. Le pseudo-code d'un tel analyseur est donné dans l'algorithme 7.3.

```
// La fonction principale est tdlrp: top-down left-right parsing.
// α est le proto-mot courant, u l'entrée à reconnaître.
Function tdlrp(α, u) : bool is
    if α = u then return true ;
    α = u_1 ... u_k A γ ;
    while ∃A → u_{k+1}...u_{k+l} δ avec δ = ε ou δ = A ... do
        if tdlrp(u_1 ... u_{k+l} δ γ) = true then return true ;
    return false
```

Algorithme 7.3 – Parsage descendant en profondeur d'abord

Une implantation classique de cette stratégie représente α sous la forme d'une *pile* de laquelle est exclu le préfixe terminal $u_1 \dots u_k$: à chaque étape il s'agit de dépiler le non-terminal A en tête de la pile, et d'empiler à la place un suffixe de la partie droite correspondante (β),

après avoir vérifié que l'(éventuel) préfixe terminal de β ($u_{uk+1} \dots u_k + l$) était effectivement compatible avec la partie non-encore analysée de u.

Pour que cette stratégie soit efficace, il est possible de pré-calculer dans des tables l'ensemble des terminaux qui peuvent débuter l'expansion d'un non-terminal. En effet, considérons de nouveau la grammaire 6.1 : cette grammaire possède également les productions suivantes, $GN \rightarrow NP$, $NP \rightarrow Louis$, $NP \rightarrow Paul$, correspondant à des noms propres. À l'étape 2 du parcours précédent, l'algorithme pourrait être (inutilement) conduit à prédire un constituant de catégorie NP, alors même qu'aucun nom propre ne peut débuter par *le*, qui est le premier terminal de l'énoncé à analyser. Cette intuition est formalisée dans les analyseurs LL, qui sont présentés dans la section 8.1.

Dans l'optique d'une stratégie de reconnaissance descendante, notre grammaire possède un vice de forme manifeste, à savoir la production $GN \rightarrow GN \quad GNP$, qui est récursive à gauche : lorsque cette production est considérée, elle insère en tête du proto-mot courant un nouveau symbole GN, qui à son tour peut être remplacé par un nouveau GN, conduisant à un accroissement indéfini de α. Dans le cas présent, la récursivité gauche est relativement simple à éliminer. Il existe des configurations plus complexes, dans lesquelles cette récursivité gauche résulte non pas d'une règle unique mais d'une séquence de règles, comme dans : $S \rightarrow A\alpha$, $A \rightarrow S\beta$. On trouvera au chapitre 9 une présentation des algorithmes permettant d'éliminer ce type de récursion.

Notons, pour finir, que nous avons esquissé ci-dessus les principales étapes d'une stratégie descendante en profondeur d'abord. Il est tout aussi possible d'envisager une stratégie en largeur d'abord, consistant à conduire de front l'examen de plusieurs proto-mots. L'écriture d'un algorithme mettant en œuvre cette stratégie est laissée en exercice.

7.4 Conclusion provisoire

Nous avons, dans ce chapitre, défini les premières notions nécessaires à l'étude des algorithmes d'analyse pour les grammaires algébriques. Les points de vue sur ces algorithmes diffèrent très sensiblement suivant les domaines d'application :

— dans le cas des langages informatiques, les mots (des programmes, des documents structurés) sont longs, voire très longs ; l'ambiguïté est à proscrire, pouvant conduire à des conflits d'interprétation. Les algorithmes de vérification syntaxique doivent donc avoir une faible complexité (idéalement une complexité linéaire en fonction de la taille de l'entrée) ; il suffit par ailleurs en général de produire une analyse et une seule. Les algorithmes qui répondent à ce cahier des charges sont présentés au chapitre 8.

— dans le cas des langues naturelles, l'ambiguïté est une des données du problème, qu'il faut savoir contrôler. Les mots étant courts, une complexité supérieure (polynomiale, de faible degré) pour l'analyse est supportable. On peut montrer qu'à l'aide de techniques de programmation dynamique, consistant à sauvegarder dans des tables les fragments d'analyse réalisés, il est possible de parvenir à une complexité en $O(n^3)$ pour construire d'un seul coup tous les arbres d'analyse. Leur énumération exhaustive conserve une complexité exponentielle.

Quel que soit le contexte, les algorithmes de parsage bénéficient toujours d'une étape de

prétraitement et de normalisation de la grammaire, visant en particulier à éviter les configurations problématiques (cycles et productions ε pour les analyseurs ascendants ; récursions gauches pour les analyseurs descendants). Ces prétraitements sont présentés dans le chapitre 9.

Chapitre 8

Introduction aux analyseurs déterministes

Dans ce chapitre, nous présentons deux stratégies d'analyse pour les grammaires hors-contexte, qui toutes les deux visent à produire des analyseurs déterministes. Comme présenté au chapitre 7, le parsage peut être vu comme l'exploration d'un graphe de recherche. L'exploration est rendue coûteuse par l'existence de ramifications dans le graphe, correspondant à des chemins alternatifs : ceci se produit, pour les analyseurs descendants, lorsque le terminal le plus à gauche du proto-mot courant se récrit de plusieurs manières différentes (voir la section 7.1). De telles alternatives nécessitent de mettre en œuvre des stratégies de retour arrière (recherche en profondeur d'abord) ou de pouvoir explorer plusieurs chemins en parallèle (recherche en largeur d'abord).

Les stratégies d'analyse présentées dans ce chapitre visent à éviter au maximum les circonstances dans lesquelles l'analyseur explore inutilement une branche de l'arbre de recherche (en profondeur d'abord) ou dans lesquelles il conserve inutilement une analyse dans la liste des analyses alternatives (en largeur d'abord). À cet effet, ces stratégies mettent en œuvre des techniques de pré-traitement de la grammaire, cherchant à identifier par avance les productions qui conduiront à des chemins alternatifs ; ainsi que des contrôles permettant de choisir sans hésiter la bonne ramification. Lorsque de tels contrôles existent, il devient possible de construire des analyseurs *déterministes*, c'est-à-dire des analyseurs capables de toujours faire le bon choix. Ces analyseurs sont algorithmiquement efficaces, en ce sens qu'ils conduisent à une complexité d'analyse *linéaire* par rapport à la longueur de l'entrée.

La section 8.1 présente la mise en application de ce programme pour les stratégies d'analyse descendantes, conduisant à la famille d'analyseurs prédictifs LL. La section 8.2 s'intéresse à la construction d'analyseurs ascendants, connus sous le nom d'analyseurs LR, et présente les analyseurs les plus simples de cette famille.

8.1 Analyseurs LL

Nous commençons par étudier les grammaires qui se prêtent le mieux à des analyses descendantes et dérivons un premier algorithme d'analyse pour les grammaires SLL(1). Nous généralisons ensuite cette approche en introduisant les grammaires LL(1), ainsi que les algorithmes d'analyse correspondants.

8.1.1 Une intuition simple

Comme expliqué à la section 7.3, les analyseurs descendants essayent de construire de proche en proche une dérivation gauche du mot u à analyser : partant de S, il s'agit d'aboutir, par des récritures successives du (ou des) proto-mots(s) courant(s), à u. À chaque étape de l'algorithme, le non-terminal A le plus à gauche est récrit par $A \to \alpha$, conduisant à l'exploration d'une nouvelle branche dans l'arbre de recherche. Deux cas de figure sont alors possibles :

(i) soit α débute par au moins un terminal : dans ce cas on peut *immédiatement* vérifier si le proto-mot courant est compatible avec le mot à analyser et, le cas échéant, revenir sur le choix de la production $A \to \alpha$. C'est ce que nous avons appelé la phase d'*appariement*.

(ii) soit α débute par un non-terminal (ou est vide) : dans ce cas, il faudra continuer de développer le proto-mot courant pour valider la production choisie et donc *différer* la validation de ce choix.

Une première manière de simplifier la tâche des analyseurs consiste à éviter d'utiliser dans la grammaire les productions de type point ii : ainsi l'analyseur pourra toujours immédiatement vérifier le bien-fondé des choix effectués, lui évitant de partir dans l'exploration d'un cul-de-sac.

Ceci n'élimine toutefois pas toute source de non-déterminisme : s'il existe un non-terminal A apparaissant dans deux productions de type point i $A \to a\alpha$ et $A \to a\beta$, alors il faudra quand même considérer (en série ou en parallèle) plusieurs chemins alternatifs.

Considérons, à titre d'exemple, le fragment présenté par la grammaire 8.1. Elle présente la particularité de ne contenir que des productions de type point i ; de plus, les coins gauches des productions associées à ce terminal sont toutes différentes.

$$
\begin{aligned}
S &\to \mathit{if}\ (\ B\ )\ \mathit{then}\ \{\ I\ \} \\
S &\to \mathit{while}\ (\ B\ )\ \{\ I\ \} \\
S &\to \mathit{do}\ \{\ I\ \}\ \mathit{until}\ (\ B\ ) \\
B &\to \mathit{false} \mid \mathit{true} \\
I &\to \ldots
\end{aligned}
$$

Grammaire 8.1 – Fragments d'un langage de commande

Il est clair qu'un analyseur très simple peut être envisagé, au moins pour explorer sans hésitation (on dit aussi : *déterministiquement*) les dérivations de S : soit le premier symbole à apparier est le mot-clé *if*, et il faut appliquer la première production ; soit c'est *while* et il faut appliquer la seconde ; soit c'est *do* et il faut appliquer la troisième. Tous les autres cas de figure correspondent à des situations d'erreur.

Si l'on suppose que tous les non-terminaux de la grammaire possèdent la même propriété que S dans la grammaire 8.1, alors c'est l'intégralité de l'analyse qui pourra être conduite de manière déterministe par l'algorithme suivant : on initialise le proto-mot courant avec S et à chaque étape, on consulte les productions du non-terminal A le plus à gauche, en examinant s'il en existe une dont le coin gauche est le premier symbole u_i non encore apparié. Les conditions précédentes nous assurant qu'il existe au plus une telle production, deux configurations sont possibles :

— il existe une production $A \to u_i\alpha$, et on l'utilise ; le prochain symbole à apparier devient u_{i+1} ;

— il n'en existe pas : le mot à analyser n'est pas engendré par la grammaire.

Cet algorithme se termine lorsque l'on a éliminé tous les non-terminaux du proto-mot : on vérifie alors que les terminaux non-encore appariés dans l'entrée correspondent bien au suffixe du mot engendré : si c'est le cas, l'analyse a réussi. Illustrons cet algorithme en utilisant la grammaire 8.2 ; le tableau 8.3 décrit pas-à-pas les étapes de l'analyse de *aacddcbb* par cette grammaire.

$$\begin{aligned} S &\to aSb \\ S &\to cC \\ C &\to dC \\ C &\to c \end{aligned}$$

Grammaire 8.2 – Une grammaire LL(1) simple

itération	apparié	à apparier	prédit	règle
0	ε	*aacddcbb*	S	$S \to aSb$
1	*a*	*acddcbb*	Sb	$S \to aSb$
2	*aa*	*cddcbb*	Sbb	$S \to cC$
3	*aac*	*ddcbb*	Cbb	$C \to dC$
4	*aacd*	*dcbb*	Cbb	$C \to dC$
5	*aacdd*	*cbb*	Cbb	$C \to c$
6	*aacddc*	*bb*	*bb*	

À l'itération 3 de l'algorithme, le proto-mot courant commande la récriture de C : le symbole à apparier étant un d, le seul choix possible est d'appliquer la production $C \to dC$; ceci a pour effet de produire un nouveau proto-mot et de décaler vers la droite le symbole à apparier.

TABLE 8.3 – Étapes de l'analyse de aacddcbb

L'application de l'algorithme d'analyse précédent demande des consultations répétitives de la grammaire pour trouver quelle règle appliquer. Il est possible de construire à l'avance une table enregistrant les productions possibles. Cette table, dite *table d'analyse prédictive*, contient pour chaque non-terminal A et pour chaque symbole d'entrée a l'indice de la production à utiliser lorsque A est le plus à gauche dans le proto-mot courant, alors que l'on veut apparier a. Cette table est en fait exactement analogue à la table de transition d'un automate déterministe : pour chaque non-terminal (état) et chaque symbole d'entrée, elle indique la

production (la transition) à utiliser. Pour la grammaire 8.2, cette matrice est reproduite dans le tableau 8.4.

	a	b	c	d
S	$a\ S\ b$		$c\ C$	
C			c	$d\ C$

Si S est prédit et a le premier terminal non apparié, alors récrire S par aSb. Les cases vides sont des situations d'échec.

TABLE 8.4 – Table d'analyse prédictive

Cette analogie[1] suggère que la stratégie d'analyse appliquée ci-dessus a une complexité linéaire : un mot de longueur k ne demandera jamais un nombre d'étapes supérieur à k : le déterminisme rend l'analyse algorithmiquement efficace, ce qui justifie les efforts pour rechercher des algorithmes déterministes.

Les grammaires présentant les deux propriétés précédentes sont appelées des grammaires LL(1) *simples* et sont définies formellement par :

Définition 8.1 (Grammaire SLL(1))**.** *Une grammaire* $G = (N, \Sigma, S, P)$ *est une grammaire* SLL(1) *si et seulement si :*

(i) $\forall (A \to \alpha) \in P, \exists a \in \Sigma, \alpha = a\beta$

(ii) $\forall A \in N, (A \to a_1\alpha_1) \in P$ *et* $(A \to a_2\alpha_2) \in P$ *et* $a_1\alpha_1 \neq a_2\alpha_2 \Rightarrow a_1 \neq a_2$

Pourquoi cette terminologie ? Parce qu'elles permettent directement de mettre en œuvre des analyseurs construisant de manière déterministe de gauche à droite (*Left-to-right*) des dérivations gauches (*Leftmost derivation*) avec un regard avant (en anglais *Lookahead*) de 1 seul symbole. Dans la mesure où la forme de la grammaire rend immédiate cette construction, ces grammaires sont additionnellement qualifiées de *simples*, d'où la terminologie *Simple LL*.

La question qui se pose alors est la suivante : tout langage hors-contexte est-il susceptible d'être analysé par une telle technique ? En d'autres termes, est-il possible de construire une grammaire SLL(1) pour tout langage algébrique ? La réponse est malheureusement non : il existe des langages intrinsèquement ambigus (voir la section 6.2.3) pour lesquels il est impossible de construire un analyseur déterministe. Il est, en revanche, possible de transformer toute grammaire en une grammaire satisfaisant la propriété (i) par une suite de transformations aboutissant à la forme normale de Greibach (voir la section 9.2.2). Cette transformation conduisant toutefois à des grammaires (et donc à des dérivations) très éloignées de la grammaire initiale, il est tentant de chercher à généraliser les stratégies développées dans cette section, de manière à pouvoir les appliquer à des grammaires moins contraintes. Cette démarche est poursuivie dans la suite de ce chapitre avec l'introduction des grammaires LL.

1. C'est un peu plus qu'une analogie : une grammaire régulière, par définition, vérifie la propriété (i) ci-dessus ; si elle vérifie de plus la propriété (ii), alors vous vérifierez que l'algorithme de transformation de la grammaire en automate décrit à la section 5.2.4 aboutit en fait à un automate *déterministe*.

8.1.2 Grammaires LL(1)

La clé du succès des analyseurs pour les grammaires SLL(1) est la possibilité de contrôler sans attendre la validité d'une prédiction, rendue possible par la propriété que tout non-terminal récrit comme premier symbole un symbole terminal.

Considérons alors une grammaire qui n'a pas cette propriété, telle que celle de la grammaire 8.5, qui engendre des formules arithmétiques.

$$\begin{aligned} S &\to S + F \mid F \\ F &\to F * T \mid T \\ T &\to (S) \mid D \\ D &\to 0 \mid \cdots \mid 9 \mid 0D \mid \cdots \mid 9D \end{aligned}$$

Grammaire 8.5 – Une grammaire pour les expressions arithmétiques

Le non-terminal F de la grammaire 8.5, par exemple, se récrit toujours en un non-terminal. En examinant les dérivations de ce terminal, on constate toutefois que son élimination d'une dérivation gauche conduit toujours à avoir comme nouveau non-terminal le plus à gauche un T, par des dérivations de la forme :

$$F \underset{A}{\Rightarrow} F * T \overset{\star}{\underset{A}{\Rightarrow}} F * T * T * \cdots * T \underset{A}{\Rightarrow} T * T * \ldots * T$$

Examinons alors les productions de T : l'une commence par récrire le terminal '(' : cela signifie donc que T, et donc F aussi, dérive des proto-mots de type $(\alpha$. T peut également se récrire D ; ce non-terminal respecte la contrainte précédente, nous garantissant qu'il récrit toujours en premier un chiffre entre 0 et 9. On en déduit que les dérivations de T débutent soit par '(', soit par un chiffre ; il en va alors de même pour les dérivations de F. À quoi nous sert cette information ? D'une part, à détecter une erreur dès que F apparaît en tête d'un proto-mot alors qu'un autre terminal (par exemple + ou $*$) doit être apparié. Mais on peut faire bien mieux encore : supposons en effet que l'on cherche à dériver T. Deux productions sont disponibles : le calcul précédent nous fournit un moyen infaillible de choisir entre elles : si le symbole à apparier est '(', il faut choisir $T \to (S)$; sinon si c'est un chiffre, il faut choisir $T \to D$.

En d'autres termes, le calcul des terminaux qui sont susceptibles d'initier une dérivation gauche permet de sélectionner au plus tôt entre productions ayant une même partie gauche, ainsi que d'anticiper sur l'application erronée de productions. En fait, ces symboles jouent le même rôle que les terminaux apparaissant en coin gauche des productions de grammaire SLL(1) et leur calcul est donc essentiel pour anticiper sur les bonnes prédictions.

L'exemple précédent suggère une approche récursive pour effectuer ce calcul, consistant à examiner récursivement les coins gauches des productions de la grammaire jusqu'à tomber sur un coin gauche terminal. Formalisons maintenant cette intuition.

8.1.3 NULL, FIRST et FOLLOW

FIRST Pour commencer, définissons l'ensemble FIRST(A) des symboles terminaux pouvant apparaître en tête d'une dérivation gauche de A, soit :

Définition 8.2 (FIRST). *Soit $G = (N, \Sigma, S, P)$ une grammaire CF et A un élément de N. On appelle* FIRST(A) *le sous-ensemble de Σ défini par :*

$$\text{FIRST}(A) = \{a \in \Sigma, \exists \alpha \in (N \cup \Sigma), A \overset{\star}{\underset{G}{\Rightarrow}} a\alpha\}$$

Ainsi, dans la grammaire 8.5, on a :

$$\text{FIRST}(F) = \text{FIRST}(T) = \{(, 0, 1, \ldots, 9\}$$

Comment construire automatiquement cet ensemble ? Une approche naïve consisterait à implanter une formule récursive du type :

$$\text{FIRST}(A) = \bigcup_{X, A \to X\alpha \in P} \text{FIRST}(X)$$

Cette approche se heurte toutefois à deux difficultés :

— les productions récursives à gauche, qui sont du type du type $A \to A\alpha$; ces productions sont hautement nocives pour les analyseurs descendants (voir la section 7.3) et il faudra dans tous les cas s'en débarrasser. Des techniques idoines pour ce faire sont présentées à la section 9.2.2 ; dans la suite de l'exposé on considérera qu'il n'y a plus de production (ni de chaîne de productions) récursive à gauche ;

— les productions $A \to X\alpha$ pour lesquelles le non-terminal X est tel que $X \overset{\star}{\underset{G}{\Rightarrow}} \varepsilon$: dans ce cas, il faudra, pour calculer correctement FIRST(A), tenir compte du fait que le coin gauche X des A-productions peut dériver le mot vide.
Ces non-terminaux soulèvent toutefois un problème nouveau : supposons en effet que P contienne une règle de type $A \to \varepsilon$, et considérons l'état d'un analyseur descendant tentant de faire des prédictions à partir d'un proto-mot de la forme $uA\alpha$. Le non-terminal A pouvant ne dériver aucun symbole, il paraît difficile de contrôler les applications erronées de la production $A \to \varepsilon$ en regardant simplement FIRST(A) et le symbole à apparier. Notons qu'il en irait de même si l'on avait $A \overset{\star}{\underset{G}{\Rightarrow}} \varepsilon$. Comment alors anticiper sur les dérivations erronées de tels non-terminaux et préserver le déterminisme de la recherche ?

Pour résoudre les problèmes causés par l'existence de non-terminaux engendrant le mot vide, introduisons deux nouveaux ensembles : NULL et FOLLOW.

NULL NULL est l'ensemble des non-terminaux dérivant le mot ε. Il est défini par :

Définition 8.3 (NULL). *Soit $G = (N, \Sigma, S, P)$ une grammaire CF. On appelle NULL le sous-ensemble de N défini par :*

$$\text{NULL} = \{A \in N, A \overset{\star}{\underset{G}{\Rightarrow}} \varepsilon\}$$

Cet ensemble se déduit très simplement de la grammaire G par la procédure consistant à initialiser NULL avec $\emptyset$ et à ajouter itérativement dans NULL tous les non-terminaux A tels qu'il existe une production $A \to \alpha$ et que tous les symboles de α sont soit déjà dans NULL,

$$
\begin{aligned}
S &\rightarrow c \mid ABS \\
A &\rightarrow B \mid a \\
B &\rightarrow b \mid \varepsilon
\end{aligned}
$$

Grammaire 8.6 – Une grammaire (ambiguë) pour $(a \mid b)^\star c$

```
// Initialisation.
foreach A ∈ N do FIRST^0(A) := ∅;
k := 0;
cont := true;
while cont = true do
    k := k + 1;
    foreach A ∈ N do
        FIRST^k(A) := FIRST^{k-1}(A)
    foreach (A → X_1 ... X_k) ∈ P do
        j := 0;
        repeat
            j := j+1;
            // Par convention, si X_j est terminal, FIRST(X_j) = {X_j}
            FIRST^k(A) := FIRST^k(A) ∪ FIRST^k(X_j)
        until X_j ∉ NULL;
    // Vérifie si un des FIRST a changé; sinon stop
    cont := false;
    foreach A ∈ N do
        if FIRST^k(A) ≠ FIRST^{k-1}(A) then
            cont := true
```

Algorithme 8.7 – Calcul de FIRST

soit égaux à ε. Cette procédure s'achève lorsqu'un examen de toutes les productions de P n'entraîne aucun nouvel ajout dans NULL. Il est facile de voir que cet algorithme termine en un nombre fini d'étapes (au plus $|N|$). Illustrons son fonctionnement sur la grammaire 8.6.

Un premier examen de l'ensemble des productions conduit à insérer B dans NULL (à cause de $B \rightarrow \varepsilon$) ; la seconde itération conduit à ajouter A, puisque $A \rightarrow B$. Une troisième et dernière itération n'ajoute aucun autre élément dans NULL : en particulier S ne dérive pas ε puisque tout mot du langage engendré par cette grammaire contient au moins un c en dernière position.

Calculer FIRST Nous savons maintenant calculer NULL : il est alors possible d'écrire une procédure pour calculer FIRST(A) pour tout A. Le pseudo-code de cette procédure est donné par l'algorithme 8.7.

Illustrons le fonctionnement de cet algorithme sur la grammaire 8.6 : l'initialisation conduit à faire FIRST(S) = FIRST(A) = FIRST(B) = $\emptyset$. La première itération ajoute c dans FIRST(S), puis a dans FIRST(A) et b dans FIRST(B) ; la seconde itération conduit à augmenter FIRST(S) avec a (qui est dans FIRST(A)), puis avec b (qui n'est pas dans FIRST(A), mais comme A est dans NULL, il faut aussi considérer les éléments de FIRST(B)) ; on ajoute également durant cette

itération b dans FIRST(A). Une troisième itération ne change pas ces ensembles, conduisant finalement aux valeurs suivantes :

$$\begin{aligned} FIRST(S) &= \{a, b, c\} \\ FIRST(A) &= \{a, b\} \\ FIRST(B) &= \{b\} \end{aligned}$$

FOLLOW FOLLOW est nécessaire pour contrôler par anticipation la validité de prédictions de la forme $A \to \varepsilon$, ou, plus généralement $A \overset{\star}{\underset{G}{\Rightarrow}} \varepsilon$: pour valider un tel choix, il faut en effet connaître les terminaux qui peuvent apparaître après A dans un proto-mot. Une fois ces terminaux connus, il devient possible de valider l'application de $A \to \varepsilon$ en vérifiant que le symbole à apparier fait bien partie de cet ensemble ; si ce n'est pas le cas, alors l'utilisation de cette production aboutira nécessairement à un échec.

Formellement, on définit :

Définition 8.4 (FOLLOW)**.** *Soit $G = (N, \Sigma, S, P)$ une grammaire CF et A un élément de N. On appelle* FOLLOW(A) *le sous-ensemble de Σ défini par :*

$$\text{FOLLOW}(A) = \{a \in \Sigma, \exists u \in \Sigma^\star, \alpha, \beta \in (N \cup \Sigma)^\star, S \overset{\star}{\Rightarrow} uA\alpha \Rightarrow uAa\beta\}$$

Comment calculer ces ensembles ? En fait, la situation n'est guère plus compliquée que pour le calcul de FIRST : la base de la récursion est que si $A \to X_1 \dots X_n$ est une production de G, alors tout symbole apparaissant après A peut apparaître après X_n. La prise en compte des non-terminaux pouvant dériver ε complique un peu le calcul, et requiert d'avoir au préalable calculé NULL et FIRST. Ce calcul se formalise par l'algorithme 8.8.

Illustrons, de nouveau, le fonctionnement de cette procédure sur la grammaire 8.6. La première itération de cet algorithme conduit à examiner la production $S \to ABS$: cette règle présente une configuration traitée dans la première boucle `for` avec $X_j = A$, $l = 0$: les éléments de FIRST(B), soit b, sont ajoutés à FOLLOW$^1(A)$. Comme B est dans NULL, on obtient aussi que les éléments de FIRST(S) sont dans FOLLOW$^1(A)$, qui vaut alors $\{a, b, c\}$ ($j = 1, l = 1$). En considérant, toujours pour cette production, le cas $j = 2, l = 0$, il s'avère que les éléments de FIRST(S) doivent être insérés également dans FOLLOW$^1(B)$. La suite du déroulement de l'algorithme n'apportant aucun nouveau changement, on en reste donc à :

$$\begin{aligned} FOLLOW(S) &= \emptyset \\ FOLLOW(A) &= \{a, b, c\} \\ FOLLOW(B) &= \{a, b, c\} \end{aligned}$$

On notera que FOLLOW(S) est vide : aucun terminal ne peut apparaître à la droite de S dans un proto-mot. Ceci est conforme à ce que l'on constate en observant quelques dérivations : S figure, en effet, toujours en dernière position des proto-mots qui le contiennent.

```
// Initialisation
foreach A ∈ N do FOLLOW^0(A) := ∅ ;
foreach (A → X_1 ... X_n) ∈ P do
    for j = 1 to n − 1 do
        if X_j ∈ N then
            for l = 0 to n − j do
                // Inclut le cas où X_{j+1} ... X_{j+l+1} = ε
                if X_{j+1} ... X_{j+l} ∈ NULL* then
                    FOLLOW^0(X_j) = FOLLOW^0(X_j) ∪ FIRST(X_{j+l+1})
k := 0;
cont := true;
while cont = true do
    k := k + 1;
    foreach A ∈ N do
        FOLLOW^k(A) := FOLLOW^{k−1}(A)
    foreach (A → X_1 ... X_n) ∈ P do
        j := n+1;
        repeat
            j := j-1;
            FOLLOW^k(X_j) := FOLLOW^k(X_j) ∪ FOLLOW^k(A)
        until X_j ∉ NULL;
    // Vérifie si un des FOLLOW a changé; sinon stop
    cont := false;
    foreach A ∈ N do
        if FOLLOW^k(A) ≠ FOLLOW^{k−1}(A) then cont := true ;
```

Algorithme 8.8 – Calcul de FOLLOW

8.1.4 La table de prédiction

Nous avons montré à la section précédente comment calculer les ensembles FIRST() et FOLLOW(). Nous étudions, dans cette section, comment les utiliser pour construire des analyseurs efficaces. L'idée que nous allons développer consiste à déduire de ces ensembles une table M (dans $N \times \Sigma$) permettant de déterminer à coup sûr les bons choix à effectuer pour construire déterministiquement une dérivation gauche.

Comme préalable, généralisons la notion de FIRST à des séquences quelconques de $(N \cup \Sigma)^\star$ de la manière suivante [2].

$$\text{FIRST}(\alpha = X_1 \cdots X_k) = \begin{cases} \mathit{FIRST}(X_1) \text{ si } X_1 \notin \text{NULL} \\ \mathit{FIRST}(X_1) \cup \text{FIRST}(X_2 \cdots X_k) \text{ sinon} \end{cases}$$

Revenons maintenant sur l'intuition de l'analyseur esquissé pour les grammaires SLL(1) : pour ces grammaires, une production de type $A \to a\alpha$ est sélectionnée à coup sûr dès que A

2. Rappelons que nous avons déjà étendu la notion de FIRST (et de FOLLOW) pour des terminaux quelconques par FIRST(a) = FOLLOW(a) = a.

est le plus à gauche du proto-mot courant et que a est le symbole à apparier dans le mot en entrée.

Dans notre cas, c'est l'examen de l'ensemble des éléments pouvant apparaître en tête de la dérivation de la partie droite d'une production qui va jouer le rôle de sélecteur de la « bonne » production. Formellement, on commence à remplir M en appliquant le principe suivant :

> Si $A \to \alpha$, avec $\alpha \neq \varepsilon$, est une production de G et a est un élément de FIRST(α), alors on insère α dans $M(A, a)$

Ceci signifie simplement que lorsque l'analyseur descendant voit un A en tête du proto-mot courant et qu'il cherche à apparier un a, alors il est licite de prédire α, dont la dérivation peut effectivement débuter par un a.

Reste à prendre en compte le cas des productions de type $A \to \varepsilon$: l'idée est que ces productions peuvent être appliquées lorsque A est le plus à gauche du proto-mot courant, et que le symbole à apparier *peut suivre* A. Ce principe doit en fait être généralisé à toutes les productions $A \to \alpha$ où α ne contient que des symboles dans NULL ($\alpha \in \text{NULL}^\star$). Ceci conduit à la seconde règle de remplissage de M :

> Si $A \to \alpha$, avec $\alpha \in \text{NULL}^\star$, est une production de G et a est un élément de FOLLOW(A), alors insérer α dans $M(A, a)$.

Considérons alors une dernière fois la grammaire 8.6. L'application du premier principe conduit aux opérations de remplissage suivantes :

— par le premier principe on insérera c dans $M(S, c)$, puis ABS dans les trois cases $M(S, a)$, $M(S, b)$, $M(S, c)$. Ce principe conduit également à placer a dans $M(A, a)$, B dans $M(A, b)$ puis b dans $M(B, b)$.

— reste à étudier les deux productions concernées par le second principe de remplissage. Commençons par la production $B \to \varepsilon$: elle est insérée dans $M(B, x)$ pour tout symbole dans FOLLOW(B), soit dans les trois cases $M(B, a)$, $M(B, b)$, $M(B, c)$. De même, la partie droite de $A \to B$ doit être insérée dans toutes les cases $M(A, x)$, avec x dans FOLLOW(A), soit dans les trois cases : $M(A, a)$, $M(A, b)$, $M(A, c)$.

On parvient donc à la configuration du tableau 8.9.

	a	b	c
S	ABS	ABS	ABS c
A	a B	B	B
B	ε	ε b	ε

TABLE 8.9 – Table d'analyse prédictive pour la grammaire 8.6

L'examen du tableau 8.9 se révèle instructif pour comprendre comment analyser les mots avec cette grammaire. Supposons, en effet, que l'on souhaite analyser le mot bc. Le proto-mot courant étant initialisé avec un S, nous consultons la case $M(S, b)$ du tableau 8.9, qui nous

prescrit de récrire S en ABS. Comme nous n'avons vérifié aucun symbole dans l'opération, nous consultons maintenant la case $M(A, b)$. De nouveau la réponse est sans ambiguïté : appliquer $A \to B$. À ce stade, les choses se compliquent : l'opération suivante demande de consulter la case $M(B, b)$, qui contient deux productions possibles : $B \to b$ et $B \to \varepsilon$. Si, toutefois, on choisit la première, on aboutit rapidement à un succès de l'analyse : b étant apparié, il s'agit maintenant d'apparier le c, à partir du proto-mot courant : BS. Après consultation de la case $M(B, c)$, on élimine le B ; choisir $S \to c$ dans la case $M(S, c)$ achève l'analyse.

Pour aboutir à ce résultat, il a toutefois fallu faire des choix, car le tableau 8.9 ne permet pas de mettre en œuvre une analyse déterministe. Ceci est dû à l'ambiguïté de la grammaire 8.6, dans laquelle plusieurs (en fait une infinité de) dérivations gauches différentes sont possibles pour le mot c (comme pourra s'en persuader le lecteur en en listant quelques-unes).

8.1.5 Analyseurs LL(1)

Nous sommes finalement en mesure de définir les grammaires LL(1) et de donner un algorithme pour les analyser.

Définition 8.5 (Grammaire LL(1))**.** *Une grammaire hors-contexte est* LL(1) *si et seulement si sa table d'analyse prédictive* $M()$ *contient au plus une séquence de* $(N \cup \Sigma)^\star$ *pour chaque valeur* (A, a) *de* $N \times \Sigma$*.*

Toutes les grammaires ne sont pas LL(1), comme nous l'avons vu à la section précédente en étudiant une grammaire ambiguë. Il est, en revanche, vrai que toute grammaire LL(1) est non ambiguë. La démonstration est laissée en exercice.

Une grammaire LL(1) est susceptible d'être analysée par algorithme 8.10 (on suppose que la construction de M est une donnée de l'algorithme).

8.1.6 LL(1)-isation

Les grammaires LL(1) sont particulièrement sympathiques, puisqu'elles se prêtent à une analyse déterministe fondée sur l'exploitation d'une table de prédiction. Bien que toutes les grammaires ne soient pas aussi accommodantes, il est instructif d'étudier des transformations simples qui permettent de se rapprocher du cas LL(1). Un premier procédé de transformation consiste à supprimer les récursions gauches (directes et indirectes) de la grammaire : cette transformation est implicite dans le processus de mise sous forme normale de Greibach et est décrit à la section 9.2.2. Une seconde transformation bien utile consiste à *factoriser à gauche* la grammaire.

Pour mesurer l'utilité de cette démarche, considérons la grammaire 8.11.

Ce fragment de grammaire n'est pas conforme à la définition donnée pour les grammaires LL(1), puisqu'il apparaît clairement que le mot-clé *if* sélectionne deux productions possibles pour S. Il est toutefois possible de transformer la grammaire pour se débarrasser de cette configuration : il suffit ici de factoriser le plus long préfixe commun aux deux parties droites, et d'introduire un nouveau terminal S'. Ceci conduit à la grammaire 8.12.

```
// le mot en entrée est: u = u_1 ... u_n
// initialisation
matched := ε;
tomatch := u_1;
left := S;
i := 1;
while left ≠ ε ∧ i ≤ |u| do
    left = Xβ;
    if X ∈ N then
        if undefined(M(X, tomatch)) then return false ;
        α := M(X, tomatch);
        // Remplace A par α
        left := αβ
    else
        // le symbole de tête est terminal: on apparie simplement
        if X ≠ u_i then return false ;
    matched := matched · u_i;
    tomatch := u_{i+1};
    i := i + 1
if left = ε ∧ tomatch = ε then
    return true
else
    return false
```

Algorithme 8.10 – Analyseur pour grammaire LL(1)

$$
\begin{aligned}
S &\rightarrow \textit{ if } B \textit{ then } S \\
S &\rightarrow \textit{ if } B \textit{ then } S \textit{ else } S \\
&\dots
\end{aligned}
$$

Grammaire 8.11 – Une grammaire non-LL(1) pour *if-then-else*

Le mot clé *if* sélectionne maintenant une production unique; c'est aussi vrai pour *else*. Le fragment décrit dans la grammaire 8.12 devient alors susceptible d'être analysé déterministiquement (vérifiez-le en examinant comment appliquer à coup sûr $S' \rightarrow \varepsilon$).

Ce procédé se généralise au travers de la construction utilisée pour démontrer le théorème suivant :

Théorème 8.6 (Factorisation gauche). *Soit $G = (N, \Sigma, S, P)$ une grammaire hors-contexte, alors il existe une grammaire équivalente G' telle que si $A \rightarrow X_1 \cdots X_k$ et $A \rightarrow Y_1 \cdots Y_l$ sont deux A-productions de G', alors $X_1 \neq Y_1$.*

$$
\begin{aligned}
S &\rightarrow \textit{ if } B \textit{ then } S\ S' \\
S' &\rightarrow \textit{ else } S \mid \varepsilon \\
&\dots
\end{aligned}
$$

Grammaire 8.12 – Une grammaire factorisée à gauche pour *if-then-else*

Démonstration. La preuve repose sur le procédé de transformation suivant. Supposons qu'il existe une série de A-productions dont les parties droites débutent toutes par le même symbole X. En remplaçant toutes les productions $A \to X\alpha_i$ par l'ensemble $\{A \to XA' , A' \to \alpha_i\}$, où A' est un nouveau symbole non-terminal introduit pour la circonstance, on obtient une grammaire équivalente à G. Si, à l'issue de cette transformation, il reste des A'-productions partageant un préfixe commun, il est possible de répéter cette procédure, et de l'itérer jusqu'à ce qu'une telle configuration n'existe plus. Ceci arrivera au bout d'un nombre fini d'itérations, puisque chaque transformation a le double effet de réduire le nombre de productions potentiellement problématiques, ainsi que de réduire la longueur des parties droites. □

8.1.7 Quelques compléments

Détection des erreurs

Le rattrapage sur erreur La détection d'erreur, dans un analyseur LL(1), correspond à une configuration (non-terminal A le plus à gauche, symbole a à apparier) pour laquelle la table d'analyse ne prescrit aucune action. Le message à donner à l'utilisateur est alors clair :

> Arrivé au symbole numéro X, j'ai rencontré un a alors que j'aurais dû avoir . . . (suit l'énumération des symboles tels que $M(A, .)$ est non-vide).

Stopper là l'analyse est toutefois un peu brutal pour l'utilisateur, qui, en général, souhaite que l'on détecte en une seule passe toutes les erreurs de syntaxe. Deux options sont alors possibles pour continuer l'analyse :

- s'il n'existe qu'un seul b tel que $M(A, b)$ est non-vide, on peut faire comme si on venait de voir un b, et continuer. Cette stratégie de récupération d'erreur par insertion comporte toutefois un risque : celui de déclencher une cascade d'insertions, qui pourraient empêcher l'analyseur de terminer correctement ;
- l'alternative consisterait à détruire le a et tous les symboles qui le suivent jusqu'à trouver un symbole pour lequel une action est possible. Cette méthode est de loin préférable, puisqu'elle conduit à une procédure qui est assurée de se terminer. Pour obtenir un dispositif de rattrapage plus robuste, il peut être souhaitable d'abandonner complètement tout espoir d'étendre le A et de le faire disparaître : l'analyse reprend alors lorsque l'on trouve, dans le flux d'entrée, un symbole dans FOLLOW(A).

Les grammaires LL(k) Nous l'avons vu, toutes les grammaires ne sont pas LL(1), même après élimination des configurations les plus problématiques (récursions gauches...). Ceci signifie que, pour au moins un couple (A, a), $M(A, a)$ contient plus d'une entrée, indiquant que plusieurs prédictions sont en concurrence. Une idée simple pour lever l'indétermination concernant la « bonne » expansion de A consiste à augmenter le regard avant. Cette intuition se formalise à travers la notion de grammaire *LL(k)* et d'analyseur LL(k) : pour ces grammaires, il suffit d'un regard avant de k symboles pour choisir sans hésitation la A-production à appliquer ; les tables d'analyse correspondantes croisent alors des non-terminaux (en ligne) avec des mots de longueur k (en colonne). Il est important de réaliser que ce procédé ne permet pas de traiter toutes les grammaires : c'est évidemment vrai pour les grammaires

ambiguës ; mais il existe également des grammaires non-ambiguës, pour lesquelles aucun regard avant de taille borné ne permettra une analyse descendante déterministe.

Ce procédé de généralisation des analyseurs descendants, bien que fournissant des résultats théoriques importants, reste d'une donc utilité pratique modeste, surtout si on compare cette famille d'analyseurs à l'autre grande famille d'analyseurs déterministes, les analyseurs de type LR, qui font l'objet de la section suivante.

8.1.8 Un exemple complet commenté

Nous détaillons dans cette section les constructions d'un analyseur LL pour la grammaire 8.5, qui est rappelée dans la grammaire 8.13, sous une forme légèrement modifiée. L'axiome est la variable Z.

$$\begin{aligned} Z &\to S\# \\ S &\to S + T \mid T \\ T &\to T * F \mid F \\ F &\to (S) \mid D \\ D &\to 1 \mid 2 \end{aligned}$$

Grammaire 8.13 – Une grammaire (simplifiée) pour les expressions arithmétiques

La première étape de la construction consiste à se débarrasser des productions récursives à gauche, puis à factoriser à gauche la grammaire résultante. Au terme de ces deux étapes, on aboutit à la grammaire 8.14, qui comprend deux nouvelles variables récursives à droite, S' et T', et dont nous allons montrer qu'elle est effectivement LL(1).

$$\begin{aligned} Z &\to S\# \\ S &\to TS' \\ S' &\to +TS' \mid \varepsilon \\ T &\to FT' \\ T' &\to *FT' \mid \varepsilon \\ F &\to (S) \mid D \\ D &\to 1 \mid 2 \end{aligned}$$

Grammaire 8.14 – Une grammaire LL pour les expressions arithmétiques

Le calcul de NULL permet d'identifier S' et T' comme les seules variables dérivant le mot vide. Les étapes du calcul des ensembles FIRST et FOLLOW sont détaillées dans le tableau 8.15.

Il est alors possible de déterminer la table d'analyse prédictive, représentée dans le tableau 8.16. Cette table est sans conflit, puisque chaque case contient au plus une production.

Cette table permet effectivement d'analyser déterministiquement des expressions arithmétiques simples, comme le montre la trace d'exécution du tableau 8.17.

	FIRST				
It.	1	2	3	4	5
Z				(	(12
S			(	(12	(12
S'	+	+	+	+	+
T		(	(12	(12	(12
T'	*	*	*	*	*
F	(	(12	(12	(12	(12
D	12	12	12	12	12

	FOLLOW		
It.	0	1	2
Z			
S	#)	#)	#)
S'		#)	#)
T	+	+#)	+#)
T'		+#)	+#)
F	*	*+#)	*+#)
D		*+#)	*+#)

TABLE 8.15 – Calcul de FIRST et FOLLOW

	#	+	*	(	)	1	2
Z				$Z \to S\#$		$Z \to S\#$	$Z \to S\#$
S				$S \to TS'$		$S \to TS'$	$S \to TS'$
S'	$S' \to \varepsilon$	$S' \to +TS'$			$S' \to \varepsilon$		
T				$T \to FT'$		$T \to FT'$	$T \to FT'$
T'	$T' \to \varepsilon$	$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$		
F				$T \to (S)$		$T \to D$	$T \to D$
D						$D \to 1$	$D \to 2$

TABLE 8.16 – Table d'analyse prédictive pour la grammaire de l'arithmétique

8.2 Analyseurs LR

8.2.1 Concepts

Comme évoqué à la section 7.2, les analyseurs ascendants cherchent à récrire le mot à analyser afin de se ramener, par des réductions successives, à l'axiome de la grammaire. Les bifurcations dans le graphe de recherche correspondent alors aux alternatives suivantes :

(i) la partie droite α d'une production $A \to \alpha$ est trouvée dans le proto-mot courant ; on choisit de remplacer α par A pour construire un nouveau proto-mot et donc d'appliquer une *réduction*.

(ii) on poursuit l'examen du proto-mot courant en considérant un autre facteur α', obtenu, par exemple, en étendant α par la droite. Cette action correspond à un *décalage* de l'entrée.

Afin de rationaliser l'exploration du graphe de recherche correspondant à la mise en œuvre de cette démarche, commençons par décider d'une stratégie d'examen du proto-mot courant : à l'instar de ce que nous avons mis en œuvre dans l'algorithme 7.1, nous l'examinerons toujours depuis la gauche vers la droite. Si l'on note α le proto-mot courant, factorisé en $\alpha = \beta\gamma$, où β est la partie déjà examinée, l'alternative précédente se récrit selon :

— β possède un suffixe δ correspondant à la partie droite d'une règle : réduction et développement d'un nouveau proto-mot.

— β est étendu par la droite par décalage d'un nouveau terminal.

Pile	Symbole
Z	2
S#	2
TS'#	2
FT'S'#	2
DT'S'#	2
2T'S'#	*
2*FT'S'#	(
2*(S)T'S'#	1
2*(TS')T'S'#	1
2*(FT'S')T'S'#	1
2*(DT'S')T'S'#	1

Pile (suite)	Symbole
2*(1T'S')T'S'#	+
2*(1S')T'S'#	+
2*(1+TS')T'S'#	+
2*(1+FT'S')T'S'#	+
2*(1+DT'S')T'S'#	+
2*(1+2T'S')T'S'#	)
2*(1+2S')T'S'#	)
2*(1+2)T'S'#	#
2*(1+2)S'#	#
2*(1+2)#	#
succès	

TABLE 8.17 – Trace de l'analyse déterministe de $2 * (1 + 2)$

Cette stratégie conduit à la construction de dérivations *droites* de l'entrée courante. Pour vous en convaincre, remarquez que le suffixe γ du proto-mot courant n'est jamais modifié : il n'y a toujours que des terminaux à droite du symbole récrit, ce qui est conforme à la définition d'une dérivation droite.

Illustrons ce fait en considérant la grammaire 8.18.

$$\begin{aligned} S &\rightarrow AB \\ A &\rightarrow aA \mid b \\ B &\rightarrow bB \mid a \end{aligned}$$

Grammaire 8.18 – Une grammaire pour $a^\star bb^\star a$

Soit alors $u = abba$; une analyse ascendante de ce mot est reproduite au tableau 8.19.

En considérant les réductions opérées, on obtient la dérivation : $S \underset{G}{\Rightarrow} AB \underset{G}{\Rightarrow} AbB \underset{G}{\Rightarrow} Aba \underset{G}{\Rightarrow} aAba \underset{G}{\Rightarrow} abba$, qui est effectivement une dérivation droite. On note également qu'on a introduit un troisième type d'action de l'analyseur, consistant à *accepter* l'entrée comme un mot de la grammaire. Il existe enfin un quatrième type d'action, non représenté ici, consistant à diagnostiquer une situation d'échec de l'analyse.

Dernière remarque concernant cette trace : les différentes transformations possibles de α via les actions de réduction et de décalage n'agissent que sur les suffixes de β. Ceci suggère d'implanter β sous la forme d'une pile, sur laquelle s'accumulent (puis se réduisent) progressivement les symboles de u. Si l'on adopte ce point de vue, une pile correspondant à la trace du tableau 8.19 passerait par les états successifs suivants : a, ab, aA, A, Ab, Aba, AbB, AB, S. Bien naturellement, cette implantation demanderait également de conserver un pointeur vers la position courante dans u, afin de savoir quel symbole empiler. Dans la suite, nous ferons l'hypothèse que β est effectivement implanté sous la forme d'une pile.

Comment procéder pour rendre ce processus déterministe ? Cherchons des éléments de réponse dans la trace d'analyse présentée dans le tableau 8.19. Ce processus contient quelques

α	β	γ	Action
abba	ε	*abba*	décaler
abba	*a*	*bba*	décaler
abba	*ab*	*ba*	réduire par $A \to b$
aAba	*aA*	*ba*	réduire par $A \to aA$
Aba	*A*	*ba*	décaler
Aba	*Ab*	*a*	décaler
Aba	*Aba*	ε	réduire par $B \to a$
AbB	*AbB*	ε	réduire par $B \to bB$
AB	*AB*	ε	réduire par $S \to AB$
S			fin : accepter l'entrée

TABLE 8.19 – Analyse ascendante de $u = abba$

actions déterministes, comme la première opération de décalage : lorsqu'en effet β (la pile) ne contient aucun suffixe correspondant à une partie droite de production, décaler est l'unique action possible.

De manière duale, lorsque γ est vide, décaler est impossible : il faut nécessairement réduire, lorsque cela est encore possible : c'est, par exemple, ce qui est fait durant la dernière réduction.

Un premier type de choix correspond au second décalage : $\beta = a$ est à ce moment égal à une partie droite de production ($B \to a$), pourtant on choisit ici de décaler (ce choix est presque toujours possible) plutôt que de réduire. Notons que la décision prise ici est la (seule) bonne décision : réduire prématurément aurait conduit à positionner un B en tête de β, conduisant l'analyse dans une impasse : B n'apparaissant en tête d'aucune partie droite, il aurait été impossible de le réduire ultérieurement. Un second type de configuration (non représentée ici) est susceptible de produire du non-déterminisme : il correspond au cas où l'on trouve en queue de β deux parties droites de productions : dans ce cas, il faudra choisir entre deux réductions concurrentes.

Résumons-nous : nous voudrions, de proche en proche, et par simple consultation du sommet de la pile, être en mesure de décider déterministiquement si l'action à effectuer est un décalage ou bien une réduction (et dans ce cas quelle est la production à utiliser), ou bien encore si l'on se trouve dans une situation d'erreur. Une condition suffisante serait que, pour chaque production p, on puisse décrire l'ensemble L_p des configurations de la pile pour lesquelles une réduction par p est requise ; et que, pour deux productions p_1 et p_2 telles que $p_1 \neq p_2$, L_{p_1} et L_{p_2} soient toujours disjoints. Voyons à quelle(s) condition(s) cela est possible.

8.2.2 Analyseurs LR(0)

Pour débuter, formalisons la notion de contexte LR(0) d'une production.

Définition 8.7 (Contexte LR(0))**.** *Soit G une grammaire hors-contexte, et $p = (A \to \alpha)$ une*

production de G, on appelle contexte LR(0) *de p le langage* $L_{A\to\alpha}$ *défini par :*

$$L_{A\to\alpha} = \{\gamma = \beta\alpha \in (\Sigma \cup N)^\star \ tq.\ \exists v \in \Sigma^\star, S \overset{\star}{\underset{D}{\Rightarrow}} \beta A v \underset{D}{\Rightarrow} \beta\alpha v\}$$

En d'autres termes, tout mot γ de $L_{A\to\alpha}$ contient un suffixe α et est tel qu'il existe une réduction d'un certain γv en S qui débute par la réduction de α en A. Chaque fois qu'un tel mot γ apparaît dans la pile d'un analyseur ascendant gauche-droit, il est utile d'opérer la réduction $A \to \alpha$; à l'inverse, si l'on trouve dans la pile un mot absent de $L_{A\to\alpha}$, alors cette réduction ne doit pas être considérée, même si ce mot se termine par α.

Examinons maintenant de nouveau la grammaire 8.18 et essayons de calculer les langages L_p pour chacune des productions. Le cas de la première production est clair : il faut impérativement réduire lorsque la pile contient AB, et c'est là le seul cas possible. On déduit directement que $L_{S\to AB} = \{AB\}$. Considérons maintenant $A \to aA$: la réduction peut survenir quel que soit le nombre de a présents dans la pile. En revanche, si la pile contient un symbole différent de a, c'est qu'une erreur aura été commise. En effet :

— une pile contenant une séquence baA ne pourra que se réduire en AA, dont on ne sait plus que faire ; ceci proscrit également les piles contenant plus d'un A, qui aboutissent pareillement à des configurations d'échec ;

— une pile contenant une séquence $B \ldots A$ ne pourra que se réduire en BA, dont on ne sait non plus comment le transformer.

En conséquence, on a : $L_{A\to aA} = aa^\star A$. Des considérations similaires nous amènent à conclure que :

— $L_{A\to b} = a^\star b$;

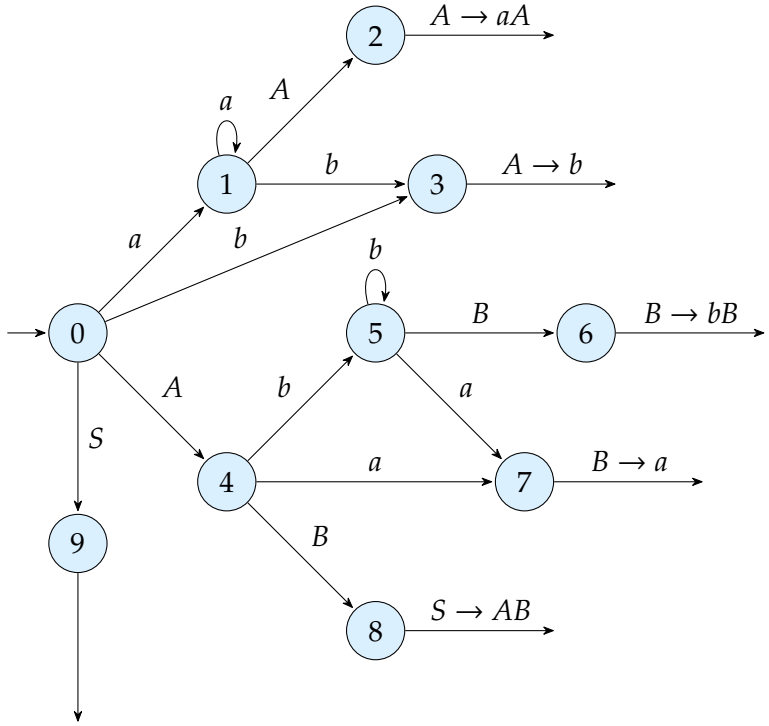
— $L_{B\to bB} = Abb^\star B$;

— $L_{B\to a} = Ab^\star a$;

Chacun des ensembles L_p se décrivant par une expression rationnelle, on déduit que l'ensemble des L_p se représente sur l'automate 8.20, qui réalise l'union des langages L_p. Vous noterez de plus que (i) l'alphabet d'entrée de cet automate contient à la fois des symboles terminaux et non-terminaux de la grammaire ; (ii) les états finaux de l'automate 8.20 sont associés aux productions correspondantes ; (iii) toute situation d'échec dans l'automate correspond à un échec de l'analyse : en effet, ces configurations sont celles où la pile contient un mot dont l'extension ne peut aboutir à aucune réduction : il est alors vain de continuer l'analyse.

Comment utiliser cet automate ? Une approche naïve consiste à mettre en œuvre la procédure suivante : partant de l'état initial $q_i = q_0$ et d'une pile vide on applique des décalages jusqu'à atteindre un état final q. On réduit ensuite la pile selon la production associée à q, donnant lieu à une nouvelle pile β qui induit un repositionnement dans l'état $q_i = \delta^\star(q_0, \beta)$ de l'automate. La procédure est itérée jusqu'à épuisement simultané de la pile et de u. Cette procédure est formalisée à travers l'algorithme 8.21.

L'algorithme 8.21 est inefficace : en effet, à chaque réduction, on se repositionne à l'état initial de l'automate, perdant ainsi le bénéfice de l'analyse des symboles en tête de la pile. Une meilleure implantation consiste à mémoriser, pour chaque symbole de la pile, l'état q atteint



Automate 8.20 – L'automate des réductions licites

```
// Initialisation.
β := ε // Initialisation de la pile.
q := q0 // Se positionner dans l'état initial.
i := j := 0;
while i ≤ |u| do
    while j ≤ |β| do
        j := j + 1;
        q := δ(q, βj)
    while q ∉ F do
        β := βui // Empilage de ui
        if δ(q, ui) existe then q := δ(q, ui) else return false ;
        i := i + 1
    // Réduction de p = A → γ.
    β := βγ^{-1}A ;
    j := 0;
    q := q0
if β = S ∧ q ∈ F then return true else return false;
```

Algorithme 8.21 – Reconnaissance ascendante guidée par un automate $A = (\Sigma \cup N, Q, q_0, F, \delta)$. Version 1

```
// Initialisations.
β := (ε, q_0) // Initialisation de la pile.
q := q_0 // Se positionner dans l'état initial.
i := 0;
while i ≤ |u| do
    while q ∉ F do
        if δ(q, u_i) existe then
            q := δ(q, u_i) // Progression dans A.
            push(β, (u_i, q)) // Empilage de (u_i, q).
            i := i + 1
        else
            return false
    // On atteint un état final: réduction de p = A → α.
    j := 0;
    // Dépilage de α.
    while j < |α| do
        pop(β);
        j := j + 1
    // (x, q) est sur le sommet de la pile: repositionnement.
    push(β, (A, δ(q, A)));
    q := δ(q, A);
if β = (S, q) ∧ q ∈ F then return true else return false ;
```

Algorithme 8.22 – Reconnaissance ascendante guidée par un automate $A = (\Sigma \cup N, Q, q_0, F, \delta)$. Version 2

dans A. À la suite d'une réduction, on peut alors directement se positionner sur q pour poursuivre l'analyse. Cette amélioration est mise en œuvre dans l'algorithme 8.22.

Notez, pour finir, que l'on effectue en fait deux sortes de transitions dans A : celles qui sont effectuées directement à l'issue d'un décalage et qui impliquent une extension de la pile ; et celles qui sont effectuées à l'issue d'une réduction et qui consistent simplement à un repositionnement ne nécessitant pas de nouveau décalage. Ces deux actions sont parfois distinguées, l'une sous le nom de décalage (*shift*), l'autre sous le nom de *goto*.

De l'automate précédent, se déduit mécaniquement une table d'analyse (dite LR(0)), donnée pour l'automate 8.20 au tableau 8.23. Cette table résume les différentes actions à effectuer en fonction de l'état courant de l'automate.

Le tableau 8.23 se consulte de la façon suivante. Pour chaque état, la colonne « Action » désigne l'unique type d'action à effectuer :

s
: effectuer un décalage. Lire le symbole en tête de pile, si c'est un x, consulter la colonne correspondante dans le groupe « Shift », et transiter dans l'état correspondant. *Attention* : le décalage a bien lieu de *manière inconditionnelle*, c'est-à-dire indépendamment de la valeur du symbole empilé ; en revanche, cette valeur détermine la transition de l'automate à exercer.

r $A \to \alpha$

État	Action	Shift a	Shift b	Goto A	Goto B
0	s	1	3	4	
1	s	1	3	2	
2	r $A \to aA$				
3	r $A \to b$				
4	s	7	5		8
5	s	7	5		6
6	r $B \to bB$				
7	r $B \to a$				
8	r $S \to AB$				

TABLE 8.23 – Une table d'analyse LR(0)

réduire la pile selon $A \to \alpha$. À l'issue de la réduction, le A sera posé sur la pile. Transiter vers l'état correspondant à A dans le groupe « Goto » et à l'état au sommet de la pile après en avoir retiré α.

Toutes les autres configurations (qui correspondent aux cases vides de la table) sont des situations d'erreur. Pour distinguer, dans la table, les situations où l'analyse se termine avec succès, il est courant d'utiliser la transformation suivante :

— on ajoute un nouveau symbole terminal représentant la fin du texte, par exemple #;

— on transforme G en G' en ajoutant un nouvel axiome Z et une nouvelle production $Z \to S\#$, où S est l'axiome de G.

— on transforme l'entrée à analyser en $u\#$;

— l'état (final) correspondant à la réduction $Z \to S\#$ est (le seul) état d'acceptation, puisqu'il correspond à la fois à la fin de l'examen de u (# est empilé) et à la présence du symbole S en tête de la pile.

Il apparaît finalement, qu'à l'aide du tableau 8.23, on saura analyser la grammaire 8.18 de manière déterministe et donc avec une complexité linéaire. Vous pouvez vérifier cette affirmation en étudiant le fonctionnement de l'analyseur pour les entrées $u = ba$ (succès), $u = ab$ (échec), $u = abba$ (succès).

Deux questions se posent alors : (i) peut-on, pour toute grammaire, construire un tel automate ? (ii) comment construire l'automate à partir de la grammaire G ? La réponse à (i) est non : il existe des grammaires (en particulier les grammaires ambiguës) qui résistent à toute analyse déterministe. Il est toutefois possible de chercher à se rapprocher de cette situation, comme nous le verrons à la section 8.2.3. Dans l'intervalle, il est instructif de réfléchir à la manière de construire automatiquement l'automate d'analyse A.

L'intuition de la construction se fonde sur les remarques suivantes. Initialement, on dispose de u, non encore analysé, qu'on aimerait pouvoir réduire en S, par le biais d'une série non-encore déterminée de réductions, mais qui s'achèvera nécessairement par une réduction de type $S \to \alpha$.

Supposons, pour l'instant, qu'il n'existe qu'une seule S-production : $S \to X_1 \dots X_k$: le but original de l'analyse « aboutir à une pile dont S est l'unique symbole en ayant décalé tous les

symboles de u » se reformule alors en : « aboutir à une pile égale à $X_1 \dots X_k$ en ayant décalé tous les symboles de u ». Ce nouveau but se décompose naturellement en une série d'étapes qui vont devoir être accomplies séquentiellement : d'abord parvenir à une configuration où X_1 est « au fond » de la pile, puis faire que X_2 soit empilé juste au-dessus de X_1...

Conserver la trace de cette série de buts suggère d'insérer dans l'automate d'analyse une branche correspondant à la production $S \to X_1 \dots X_k$, qui s'achèvera donc sur un état final correspondant à la réduction de $X_1 \dots X_k$ en S. Une telle branche est représentée à la figure 8.24.

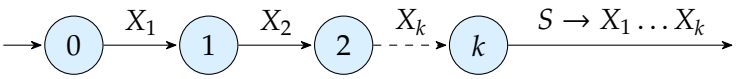


FIGURE 8.24 – Une branche de l'automate

Chaque état le long de cette branche correspond à la résolution d'un sous-but supplémentaire, la transition X_i annonçant l'empilage de X_i, et déclenchant la recherche d'un moyen d'empiler X_{i+1}. Pour formaliser cette idée, nous introduisons le concept de *production (ou règle) pointée* [3].

Définition 8.8 (Production pointée)**.** *Une* production pointée *d'une grammaire hors-contexte G est un triplet (A, α, β) de $N \times (N \cup \Sigma)^\star \times (N \cup \Sigma)^\star$, avec $A \to \gamma = \alpha\beta$ une production de G. Une production pointée est notée avec un point : $A \to \alpha \bullet \beta$.*

Une production pointée exprime la résolution partielle d'un but : $A \to \alpha \bullet \beta$ exprime que la résolution du but « empiler A en terminant par la réduction $A \to \alpha\beta$ » a été partiellement accomplie, en particulier que α a déjà été empilé et qu'il reste encore à empiler les symboles de β. Chaque état de la branche de l'automate correspondant à la production $A \to \gamma$ s'identifie ainsi à une production pointée particulière, les états initiaux et finaux de cette branche correspondant respectivement à : $A \to \bullet\gamma$ et $A \to \gamma\bullet$.

Retournons à notre problème original, et considérons maintenant le premier des sous-buts : « empiler X_1 au fond de la pile ». Deux cas de figure sont possibles :

— soit X_1 est symbole terminal : le seul moyen de l'empiler consiste à effectuer une opération de décalage, en cherchant un tel symbole en tête de la partie non encore analysée de l'entrée courante.

— soit X_1 est un non-terminal : son insertion dans la pile résulte nécessairement d'une série de réductions, dont la dernière étape concerne une X_1-production : $X_1 \to Y_1 \dots Y_n$. De nouveau, le but « observer X_1 au fond de la pile » se décompose en une série de sous-buts, donnant naissance à une nouvelle « branche » de l'automate pour le mot $Y_1 \dots Y_n$. Comment relier ces deux branches ? Tout simplement par une transition ε entre les deux états initiaux, indiquant que l'empilage de X_1 se résoudra en commençant l'empilage de Y_1 (voir la figure 8.25). S'il existe plusieurs X_1-productions, on aura une branche (et une transition ε) par production.

  Ce procédé se généralise : chaque fois qu'une branche porte une transition $\delta(q, X) = r$, avec X un non-terminal, il faudra ajouter une transition entre q et tous les états initiaux des branches correspondants aux X-productions.

3. On trouve également le terme d'*item* et en anglais de *dotted rule*.

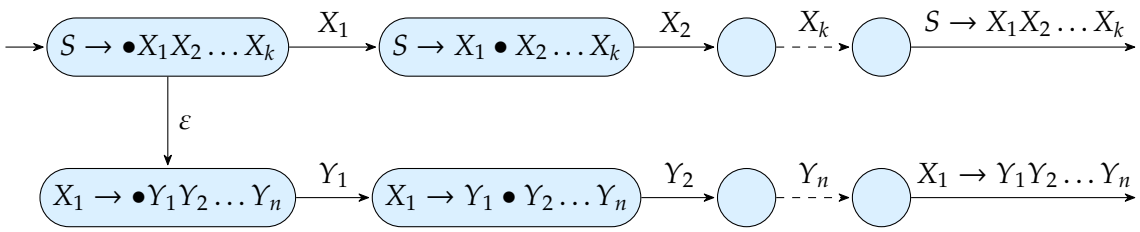


FIGURE 8.25 – Deux branches de l'automate

De ces remarques découle un procédé systématique pour construire un ε-NFA $(N \cup \Sigma, Q, q_0, F, \delta)$ permettant de guider une analyse ascendante à partir d'une grammaire $G = (\Sigma, N, Z, P)$, telle que Z est non-récursif et ne figure en partie gauche que dans l'unique règle $Z \to \alpha$:

— $Q = \{[A \to \alpha \bullet \beta] \text{ avec } A \to \alpha\beta \in P\}$

— $q_0 = [Z \to \bullet\alpha]$

— $F = \{[A \to \alpha\bullet] \text{ avec } A \to \alpha \in P\}$; à chaque état final est associée une production $A \to \alpha$

— $\forall q = [A \to \alpha \bullet X\beta] \in Q, \delta(q, X) = [A \to \alpha X \bullet \beta]$

— $\forall q = [A \to \alpha \bullet B\beta] \in Q$ tq. $B \in N, \forall q' = [B \to \bullet\gamma], \delta(q, \varepsilon) = q'$.

Le théorème suivant, non démontré ici, garantit que ce procédé de construction permet effectivement d'identifier les contextes LR(0) des productions.

Théorème 8.9. *Soit A l'automate fini dérivé de G par la construction précédente, alors : $\delta^\star(q_0, \gamma) = [A \to \alpha \bullet \beta]$ si et seulement si :*

(i) $\exists \eta, \gamma = \eta\alpha$

(ii) $\gamma\beta \in L_{A \to \alpha\beta}$

Ce résultat assure, en particulier, que lorsque l'on atteint un état final de l'automate (pour $\beta = \varepsilon$), la pile contient effectivement un mot appartenant au contexte LR(0) de la production associée à l'état final atteint.

Il est naturellement possible de déterminiser cet automate, en appliquant les algorithmes de suppression des transitions spontanées (voir le théorème 4.13), puis la construction des sous-ensembles décrite dans le théorème 4.9 ; on prendra soin de propager lors de ces transformations l'information de réduction associée aux états finaux des branches. En clair, si un état q du déterminisé contient une production pointée originale de type $A \to \alpha\bullet$, alors q sera un état (final) dans lequel cette réduction est possible. Le lecteur est vivement encouragé à entreprendre cette démarche et vérifier qu'il retrouve bien, en partant de la grammaire 8.18, un automate ressemblant fortement à l'automate 8.20.

Une construction directe de l'automate LR(0) consiste à construire à la volée les ensembles d'items qui définissent les états de l'automate déterminisé, ainsi que les transitions associées. Soit I un ensemble d'items, on définit tout d'abord la *clôture* de I comme :

Définition 8.10 (Clôture d'un ensemble d'items LR(0))**.** *La clôture $cl(I)$ d'un ensemble I d'items LR(0) est le plus petit ensemble vérifiant :*

— $I \subset cl(I)$

— *si* $[X \to \alpha \bullet Y\beta] \in cl(I)$, *alors* $\forall Y \to \gamma$, $[Y \to \bullet\gamma] \in cl(I)$

L'automate LR(0) est alors défini par :

— $q_0 = cl([z \to \bullet S\#]$ est l'état initial
— $\forall I, \forall a \in \Sigma, \delta(I, a) = cl\{[X \to \alpha a \bullet \beta]$, avec $[X \to \alpha \bullet a\beta] \in I\}$
— $\forall I, \forall X \in N, \delta(I, X) = cl\{[X \to \alpha X \bullet \beta]$, avec $[X \to \alpha \bullet X\beta] \in I\}$
— $F = \{I, \exists [X \to \alpha\bullet] \in I\}$

On déduit finalement de cet automate déterministe, par le procédé utilisé pour construire le tableau 8.23, la *table d'analyse LR(0)*.

Définition 8.11 (Grammaire LR(0))**.** *Une grammaire hors-contexte est* LR(0)[4] *si sa table* $T()$ *d'analyse LR(0) est telle que : pour toute ligne i (correspondant à un état de l'automate), soit il existe* $x \in N \cup \Sigma$ *tel que* $T(i, x)$ *est non-vide et* $T(i, *)$ *est vide ; soit* $T(i, *)$ *est non-vide et contient une réduction unique.*

Une grammaire LR(0) peut être analysée de manière déterministe. La définition d'une grammaire LR(0) correspond à des contraintes qui sont souvent, dans la pratique, trop fortes pour des grammaires « réelles », pour lesquelles la construction de la table LR(0) aboutit à des conflits. Le premier type de configuration problématique correspond à une indétermination entre décaler et réduire (conflit *shift/reduce*) ; le second type correspond à une indétermination sur la réduction à appliquer ; on parle alors de conflit *reduce/reduce*. Vous noterez, en revanche, qu'il n'y a jamais de conflit *shift/shift*. Pourquoi ?

Comme pour les analyseurs descendants, il est possible de lever certaines indéterminations en s'autorisant un regard en avant sur l'entrée courante ; les actions de l'analyseur seront alors conditionnées non seulement par l'état courant de l'automate d'analyse, mais également par les symboles non-analysés. Ces analyseurs font l'objet de la section qui suit.

8.2.3 Analyseurs SLR(1), LR(1), LR(k)...

Regarder vers l'avant

Commençons par étudier le cas le plus simple, celui où les conflits peuvent être résolus avec un regard avant de 1. C'est, par exemple, le cas de la grammaire 8.26 :

$$\begin{array}{ll} S \to E\# & (1) \\ E \to T + E \mid T & (2), (3) \\ T \to x & (4) \end{array}$$

Grammaire 8.26 – Une grammaire non-LR(0)

Considérons une entrée telle que $x + x + x$: sans avoir besoin de construire la table LR(0), il apparaît qu'après avoir empilé le premier x, puis l'avoir réduit en T, par $T \to x$, deux alternatives vont s'offrir à l'analyseur :

4. Un 'L' pour *left-to-right*, un 'R' pour *rightmost derivation*, un 0 pour *0 lookahead* ; en effet, les décisions (décalage vs. réduction) sont toujours prises étant uniquement donné l'état courant (le sommet de la pile).

— soit immédiatement réduire T en E

— soit opérer un décalage et continuer de préparer une réduction par $E \to T + E$.

Il apparaît pourtant que '+' ne peut jamais suivre E dans une dérivation réussie : vous pourrez le vérifier en construisant des dérivations droites de cette grammaire. Cette observation anticipée du prochain symbole à empiler suffit, dans le cas présent, à restaurer le déterminisme. Comment traduire cette intuition dans la procédure de construction de l'analyseur ?

La manière la plus simple de procéder consiste à introduire des conditions supplémentaires aux opérations de réduction : dans les analyseurs LR(0), celles-ci s'appliquent de manière inconditionnelle, c'est-à-dire quel que soit le prochain symbole apparaissant dans l'entrée courante (le regard avant) : à preuve, les actions de réduction apparaissent dans la table d'analyse dans une colonne séparée. Pourtant, partant d'une configuration dans laquelle la pile est $\beta\gamma$ et le regard avant est a, réduire par $X \to \gamma$ aboutit à une configuration dans laquelle la pile vaut βX. Si l'analyse devait se poursuivre avec succès, on aurait alors construit une dérivation droite $S \overset{\star}{\Rightarrow} \beta Xa... \Rightarrow \beta\gamma a$, dans laquelle la lettre a suit directement X. En conséquence, la réduction ne peut déboucher sur un succès que si a est un successeur de X, information qui est donnée dans l'ensemble FOLLOW(X) (cf. la section 8.1.3).

Ajouter cette contrainte supplémentaire à la procédure de construction de la table d'analyse consiste à conditionner les actions de réduction $X \to \gamma$ par l'existence d'un regard avant appartenant à l'ensemble FOLLOW(X) : elles figureront dans la table d'analyse dans les colonnes dont l'en-tête est un terminal vérifiant cette condition.

Si cette restriction permet d'aboutir à une table ne contant qu'une action par cellule, alors on dit que la grammaire est *SLR(1)* (pour *simple LR*, quand on s'autorise un regard avant de 1).

Illustrons cette nouvelle procédure de la construction de la table d'analyse. Les actions de l'automate LR(0) sont données dans le tableau 8.27a, dans laquelle on observe un conflit décaler/réduire dans l'état 2. Le calcul des ensembles FOLLOW aboutit à FOLLOW(S) = $\emptyset$, FOLLOW(E) = {#}, FOLLOW(T) = {#, +}, ce qui permet de préciser les conditions d'application des réductions, donnant lieu à la table d'analyse reproduite dans le tableau 8.27b. La grammaire est donc SLR(1).

En guise d'application, montrez que la grammaire 8.5 est SLR(1). On introduira pour l'occasion une nouvelle production $Z \to S\#$; vous pourrez également considérer D comme un terminal valant pour n'importe quel nombre entier.

LR(1)

Si cette procédure échoue, l'étape suivante consiste à modifier la méthode de construction de l'automate LR(0) en choisissant comme ensemble d'états toutes les paires [5] constituées d'une production pointée et d'un terminal. On note ces états $[A \to \beta \bullet \gamma, a]$. La construction de l'automate d'analyse LR (ici LR(1)) $(N \cup \Sigma, Q, q_0, F, \delta)$ se déroule alors comme suit (on suppose toujours que l'axiome Z n'est pas récursif et n'apparaît que dans une seule règle) :

— $Q = \{[A \to \alpha \bullet \beta, a]$ avec $A \to \alpha\beta \in P$ et $a \in \Sigma\}$

5. En fait, les prendre *toutes* est un peu excessif, comme il apparaîtra bientôt.

		Shift			Goto	
État	Action	x	+	#	E	T
0	s	3			1	2
1	s			6		
2	s/r3		4			
3	r4					
4	s	3			5	2
5	r2					
6	acc					

(a) Table d'analyse LR(0) avec conflit (état 2)

	Action			Goto	
État	x	+	#	E	T
0	s3			g1	g2
1			s6		
2	s4		r3		
3		r4	r4		
4	s3			g5	g2
5			r2		
6			acc		

(b) Table d'analyse SLR(1) sans conflit

TABLE 8.27 – Tables d'analyse LR(0) et SLR(1) de la grammaire 8.26

Les numéros de règles associés aux réductions correspondent à l'ordre dans lesquelles elles apparaissent dans la table ci-dessus (1 pour $S \to E\#...$). On notera que la signification des en-têtes de colonne change légèrement entre les deux représentations : dans la table LR(0), les actions sont effectuées sans regard avant ; les labels de colonnes indiquent des transitions ; dans la table SLR, les actions « réduire » et « décaler » sont effectuées *après* consultation du regard avant. L'état 6 est un état d'acceptation ; par la convention selon laquelle un fichier se termine par une suite infinie de #, cette action est reportée en colonne # : en quelque sorte FOLLOW(S) = {#}. Par conséquent, lire un # dans l'état 1 de la table SLR conduira bien à accepter l'entrée (a).

— $q_0 = [Z \to \bullet\alpha, ?]$, où ? est un symbole « joker » qui vaut pour n'importe quel symbole terminal ;

— $F = \{[A \to \alpha\bullet, a],$ avec $A \to \alpha \in P\}$; à chaque état final est associée la production $A \to \alpha$, correspondant à la réduction à opérer ;

— $\forall q = [A \to \alpha \bullet X\beta, a] \in Q, \delta(q, X) = [A \to \alpha X \bullet \beta, a]$; comme précédemment, ces transitions signifient la progression de la résolution du but courant par empilement de X ;

— $\forall q = [A \to \alpha \bullet B\beta, a] \in Q$ tq. $B \in N, \forall q' = [B \to \bullet\gamma, b]$ tq. $b \in \text{FIRST}(\beta a), \delta(q, \varepsilon) = q'$.
La condition supplémentaire $b \in \text{FIRST}(\beta a)$ (voir la section 8.1.3) introduit une information de désambiguïsation qui permet de mieux caractériser les réductions à opérer. En particulier, dans le cas LR(1), on espère qu'en prenant en compte la valeur du premier symbole dérivable depuis βa , il sera possible de sélectionner la bonne action à effectuer en cas de conflit sur B.
Pour illustrer ce point, considérons de nouveau la grammaire 8.26. L'état initial de l'automate LR(1) correspondant est $[S \to \bullet E\#]$; lequel a deux transitions ε, correspondant aux deux façons d'empiler un E, atteignant respectivement les états $q_1 = [E \to \bullet T + E, \#]$ et $q_2 = [E \to \bullet T, \#]$. Ces deux états se distinguent par leurs transitions ε sortantes : dans le cas de q_1, vers $[T \to \bullet\, x, +]$; dans le cas de q_2, vers $[T \to \bullet\, x, \#]$. À l'issue de la réduction d'un x en T, le regard avant permet de décider sans ambiguïté de l'action : si c'est un '+', il faut continuer de chercher une réduction $E \to T + E$; si c'est un '#', il faut réduire selon $E \to T$.

Construction pas-à-pas de l'automate d'analyse

La construction de la table d'analyse à partir de l'automate se déroule comme pour la table LR(0). Détaillons-en les principales étapes : après construction et déterminisation de l'automate LR(1), on obtient un automate fini dont les états finaux sont associés à des productions de G. On procède alors comme suit :

— pour chaque transition de q vers r étiquetée par un terminal a, la case $T(q, a)$ contient la séquence d'actions (décaler, consommer a en tête de la pile, aller en r) ;

— pour chaque transition de q vers r étiquetée par un non-terminal A, la case $T(q, A)$ contient la séquence d'actions (consommer A en tête de la pile, aller en r) ;

— pour chaque état final $q = [A \to \alpha\bullet, a]$, la case $T(q, a)$ contient l'unique action (réduire la pile selon $A \to \alpha$) : la décision de réduction (ainsi que la production à appliquer) est maintenant conditionnée par la valeur du regard avant associé à q.

Lorsque $T()$ ne contient pas de conflit, la grammaire est dite *LR(1)*. Il existe des grammaires LR(1) ou « presque [6] » LR(1) pour la majorité des langages informatiques utilisés dans la pratique.

En revanche, lorsque la table de l'analyseur LR(1) contient des conflits, il est de nouveau possible de chercher à augmenter le regard avant pour résoudre les conflits restants. Dans la pratique [7], toutefois, pour éviter la manipulation de tables trop volumineuses, on préférera chercher des moyens ad-hoc de résoudre les conflits dans les tables LR(1) plutôt que d'envisager de construire des tables LR(2) ou plus. Une manière courante de résoudre les conflits consiste à imposer des priorités via des règles du type : « en présence d'un conflit shift/reduce, toujours choisir de décaler [8] »...

En guise d'application, le lecteur est invité à s'attaquer à la construction de la table LR(1) pour la grammaire 8.26 et d'en déduire un analyseur déterministe pour cette grammaire. Idem pour la grammaire 8.28, qui engendre des mots tels que $x = **x$.

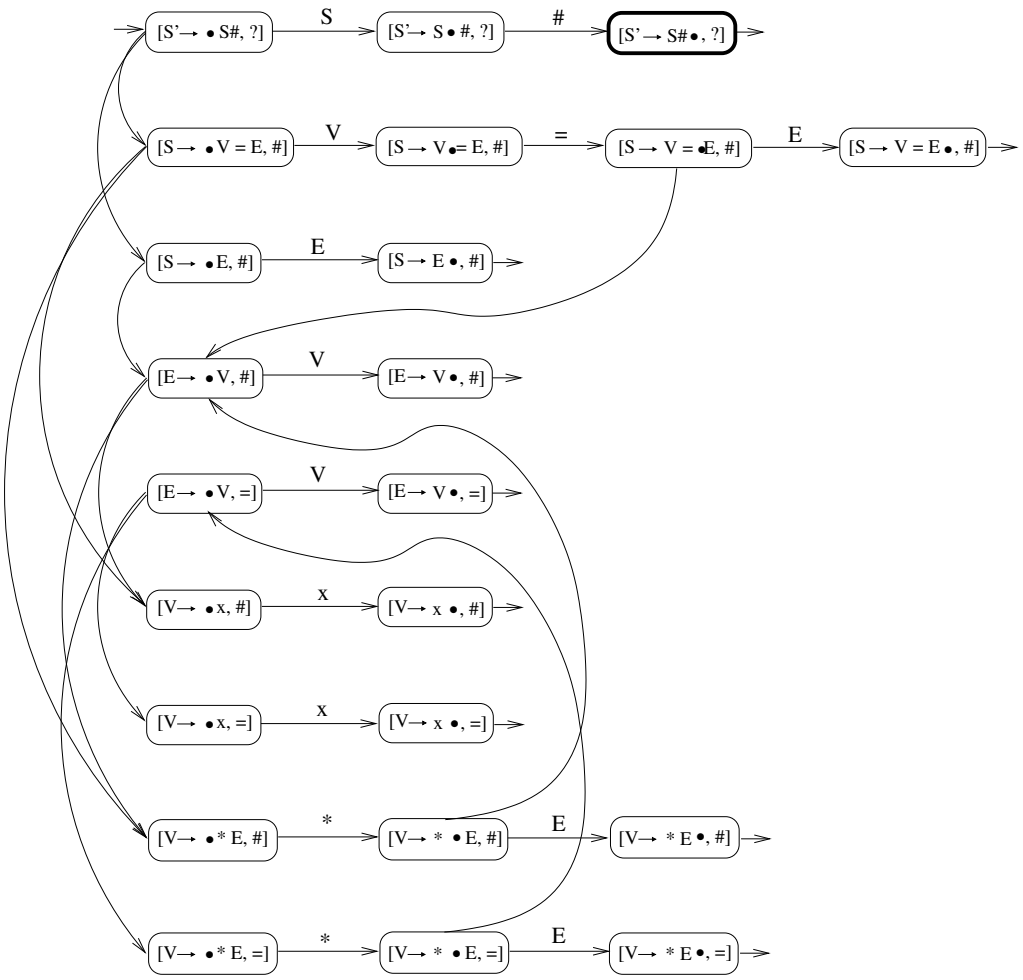
$$
\begin{aligned}
S' &\to S\# \\
S &\to V = E \mid E \\
E &\to V \\
V &\to *E \mid x
\end{aligned}
$$

Grammaire 8.28 – Une grammaire pour les manipulations de pointeurs

6. C'est-à-dire que les tables correspondantes sont presque sans conflit.

7. Il existe une autre raison, théorique celle-là, qui justifie qu'on se limite aux grammaires LR(1) : les grammaires LR(1) engendrent tous les langages hors-contexte susceptibles d'être analysés par une procédure déterministe ! En d'autres termes, l'augmentation du regard avant peut conduire à des grammaires plus simples à analyser ; mais ne change rien à l'expressivité des grammaires. La situation diffère donc ici de ce qu'on observe pour la famille des grammaires LL(k), qui induit une hiérarchie stricte de langages.

8. Ce choix de privilégier le décalage sur la réduction n'est pas innocent : en cas d'imbrication de structures concurrentes, il permet de privilégier la structure la plus intérieure, ce qui correspond bien aux attentes des humains. C'est ainsi que le mot `if cond1 then if cond2 then inst1 else inst2` sera plus naturellement interprété `if cond1 then (if cond2 then inst1 else inst2)` que `if cond1 then (if cond2 then inst1) else inst2` en laissant simplement la priorité au décalage du `else` plutôt qu'à la réduction de `if cond2 then inst1`.



Pour améliorer la lisibilité, les ε ne sont pas représentés. L'état d'acceptation est représenté en gras.

Automate 8.29 – Les réductions licites pour la grammaire 8.28

Pour vous aider dans votre construction, l'automate 8.29 permet de visualiser l'effet de l'ajout du regard avant sur la construction de l'automate d'analyse.

Une construction directe

Construire manuellement la table d'analyse LR est fastidieux, en particulier à cause du passage par un automate non-déterministe, qui implique d'utiliser successivement des procédures de suppression des transitions spontanées, puis de déterminisation. Dans la mesure où la forme de l'automate est très stéréotypée, il est possible d'envisager la construction directe de l'automate déterminisé à partir de la grammaire, en s'aidant des remarques suivantes (cf. la construction directe de l'analyseur LR(0) supra) :

— chaque état du déterminisé est un ensemble d'éléments de la forme [production pointée,

terminal] : ceci du fait de l'application de la construction des sous-ensembles (cf. la section 4.1.4)

— la suppression des transitions spontanées induit la notion de *fermeture* : la ε-fermeture d'un état étant l'ensemble des états atteints par une ou plusieurs transitions ε.

Cette procédure de construction du DFA A est détaillée dans l'algorithme 8.30, qui utilise deux fonctions auxiliaires pour effectuer directement la déterminisation.

```
Function LR0 is                                          // Programme principal.
    q_0 = Closure([S' → •S#, ?]) // L'état initial de A.
    Q := {q_0} // Les états de A.
    T := ∅ // Les transitions de A.
    while true do
        Q_i := Q;
        T_i := T;
        foreach q ∈ Q do                          // q est lui-même un ensemble !
            foreach [A → β • Xγ, a] ∈ q do
                r := Successor(q, X);
                Q := Q ∪ {r};
                T := T ∪ {(q, X, r)};
        // Test de stabilisation.
        if Q = Q_i ∧ T = T_i then break ;
// Procédures auxiliaires.
Function Closure(q) is          // Construction directe de la ε-fermeture.
    while true do
        q_i := q;
        foreach [A → β • Xγ, a] ∈ q do
            foreach (X → α) ∈ P do
                foreach b ∈ FIRST(γa) do
                    q := q ∪ [X → •α, b]
        // Test de stabilisation.
        if q = q_i then break ;
    return q
Function Successor(q, X) is                    // Développement des branches.
    r := ∅;
    foreach [A → β • Xγ, a] ∈ q do
        r := r ∪ [A → βX • γ, a]
    return Closure(r)
```

Algorithme 8.30 – Construction de l'automate d'analyse LR(1)

En guise d'application, vous vérifierez que la mise en œuvre de l'algorithme 8.30 construit directement le déterminisé de l'automate 8.29.

8.2.4 Compléments

LR et LL La famille des analyseurs *LR(k)* permet d'analyser tous les langages LL(k) et bien d'autres langages non-ambigus. La raison de cette plus grande généricité des analyseurs LR est au fond leur plus grande « prudence » : alors qu'un analyseur LL(k) doit pouvoir sélectionner sans erreur une production $A \to \alpha$ sur la seule base des k symboles terminaux non encore appariés (dont tout ou partie peut être dérivé de A), un analyseur LR(k) fonde sa décision d'appliquer une réduction $A \to \alpha$ sur (i) la connaissance de l'intégralité de la partie droite α et (ii) la connaissance des k symboles terminaux à droite de A. Pour k fixé, il est alors normal qu'un analyseur LR(k), ayant plus d'information à sa disposition qu'un analyseur LL(k), fasse des choix plus éclairés, faisant ainsi porter moins de contraintes sur la forme de la grammaire.

Génération d'analyseurs Lorsque l'on s'intéresse à des grammaires réelles, la construction de l'automate et de la table d'analyse LR peut rapidement conduire à de très gros automates : il est en fait nécessaire de déterminiser un automate dont le nombre d'états est proportionnel à la somme des longueurs des parties droites des productions de G ; étape qui peut conduire (cf. la section 4.1.4) à un automate déterministe ayant exponentiellement plus d'états que le non-déterministe d'origine. Il devient alors intéressant de recourir à des programmes capables de construire automatiquement un analyseur pour une grammaire LR : il en existe de nombreux, dont le plus fameux, `yacc` est disponible et documenté (dans sa version libre, connue sous le nom de `bison`) à l'adresse suivante : `http://www.gnu.org/software/bison/bison.html`.

LR et LALR et ... Utiliser un générateur d'analyseurs tel que `bison` ne fait que reporter sur la machine la charge de construire (et manipuler) un gros automate ; ce qui, en dépit de l'indéniable bonne volonté générale des machines, peut malgré tout poser problème. Le remède le plus connu est d'essayer de compresser *avec perte*, lorsque cela est possible (et c'est le cas général) les tables d'analyse LR(1), donnant lieu à la classe d'analyseurs LALR(1), qui sont ceux que construit `yacc`. Il existe de nombreuses autres variantes des analyseurs LR visant à fournir des solutions pour sauver le déterminisme de l'analyse, tout en maintenant les tables dans des proportions raisonnables.

Notes historiques L'invention de LL est due à Conway (1963). Sa formalisation fut faite par Lewis et Stearns (1968). On doit la découverte de la famille LR à Donald Knuth en 1965 (Knuth, 1965). Il est également à l'origine de l'appellation LL.

Chapitre 9

Normalisation des grammaires CF

Dans ce chapitre, nous étudions quelques résultats complémentaires concernant les grammaires hors-contexte, résultats qui garantissent d'une part que les cas de règles potentiellement problématiques pour l'analyse (productions ε, récursions gauches...) identifiées dans les discussions du chapitre 7 possèdent des remèdes bien identifiés ; d'autre part, qu'il est possible d'imposer *a priori* des contraintes sur la forme de la grammaire, en utilisant des transformations permettant de mettre les productions sous une forme standardisée. On parle, dans ce cas, de grammaires sous *forme normale*. Nous présentons dans ce chapitre les deux formes normales les plus utiles, la forme normale de Chomsky et la forme normale de Greibach.

9.1 Simplification des grammaires CF

Dans cette section, nous nous intéressons tout d'abord aux procédures qui permettent de simplifier les grammaires CF, en particulier pour faire disparaître un certain nombre de configurations potentiellement embarrassantes pour les procédures d'analyse.

9.1.1 Quelques préliminaires

Commençons par deux résultats élémentaires, que nous avons utilisés sans les formaliser à différentes reprises.

Une première notion utile est celle du langage engendré par un non-terminal A. Formellement,

Définition 9.1 (Langage engendré par un non-terminal)**.** *Soit $G = (N, \Sigma, S, P)$ une grammaire CF. On appelle* langage engendré par le non-terminal A*, noté $L_A(G)$, le langage engendré par la grammaire $G' = (N, \Sigma, A, P)$.*

Le langage engendré par un non-terminal est donc le langage obtenu en choisissant ce non-terminal comme axiome. Le langage engendré par la grammaire G est alors simplement le langage $L(G) = L_S(G)$.

Introduisons maintenant la notion de sous-grammaire :

Définition 9.2 (Sous-grammaire)**.** *Soit $G = (N, \Sigma, S, P)$ une grammaire CF. On appelle* sous-grammaire *toute grammaire $G' = (N, \Sigma, S, P')$, avec $P' \subset P$. Si G' est une sous-grammaire de G, alors $L(G') \subset L(G)$.*

Une sous-grammaire de G n'utilise qu'un sous-ensemble des productions de G ; le langage engendré est donc un sous-ensemble du langage engendré par G.

Le procédé suivant nous fournit un premier moyen pour construire systématiquement des grammaires équivalentes à une grammaire G. De manière informelle, ce procédé consiste à court-circuiter des étapes de dérivation en les remplaçant par une dérivation en une étape. Cette opération peut être effectuée sans changer le langage engendré, comme l'énonce le résultat suivant.

Lemme 9.3 (« Court-circuitage » des dérivations)**.** *Soit $G = (N, \Sigma, S, P)$ une grammaire CF ; soient A un non-terminal et α une séquence de terminaux et non-terminaux tels que $A \overset{\star}{\underset{G}{\Rightarrow}} \alpha$. Alors $G' = (N, \Sigma, S, P \cup \{A \to \alpha\})$ est faiblement équivalente à G.*

Démonstration. Le lemme précédent concernant les sous-grammaires nous assure que $L(G) \subset L(G')$. Inversement, soit u dans $L(G')$: si sa dérivation n'utilise pas la production $A \to \alpha$, alors la même dérivation existe dans G ; sinon si sa dérivation utilise $A \to \alpha$, alors il existe une dérivation dans G utilisant $A \overset{\star}{\underset{G}{\Rightarrow}} \alpha$. □

Nous terminons cette section par un second procédé permettant de construire des grammaires équivalentes, qui utilise en quelque sorte la distributivité des productions.

Lemme 9.4 (« Distributivité » des dérivations)**.** *Soit G une CFG, $A \to \alpha_1 B \alpha_2$ une A-production de G et $\{B \to \beta_1, \ldots, B \to \beta_n\}$ l'ensemble des B-productions. Alors, la grammaire G' dérivée de G en supprimant $A \to \alpha_1 B \alpha_2$ et en ajoutant aux productions de G l'ensemble $\{A \to \alpha_1\beta_1\alpha_2, \ldots, A \to \alpha_1\beta_n\alpha_2\}$ est faiblement équivalente à G.*

La démonstration de ce second résultat est laissée en exercice.

9.1.2 Non-terminaux inutiles

La seule contrainte définitoire posée sur les productions des CFG est que leur partie gauche soit réduite à un non-terminal. Toutefois, un certain nombre de configurations posent des problèmes pratiques, notamment lorsqu'il s'agit de mettre en œuvre des algorithmes d'analyse. Nous nous intéressons, dans cette section, aux configurations qui, sans introduire de difficultés majeures, sont source d'inefficacité.

Une première source d'inefficacité provient de l'existence de non-terminaux n'apparaissant que dans des parties droites de règles. En fait, ces non-terminaux ne dérivent aucun mot et peuvent être supprimés, ainsi que les règles qui y font référence, sans altérer le langage engendré par la grammaire.

Une configuration voisine est fournie par des non-terminaux (différents de l'axiome) qui n'apparaîtraient dans aucune partie droite. Ces non-terminaux ne pouvant être dérivés de l'axiome, ils sont également inutiles et peuvent être supprimés.

Enfin, les non-terminaux improductifs sont ceux qui, bien que dérivables depuis l'axiome, n'apparaissent dans aucune dérivation réussie, parce qu'ils sont impossibles à éliminer. Pensez, par exemple, à un non-terminal X qui n'apparaîtrait (en partie gauche) que dans une seule règle de la forme $X \to aX$.

Formalisons maintenant ces notions pour construire un algorithme permettant de se débarrasser des non-terminaux et des productions inutiles.

Définition 9.5 (Utilité d'une production et d'un non-terminal)**.** *Soit G une CFG, on dit qu'une production $P = A \to \alpha$ de G est* utile *si et seulement s'il existe un mot $w \in \Sigma^*$ tel que $S \overset{\star}{\underset{G}{\Rightarrow}} xAy \overset{P}{\underset{G}{\Rightarrow}} x\alpha y \overset{\star}{\underset{G}{\Rightarrow}} w$. Sinon, on dit que P est* inutile*.*

*De même, on qualifie d'*utiles *les non-terminaux qui figurent en partie gauche des règles utiles.*

L'identification des productions et non-terminaux utiles se fait en appliquant les deux procédures suivantes :

— La première étape consiste à étudier successivement toutes les grammaires $G_A = (N, \Sigma, A, P)$ pour $A \in N$. Le langage $L(G_A)$ contient donc l'ensemble des mots qui se dérivent depuis le non-terminal A. Il existe un algorithme (cf. la section 6.3) permettant de déterminer si $L(G_A)$ est vide. On construit alors G' en supprimant de G tous les non-terminaux A pour lesquels $L(G_A) = \emptyset$, ainsi que les productions dans lesquels ils apparaissent [1]. La grammaire G' ainsi construite est *fortement* équivalente à G. En effet :

  — $L(G') \subset L(G)$, par le simple fait que G' est une sous-grammaire de G ; de plus, les dérivations gauches de G' sont identiques à celles de G.

  — $L(G) \subset L(G')$: s'il existe un mot $u \in \Sigma^\star$ tel que $S \overset{\star}{\underset{G}{\Rightarrow}} u$ mais pas $S \overset{\star}{\underset{G'}{\Rightarrow}} u$, alors nécessairement la dérivation de u contient un des non-terminaux éliminés de G. Ce non-terminal dérive un des facteurs de u, donc engendre un langage non-vide, ce qui contredit l'hypothèse.

  Cette procédure n'est toutefois pas suffisante pour garantir qu'un non-terminal est utile : il faut, de plus, vérifier qu'il peut être dérivé depuis S. C'est l'objet de la seconde phase de l'algorithme.

— Dans une seconde étape, on construit récursivement les ensembles N_U et P_U contenant respectivement des non-terminaux et des productions. Ces ensembles contiennent initialement respectivement S et toutes les productions de partie gauche S. Si, à une étape donnée de la récursion, N_U contient A, alors on ajoute à P_U toutes les règles dont A est partie gauche, et à N_U tous les non-terminaux figurant dans les parties droites de ces règles. Cette procédure s'arrête après un nombre fini d'itérations, quand plus aucun

1. La procédure esquissée ici est mathématiquement suffisante, mais algorithmiquement naïve. Une implémentation plus efficace consisterait à déterminer de proche en proche l'ensemble des non-terminaux engendrant un langage non-vide, par une procédure similaire à celle décrite plus loin. L'écriture d'un tel algorithme est laissée en exercice.

non-terminal ne peut être ajouté à N_U. Par construction, pour toute production $A \to \alpha$ de P_U, il existe α_1 et α_2 tels que $S \overset{\star}{\underset{G}{\Rightarrow}} \alpha_1 A \alpha_2 \underset{G}{\Rightarrow} \alpha_1 \alpha \alpha_2$.

En supprimant de G tous les terminaux qui ne sont pas dans N_U, ainsi que les productions P_U correspondantes, on construit, à partir de G, une nouvelle sous-grammaire G', qui par construction ne contient que des terminaux et des productions utiles. Par un argument similaire au précédent, on vérifie que G' est fortement équivalente à G.

Attention : ces deux procédures doivent être appliquées dans un ordre précis. En particulier, il faut commencer par supprimer les variables ne générant aucun mot, puis éliminer celles qui n'apparaissent dans aucune dérivation. Vous pourrez vous en convaincre en examinant la grammaire 9.1.

$$\begin{aligned} S &\to a \mid AB \\ A &\to b \end{aligned}$$

Grammaire 9.1 – Élimination des productions inutiles : l'ordre importe

9.1.3 Cycles et productions non-génératives

Les cycles correspondent à des configurations mettant en jeu des productions « improductives » de la forme $A \to B$. Ces productions, que l'on appelle *non-génératives*, effectuent un simple renommage de variable, sans réellement entraîner la génération (immédiate ou indirecte) de symboles terminaux.

Ces productions sont potentiellement nuisibles pour les algorithmes de génération ou encore d'analyse ascendante, qui peuvent être conduits dans des boucles sans fin. Ceci est évident dans le cas de productions de type $A \to A$, mais apparaît également lorsque l'on a des cycles de productions non-génératives comme dans : $A \to B$, $B \to C$, $C \to A$. Ces cycles doivent donc faire l'objet d'un traitement particulier. Fort heureusement, pour chaque mot dont la dérivation contient un cycle, il existe également une dérivation sans cycle, suggérant qu'il est possible de se débarrasser des cycles sans changer le langage reconnu. C'est précisément ce qu'affirme le théorème suivant :

Théorème 9.6 (Élimination des cycles)**.** *Soit G une CFG. On peut contruire une CFG G', faiblement équivalente à G, qui ne contient aucune production de la forme $A \to B$, où A et B sont des non-terminaux.*

Avant de rentrer dans les détails techniques, donnons l'intuition de la construction qui va être développée dans la suite : pour se débarrasser d'une règle $A \to B$ sans perdre de dérivation, il « suffit » de rajouter à la grammaire une règle $A \to \beta$ pour chaque règle $B \to \beta$: ceci permet effectivement bien de court-circuiter la production non-générative. Reste un problème à résoudre : que faire des productions $B \to C$? En d'autres termes, comment faire pour s'assurer qu'en se débarrassant d'une production non-générative, on n'en a pas ajouté une autre ? L'idée est de construire en quelque sorte la clôture transitive de ces productions non-génératives, afin de détecter (et de supprimer) les cycles impliquant de telles productions.

$S \to A$	$A \to B$	$B \to bb$	$C \to B$
$S \to B$	$A \to B$	$B \to C$	$C \to Aa$
	$A \to aB$		$C \to aAa$

Grammaire 9.2 – Une grammaire contenant des productions non-génératives

Définition 9.7 (Dérivation non-générative)**.** *$B \in N$ dérive immédiatement non-génèrativement de A dans G si et seulement si $A \to B$ est une production de G. On notera $A \underset{G}{\mapsto} B$.*

B dérive non-génèrativement de A dans G, noté $A \overset{\star}{\underset{G}{\mapsto}} B$ si et seulement si $\exists X_1 \dots X_n$ dans N tels que $A \underset{G}{\mapsto} X_1 \underset{G}{\mapsto} \cdots \underset{G}{\mapsto} X_n \underset{G}{\mapsto} B$.

Démonstration du théorème 9.6. Un algorithme de parcours du graphe de la relation $\underset{G}{\mapsto}$ permet de déterminer $C_A = \{A\} \cup \{X \in N, A \overset{\star}{\underset{G}{\mapsto}} X\}$ de proche en proche pour chaque non-terminal A.

Construisons alors G' selon :

— G' a le même axiome, les mêmes terminaux et non-terminaux que G ;

— $A \to \alpha$ est une production de G' si et seulement s'il existe dans G une production $X \to \alpha$, avec $X \in C_A$ et $\alpha \notin N$. Cette condition assure en particulier que G' est bien sans production non-générative.

En d'autres termes, on remplace les productions $X \to \alpha$ de G en court-circuitant (de toutes les manières possibles) X.

Montrons alors que G' est bien équivalente à G. Soit en effet D une dérivation gauche minimale dans G, contenant une séquence (nécessairement sans cycle) maximale de productions non-génératives de $X_1, \dots, X_k$ suivie d'une production générative $X_k \to \alpha$. Cette séquence peut être remplacée par $X_1 \to \alpha$, qui par construction existe dans G'. Inversement, toute dérivation dans G' ou bien n'inclut que des productions de G, ou bien inclut au moins une production $A \to \alpha$ qui n'est pas dans G. Mais alors il existe dans G une séquence de règles non-génératives $A \to \cdots \to X \to \alpha$ et la dérivation D existe également dans G. On notera que contrairement à l'algorithme d'élimination des variables inutiles, cette transformation a pour effet de modifier (en fait d'aplatir) les arbres de dérivation : G et G' ne sont que faiblement équivalentes. □

Pour illustrer le fonctionnement de cet algorithme, considérons la grammaire 9.2. Le calcul de la clôture transitive de $\underset{G}{\mapsto}$ conduit aux ensembles suivants :

$$\begin{aligned} C_S &= \{S, A, B, C\} \\ C_A &= \{A, B, C\} \\ C_B &= \{B, C\} \\ C_C &= \{B, C\} \end{aligned}$$

La grammaire 9.3, G', contient alors les productions qui correspondent aux quatre seules productions génératives : $a \to aB$, $B \to bb$, $C \to aAa$, $C \to Aa$.

$S \to aB$	$B \to bb$	$A \to aB$	$C \to aAa$
$S \to bb$	$B \to aAa$	$A \to bb$	$C \to Aa$
$S \to aAa$	$B \to Aa$	$A \to aAa$	$C \to bb$
$S \to Aa$		$A \to Aa$	

Grammaire 9.3 – Une grammaire débarrassée de ses productions non-génératives

9.1.4 Productions ε

Si l'on accepte la version libérale de la définition des grammaires CF (cf. la discussion de la section 5.2.6), un cas particulier de règle licite correspond au cas où la partie droite d'une production est vide : $A \to \varepsilon$. Ceci n'est pas gênant en génération et signifie simplement que le non-terminal introduisant ε peut être supprimé de la dérivation. Pour les algorithmes d'analyse, en revanche, ces productions particulières peuvent singulièrement compliquer le travail, puisque l'analyseur devra à tout moment examiner la possibilité d'insérer un non-terminal. Il peut donc être préférable de chercher à se débarrasser de ces productions avant d'envisager d'opérer une analyse : le résultat suivant nous dit qu'il est possible d'opérer une telle transformation et nous montre comment la mettre en œuvre.

Théorème 9.8 (Suppression des productions ε)**.** *Si L est un langage engendré par G, une grammaire CF telle que toute production de G est de la forme : $A \to \alpha$, avec α éventuellement vide, alors L peut être engendré par une grammaire $G' = (N \cup \{S'\}, \Sigma, S', P')$, dont les productions sont soit de la forme $A \to \alpha$, avec α non-vide, soit $S' \to \varepsilon$ et S' n'apparaît dans la partie droite d'aucune règle.*

Ce résultat dit deux choses : d'une part que l'on peut éliminer toutes les productions ε sauf peut-être une (si $\varepsilon \in L(G)$), dont la partie gauche est alors l'axiome ; d'autre part que l'axiome lui-même peut être rendu non-récursif (i.e. ne figurer dans aucune partie droite). L'intuition du premier de ces deux résultats s'exprime comme suit : si S dérive ε, ce ne peut être qu'au terme d'un enchaînement de productions n'impliquant que des terminaux qui engendrent ε. En propageant de manière ascendante la propriété de dériver ε, on se ramène à une grammaire équivalente dans laquelle seul l'axiome dérive (directement) ε.

Démonstration. Commençons par le second résultat en considérant le cas d'une grammaire G admettant un axiome récursif S. Pour obtenir une grammaire G' (faiblement) équivalente, il suffit d'introduire un nouveau non-terminal S', qui sera le nouvel axiome de G' et une nouvelle production : $S' \to S$. Cette transformation n'affecte pas le langage engendré par la grammaire G.

Supposons alors, sans perte de généralité, que l'axiome de G est non-récursif et intéressons-nous à l'ensemble N_ε des variables A de G telles que le langage engendré par A, $L_A(G)$, contient ε. Quelles sont-elles ? Un cas évident correspond aux productions $A \to \varepsilon$. Mais ε peut également se déduire depuis A par plusieurs règles, $A \to \alpha \overset{\star}{\underset{G}{\Rightarrow}} \varepsilon$, à condition que tous les symboles de α soient eux-mêmes dans N_ε. On reconnaît sous cette formulation le problème du calcul de l'ensemble NULL que nous avons déjà étudié lors de la présentation des analyseurs LL (voir en particulier la section 8.1.3).

Une fois N_ε calculé, la transformation suivante de G conduit à une grammaire G' faiblement équivalente : G' contient les mêmes non-terminaux, terminaux et axiome que G. De surcroît,

Productions de G	Productions de G'
$S \to ASB \mid c$	$S \to ASB \mid AS \mid SB \mid S \mid c$
$A \to aA \mid B$	$A \to aA \mid a \mid B$
$B \to b \mid \varepsilon$	$B \to b$

Grammaire 9.4 – Une grammaire avant et après élimination des productions ε

G' contient toutes les productions de G n'impliquant aucune variable de N_ε. Si maintenant G contient une production $A \to \alpha$ et α inclut des éléments de N_ε, alors G' contient *toutes* les productions de type $A \to \beta$, où β s'obtient depuis α en supprimant une ou plusieurs variables de N_ε. Finalement, si S est dans N_ε, alors G' contient $S \to \varepsilon$. G' ainsi construite est équivalente à G : notons que toute dérivation de G qui n'inclut aucun symbole de N_ε se déroule à l'identique dans G'. Soit maintenant une dérivation impliquant un symbole de N_ε : soit il s'agit de $S \overset{\star}{\underset{G}{\Rightarrow}} \varepsilon$ et la production $S' \to \varepsilon$ permet une dérivation équivalente dans G' ; soit il s'agit d'une dérivation $S \overset{\star}{\underset{G}{\Rightarrow}} \alpha \underset{G}{\Rightarrow} \beta \overset{\star}{\underset{G}{\Rightarrow}} u$, avec $u \neq \varepsilon$ et β contient au moins un symbole X de N_ε, mais pas α. Mais pour chaque X de β, soit X engendre un facteur vide de u, et il existe une production de G' qui se dispense d'introduire ce non-terminal dans l'étape $\alpha \underset{G}{\Rightarrow} \beta$; soit au contraire X n'engendre pas un facteur vide et la même dérivation existe dans G'. On conclut donc que $L(G) = L(G')$. □

Cette procédure est illustrée par la grammaire 9.4 : G contenant deux terminaux qui dérivent le mot vide ($N_\varepsilon = \{A, B\}$), les productions de G' se déduisent de celles de G en considérant toutes les manières possibles d'éviter d'avoir à utiliser ces terminaux.

9.1.5 Élimination des récursions gauches directes

On appelle *directement récursifs* les non-terminaux A d'une grammaire G qui sont tels que $A \overset{\star}{\Rightarrow} A\alpha$ (récursion gauche) ou $A \overset{\star}{\Rightarrow} \alpha A$ (récursion droite). Les productions impliquant des récursions gauches directes posent des problèmes aux analyseurs descendants, qui peuvent être entraînés dans des boucles sans fin (cf. les sections 7.3 et 8.1). Pour utiliser de tels analyseurs, il importe donc de savoir se débarrasser de telles productions. Nous nous attaquons ici aux récursions gauches directes ; les récursions gauches indirectes seront traitées plus loin (à la section 9.2.2).

Il existe un procédé mécanique permettant d'éliminer ces productions, tout en préservant le langage reconnu. L'intuition de ce procédé est la suivante : de manière générique, un terminal récursif à gauche est impliqué dans la partie gauche de deux types de production : celles qui sont effectivement récursives à gauche et qui sont de la forme :

$$A \to A\alpha_1 \mid A\alpha_2 \ldots \mid A\alpha_n$$

et celles qui permettent d'éliminer ce non-terminal, et qui sont de la forme :

$$A \to \beta_1 \mid \beta_2 \ldots \mid \beta_m$$

où le premier symbole x_i de β_i n'est pas un A.

L'effet net de l'utilisation de ces productions conduit donc à des dérivations gauches de A dans lesquelles on « accumule » à droite de A un nombre arbitraire de α_i ; l'élimination de A introduisant en tête du proto-mot le symbole (terminal ou non-terminal) x_j. Formellement :

$$A \underset{G}{\Rightarrow} A\alpha_{i_1} \underset{G}{\Rightarrow} A\alpha_{i_2}\alpha_{i_1} \ldots \overset{\star}{\underset{G}{\Rightarrow}} A\alpha_{i_n} \ldots \alpha_{i_1}$$

L'élimination de A par la production $A \to \beta_j$ conduit à un proto-mot

$$\beta_j \alpha_{i_n} \ldots \alpha_{i_1}$$

dont le symbole initial est donc β_j. Le principe de la transformation consiste à produire β_j sans délai et à simultanément transformer la récursion gauche (qui « accumule » simplement les α_i) en une récursion droite. Le premier de ces buts est servi par l'introduction d'un nouveau non-terminal R dans des productions de la forme :

$$A \to \beta_1 \mid \beta_2 \ldots \mid \beta_m \mid \beta_1 R \ldots \mid \beta_m R$$

La récursivité du terminal A est préservée par la nouvelle série de productions :

$$R \to \alpha_1 R \mid \alpha_2 R \ldots \alpha_n R \mid \alpha_1 \mid \alpha_2 \ldots \mid \alpha_n$$

On vérifie, par les techniques habituelles de preuve (e.g. par induction sur la longueur des dérivations) que cette transformation produit bien une grammaire (faiblement) équivalente à la grammaire de départ.

En guise d'illustration, reprenons la grammaire 8.5 des expressions arithmétiques, entrevue lors de notre présentation des analyseurs LL(1). La grammaire 9.5, qui rappelle cette grammaire[2], contient deux terminaux directement récursifs à gauche.

$$\begin{aligned} S &\to S - F \mid F \\ F &\to F/T \mid T \\ T &\to (S) \mid D \\ D &\to 0 \mid \ldots \mid 9 \mid 0D \mid \ldots \mid 9D \end{aligned}$$

Grammaire 9.5 – Une grammaire pour les expressions arithmétiques

Le premier non-terminal à traiter est S, qui contient une règle directement récursive et une règle qui ne l'est pas. On introduit donc le nouveau symbole S', ainsi que les deux productions : $S \to FS' \mid F$ et $S' \to -FS' \mid -F$. Le traitement du symbole F se déroule de manière exactement analogue.

9.2 Formes normales

La notion de *forme normale* d'une grammaire répond à la nécessité, pour un certain nombre d'algorithmes de parsage, de disposer d'une connaissance *a priori* sur la forme des productions de la grammaire. Cette connaissance est exploitée pour simplifier la programmation

2. À une différence près, et de taille : nous utilisons ici les opérateurs − et /, qui sont ceux qui sont associatifs par la gauche et qui justifient l'écriture d'une grammaire avec une récursion gauche. Si l'on avait que + et ∗, il suffirait d'écrire les règles avec une récursion droite et le tour serait joué !

d'un algorithme de parsage, ou encore pour accélérer l'analyse. Les principales formes normales (de Chomsky et de Greibach) sont décrites dans les sections qui suivent. On verra que les algorithmes de mise sous forme normale construisent des grammaires faiblement équivalentes à la grammaire d'origine : les arbres de dérivation de la grammaire normalisée devront donc être transformés pour reconstruire les dérivations (et les interprétations) de la grammaire originale.

9.2.1 Forme normale de Chomsky

Théorème 9.9 (Forme normale de Chomsky)**.** *Toute grammaire hors-contexte admet une grammaire faiblement équivalente dans laquelle toutes les productions sont soit de la forme $A \to BC$, soit de la forme $A \to a$, avec A, B, C des non-terminaux et a un terminal. Si, de surcroît, $S \overset{\star}{\underset{G}{\Rightarrow}} \varepsilon$, alors la forme normale contient également $S \to \varepsilon$. Cette forme est appelée* forme normale de Chomsky, *abrégée en CNF conformément à la terminologie anglaise (Chomsky Normal Form).*

Démonstration. Les résultats de simplification obtenus à la section 9.1 nous permettent de faire en toute généralité l'hypothèse que G ne contient pas de production ε autre que éventuellement $S \to \varepsilon$ et que si $A \to X$ est une production de G, alors X est nécessairement terminal (il n'y a plus de production non-générative).

La réduction des autres productions procède en deux étapes : elle généralise tout d'abord l'introduction des terminaux par des règles ne contenant que ce seul élément en partie droite ; puis elle ramène toutes les autres productions à la forme normale correspondant à deux non-terminaux en partie droite.

Première étape, construisons $G' = (N', \Sigma, S, P')$ selon :

- N' contient tous les symboles de N ;
- pour tout symbole a de Σ, on ajoute une nouvelle variable A_a et une nouvelle production $A_a \to a$.
- toute production de P de la forme $A \to a$ est copiée dans P'
- toute production $A \to X_1 \cdots X_k$, avec tous les X_i dans N est copiée dans P' ;
- soit $A \to X_1 \cdots X_m$ contenant au moins un terminal : cette production est transformée en remplaçant chaque occurrence d'un terminal a par la variable A_a correspondante.

Par des simples raisonnements inductifs sur la longueur des dérivations, on montre que pour toute variable de G, $A \overset{\star}{\underset{G}{\Rightarrow}} u$ si et seulement si $A \overset{\star}{\underset{G'}{\Rightarrow}} u$, puis l'égalité des langages engendrés par ces deux grammaires. En effet, si $A \to u = u_1 \cdots u_l$ est une production de G, on aura dans G' :

$$A \underset{G'}{\Rightarrow} A_{u_1} \cdots A_{u_l} \underset{G'}{\Rightarrow} u_1 A_{u_2} \cdots A_{u_l} \cdots \overset{\star}{\underset{G'}{\Rightarrow}} u$$

Supposons que cette propriété soit vraie pour toutes les dérivations de longueur n et soit A et u tels que $A \overset{\star}{\Rightarrow} u$ en $n+1$ étapes. La première production est de la forme : $A \to x_1 A_1 x_2 A_2 \cdots A_k$, où chaque x_i ne contient que des terminaux. Par hypothèse de récurrence, les portions de u engendrées dans G par les variables A_i se dérivent également dans G' ; par construction

chaque x_i se dérive dans G' en utilisant les nouvelles variables A_a, donc $A \overset{\star}{\underset{G'}{\Rightarrow}} u$. La réciproque se montre de manière similaire.

Seconde phase de la procédure : les seules productions de G' dans lesquelles apparaissent des terminaux sont du type recherché $A \to a$; il s'agit maintenant de transformer G' en G'' de manière que toutes les productions qui contiennent des non-terminaux en partie droite en contiennent exactement 2. Pour aboutir à ce résultat, il suffit de changer toutes les productions de type $A \to B_1 ... B_m, m \geq 3$ dans G' par l'ensemble des productions suivantes (les non-terminaux D_i sont créés pour l'occasion) : $\{A \to B_1\ D_1, D_1 \to B_2\ D_2, \ldots, D_{m-2} \to B_{m-1}\ B_m\}$. De nouveau, il est relativement direct de montrer que le langage engendré par G'' est le même que celui engendré par G. □

Les arbres de dérivation des grammaires CNF ont une forme extrêmement caractéristique, puisque chaque nœud interne de l'arbre a soit un fils unique et ce fils est une feuille portant un symbole terminal ; soit exactement deux fils et ces deux fils sont des variables.

Comme exemple d'application, entreprenez la mise sous forme normale de Chomsky de la grammaire 9.6. Le résultat auquel vous devez aboutir est la grammaire 9.7.

$$\begin{array}{lll} S \to bA & A \to a & B \to b \\ S \to aB & A \to aS & B \to bS \\ & A \to bAA & B \to aBB \end{array}$$

Grammaire 9.6 – Une grammaire à Chomsky-normaliser

$$\begin{array}{llll} S \to A_bA & A \to a & B \to b & A_a \to a \\ S \to A_aB & A \to A_aS & B \to A_bS & A_b \to b \\ & A \to A_bD_1 & B \to A_aD_2 & D_1 \to AA \\ & & & D_2 \to BB \end{array}$$

Grammaire 9.7 – Une grammaire Chomsky-normalisée

En plus de son importance théorique, cette forme normale présente de nombreux avantages :

— un premier avantage est que les terminaux sont introduits par des productions dédiées, de type $A \to a$ (au moins une par terminal). Ceci s'avère particulièrement bienvenu pour les algorithmes de parsage descendant.

— La mise sous CNF facilite également les étapes de réduction des analyseurs ascendants, puisqu'il suffit simplement de regarder les couples de non-terminaux successifs pour juger si une telle opération est possible.

En revanche, cette transformation conduit en général à une augmentation sensible du nombre de productions dans la grammaire. Cette augmentation joue dans le sens d'une pénalisation générale des performances des analyseurs.

9.2.2 Forme normale de Greibach

La transformation d'une grammaire en une grammaire sous forme normale de Greibach généralise en un sens le procédé d'élimination des récursions gauches directes (cf. la sec-

tion 9.1.5), dans la mesure où elle impose une contrainte qui garantit que chaque production augmente de manière strictement monotone le préfixe terminal du proto-mot en cours de dérivation. À la différence de la forme normale de Chomsky, l'existence de la forme normale de Greibach est plus utilisée pour démontrer divers résultats théoriques que pour fonder des algorithmes d'analyse.

La forme normale de Greibach est définie dans le théorème suivant :

Théorème 9.10 (Forme normale de Greibach)**.** *Tout langage hors-contexte L ne contenant pas ε peut être engendré par une grammaire dont toutes les productions sont de la forme $A \to a\alpha$, avec a un terminal et α est une séquence de non-terminaux (éventuellement vide). Cette grammaire est appelée* forme normale de Greibach*, en abrégé GNF, conformément à la terminologie anglaise.*

Si L contient ε, alors ce résultat reste valide, en ajoutant toutefois une règle $S \to \varepsilon$, qui dérive ε depuis l'axiome, comme démontré à la section 9.1.4.

Démonstration. La preuve de l'existence d'une GNF repose, comme précédemment, sur un algorithme permettant de construire explicitement la forme normale de Greibach à partir d'une grammaire CF quelconque. Cet algorithme utilise deux procédés déjà présentés, qui transforment une grammaire CF en une grammaire CF faiblement équivalente.

Le premier procédé est celui décrit au lemme 9.3 et consiste à ajouter une production $A \to \alpha$ à G si l'on observe dans G la dérivation $A \overset{\star}{\underset{G}{\Rightarrow}} \alpha$. Le second procédé est décrit dans la section 9.1.5 et consiste à transformer une récursion gauche en récursion droite par ajout d'un nouveau non-terminal.

L'algorithme de construction repose sur une numérotation arbitraire des variables de la grammaire $N = \{A_0, \ldots, A_n\}$, l'axiome recevant conventionnellement le numéro 0. On montre alors que :

Lemme 9.11. *Soit G une grammaire CF. Il existe une grammaire équivalente G' dont toutes les productions sont de la forme :*

- $A_i \to a\alpha$*, avec* $a \in \Sigma$
- $A_i \to A_j\beta$*, et A_j est classé* strictement après *A_i dans l'ordonnancement des non-terminaux ($j > i$).*

La procédure de construction de G' est itérative et traite les terminaux dans l'ordre dans lequel ils sont ordonnés. On note, pour débuter, que si l'on a pris soin de se ramener à une grammaire dont l'axiome est non récursif, ce que l'on sait toujours faire (cf. le résultat de la section 9.1.4), alors toutes les productions dont $S = A_0$ est partie gauche satisfont par avance la propriété du lemme 9.11. Supposons que l'on a déjà traité les non-terminaux numérotés de 0 à $i - 1$, et considérons le non-terminal A_i. Soit $A_i \to \alpha$ une production dont A_i est partie gauche, telle que α débute par une variable A_j. Trois cas de figure sont possibles :

- (a) $j < i$: A_j est classé avant A_i ;
- (b) $j = i$: A_j est égale à A_i ;
- (c) $j > i$: A_j est classé après A_i ;

Pour les productions de type (a), on applique itérativement le résultat du lemme 9.4 en traitant les symboles selon leur classement : chaque occurrence d'un terminal A_j, $j < i$ est remplacée par l'expansion de toutes les parties droites correspondantes. Par l'hypothèse de récurrence, tous les non-terminaux ainsi introduits ont nécessairement un indice strictement supérieur à i. Ceci implique qu'à l'issue de cette phase, toutes les productions déduites de $A_i \rightarrow \alpha$ sont telles que leur partie droite ne contient que des variables dont l'indice est au moins i. Toute production dont le coin gauche n'est pas A_i satisfait par construction la propriété du lemme 9.11 : laissons-les en l'état. Les productions dont le coin gauche est A_i sont de la forme $A_i \rightarrow A_i\beta$, sont donc directement récursives à gauche. Il est possible de transformer les A_i- productions selon le procédé de la section 9.1.5, sans introduire en coin gauche de variable précédant A_i dans le classement. En revanche, cette procédure conduit à introduire de nouveaux non-terminaux, qui sont conventionnellement numérotés depuis $n+1$ dans l'ordre dans lequel ils sont introduits. Ces nouveaux terminaux devront subir le même traitement que les autres, traitement qui est toutefois simplifié par la forme des règles (récursions droites) dans lesquels ils apparaissent. Au terme de cette transformation, tous les non-terminaux d'indice supérieur à $i+1$ satisfont la propriété du lemme 9.11, permettant à la récurrence de se poursuivre.

Notons qu'à l'issue de cette phase de l'algorithme, c'est-à-dire lorsque le terminal d'indice maximal $n+p$ a été traité, G' est débarrassée de toutes les chaînes de récursions gauches : on peut en particulier envisager de mettre en œuvre des analyses descendantes de type LL et s'atteler à la construction de la table d'analyse prédictive correspondante (voir la section 8.1).

Cette première partie de la construction est illustrée par le tableau 9.8.

Considérons maintenant ce qu'il advient après que l'on achève de traiter le dernier non-terminal A_{n+p}. Comme les autres, il satisfait la propriété que le coin gauche de toutes les A_{n+p}-productions est soit un terminal, soit un non-terminal d'indice strictement plus grand. Comme ce second cas de figure n'est pas possible, c'est donc que toutes les A_{n+p}-productions sont de la forme : $A_{n+p} \rightarrow a\alpha$, avec $a \in \Sigma$, qui est la forme recherchée pour la GNF. Considérons alors les A_{n+p-1} productions : soit elles ont un coin gauche terminal, et sont déjà conformes à la GNF ; soit elles ont A_{n+p} comme coin gauche. Dans ce second cas, en utilisant de nouveau le procédé du lemme 9.4, il est possible de remplacer A_{n+p} par les parties droites des A_{n+p}-productions et faire ainsi émerger un symbole terminal en coin gauche. En itérant ce processus pour $i = n+p$ à $i = 0$, on aboutit, de proche en proche, à une grammaire équivalente à celle de départ et qui, de plus, est telle que chaque partie droite de production débute par un terminal. Il reste encore à éliminer les terminaux qui apparaîtraient dans les queues de partie droite (cf. la mise sous CNF) pour obtenir une grammaire qui respecte les contraintes énoncées ci-dessus. □

Pour compléter cette section, notons qu'il existe des variantes plus contraintes de la GNF, qui constituent pourtant également des formes normales. Il est en particulier possible d'imposer, dans la définition du théorème 9.10, que toutes les parties droites de production contiennent au plus deux variables : on parle alors de forme normale de Greibach *quadratique* (Salomaa, 1973, p.202).

État initial	$S \to A_2A_2$ $A_1 \to A_1A_1 \mid a$ $A_2 \to A_1A_1 \mid A_2A_1 \mid b$	
Traitement de A_1	$S \to A_2A_2$ $A_1 \to a \mid aR_1$ $A_2 \to A_1A_1 \mid A_2A_1 \mid b$ $R_1 \to A_1 \mid A_1R_1$	 A_3 est nouveau récursion droite
Traitement de A_2 (première étape)	$S \to A_2A_2$ $A_1 \to aA_1 \mid aA_1R_1$ $A_2 \to aA_1A_1 \mid aA_1R_1A_1 \mid A_2A_1 \mid b$ $R_1 \to A_1 \mid A_1R_1$	
Traitement de A_2 (seconde étape)	$S \to A_2A_2$ $A_1 \to aA_1 \mid aA_1A_3$ $A_2 \to aA_1A_1 \mid aA_1A_3A_1 \mid b \mid aA_1A_1A_4 \mid aA_1A_3A_1A_4 \mid bA_4$ $A_3 \to A_1 \mid A_1A_3$ $A_4 \to A_1 \mid A_1A_4$	
Traitement de A_3	$S \to A_2A_2$ $A_1 \to aA_1 \mid aA_1A_3$ $A_2 \to aA_1A_1 \mid aA_1A_3A_1 \mid b \mid aA_1A_1A_4 \mid aA_1A_3A_1A_4 \mid bA_4$ $A_3 \to aA_1 \mid aA_1A_3 \mid aA_1A_3A_3$ $A_4 \to A_1 \mid A_1A_4$	 A_4 est nouveau
Traitement de A_4	$S \to A_2A_2$ $A_1 \to aA_1 \mid aA_1A_3$ $A_2 \to aA_1A_1 \mid aA_1A_3A_1 \mid b \mid aA_1A_1A_4 \mid aA_1A_3A_1A_4 \mid bA_4$ $A_3 \to aA_1 \mid aA_1A_3 \mid aA_1A_3A_3$ $A_4 \to aA_1 \mid aA_1A_4 \mid aA_1A_3 \mid aA_1A_3A_4$	

TABLE 9.8 – Une grammaire en cours de Greibach-normalisation

Chapitre 10

Notions de calculabilité

Ce chapitre porte sur la notion de langage décidable et semi-décidable. Nous resterons à un niveau assez informel, mais il est possible, en particulier en se fixant un modèle de calcul comme les machines de Turing, de montrer rigoureusement tous les résultats vus ici.

10.1 Décidabilité et semi-décidabilité

Nous définissions la notion de langage décidable et semi-décidable en reprenant et développant les définitions du chapitre 2, plus particulièrement de la section 2.2.2. Les définitions de cette partie sont exprimées en utilisant le terme vague d'*algorithme* : ce terme peut être interprété comme « n'importe quel programme d'un langage de programmation classique (p. ex., C ou Java) qui s'exécuterait dans un environnement sans limitation matérielle (taille mémoire, espace de stockage) ». À la fin de ce chapitre (section 10.4), nous expliquerons comment nous pouvons rendre la notion d'algorithme plus formelle et plus précise, en se donnant un *modèle de calcul*, mécanisme formel pour décrire un processus automatique prenant éventuellement des données en entrée et produisant des données en sortie. On dit qu'un algorithme *termine* (sur l'entrée x) si l'exécution de cet algorithme s'arrête après un nombre fini d'étapes (sur l'entrée x).

On se donne un alphabet Σ.

On commence par introduire les langages semi-décidables comme ceux dont les mots sont énumérés un par un par un algorithme ne prenant aucune entrée :

Définition 10.1. *Un langage $L \subset \Sigma^*$ est* récursivement énumérable *(*r.e.*, ou* semi-décidable*, ou encore* semi-calculable*) s'il existe un algorithme A qui énumère les mots de L.*

Si le langage est infini, un tel algorithme d'énumération ne termine donc pas. On peut donner une définition équivalente des langages semi-décidables, en termes d'algorithmes prenant un mot en entrée et répondant Vrai si le mot appartient au langage.

Proposition 10.2. *Un langage $L \subset \Sigma^*$ est semi-décidable si et seulement s'il existe un algorithme A tel que pour tout mot $u \in \Sigma^*$:*

Chapitre rédigé par Pierre Senellart.

— *si $u \in L$, alors A termine sur u en produisant la sortie* VRAI ;
— *si $u \notin L$, alors soit A termine sur u en produisant la sortie* FAUX, *soit A ne termine pas sur u.*

Démonstration. Pour montrer cette équivalence, fixons-nous un langage L semi-décidable et soit A un algorithme qui énumère L. On construit un algorithme A' ayant les propriétés recherchées de la manière suivante. On modifie A de manière à ce que, à chaque fois que celui-ci produit un mot en sortie, il compare ce mot avec le mot donné en entrée. Si les deux sont différents, on continue ; sinon, on termine en produisant VRAI en sortie.

Réciproquement, soit A un algorithme qui a les propriétés de la proposition. On considère également un algorithme B qui considère l'ensemble des mots de Σ^* et énumère ceux qui appartiennent à L. La principale difficulté est que le nombre de mots de Σ^* est infini, et le nombre d'étapes de calcul à exécuter sur chaque mot n'est pas borné (il peut être infini pour les mots qui ne sont pas dans L, sur lesquels A peut ne pas terminer). On procède en effectuant tous les calculs en parallèle suivant la technique du « déployeur universel » (en anglais, *dovetailing*). La technique est comme suit :

1. On énumère un mot u de Σ^* en utilisant B.
2. On lance la première étape du calcul de A sur u. Pour tous les mots v énumérés jusqu'à présent et dont le calcul de A n'a pas terminé, on lance une étape supplémentaire du calcul de A sur v.
3. Si l'un des calculs de A sur un v s'est terminé en produisant VRAI en sortie, on produit v en sortie.
4. On revient à l'étape 1. □

Les langages semi-décidables sont ainsi ceux qui sont *reconnus* par un algorithme, sans présager du comportement de cet algorithme sur les mots qui ne sont pas dans le langage. Les langages décidables, par contre, ont un algorithme qui décide si oui ou non un mot appartient au langage :

Définition 10.3. *Un langage $L \subset \Sigma^*$ est* récursif *(ou* décidable*, ou* calculable*) s'il existe un algorithme A qui, prenant un mot u de Σ^* en entrée, termine et produit* VRAI *si u est dans L et* FAUX *sinon. On dit alors que l'algorithme A* décide *le langage L.*

Évidemment, tout langage décidable est également semi-décidable ; dans ce qui suit, nous allons voir que tout langage n'est pas nécessairement semi-décidable, et que tout langage semi-décidable n'est pas nécessairement décidable.

10.2 Langages non semi-décidables

Dans cette partie, nous allons montrer qu'il existe des langages non semi-décidables. Ce résultat est assez simple à démontrer et il repose sur des arguments de *cardinalité*, c'est-à-dire, du « nombre » de langages semi-décidables par rapport au « nombre » de langages.

Nous commençons par quelques rappels de mathématiques élémentaires. Étant donné un ensemble E, l'*ensemble des parties* de E, noté 2^E, est l'ensemble de tous les sous-ensembles de E ($X \subset E$ et $X \in 2^E$ sont ainsi synonymes). Étant donnés deux ensembles A et B, on

dit qu'une fonction $f : A \to B$ est *injective* si $\forall (x, y) \in A^2, f(x) = f(y) \Rightarrow x = y$, *surjective* si $\forall x' \in B, \exists x \in A, f(x) = x'$ et *bijective* si elle est à la fois injective et surjective. À une injection $f : A \to B$ on peut naturellement associer une bijection $f^{-1} : f(A) \to A$ définie par $f^{-1}(x') = x \iff f(x) = x'$. S'il existe une injection de A vers B, il existe une surjection de B vers A et vice-versa. La composition de deux injections (respectivement, surjections) est une injection (respectivement, surjection). Un ensemble infini A est dit *infini dénombrable* s'il existe une surjection de $\mathbb{N}$ (l'ensemble des entiers naturels) vers A.

Nous aurons également besoin d'un résultat remarquable sur la relation entre un ensemble et l'ensemble de ses parties, connu sous le nom de théorème de Cantor. Sa preuve utilise un principe *diagonal* (dans l'esprit du *paradoxe du barbier* : le barbier d'une ville rase tous les gens qui ne se rasent pas eux-mêmes ; se rase-t-il lui-même ?).

Théorème 10.4 (Cantor)**.** *Soit E un ensemble quelconque. Il n'existe pas de surjection de E vers 2^E.*

Démonstration. Supposons par l'absurde l'existence d'un ensemble E et d'une surjection f de E vers 2^E. On pose $X = \{ e \in E \mid e \notin f(e) \}$. X est une partie de E ; f étant surjective, il existe $x \in E$ telle que $f(x) = X$. De deux choses l'une :

— soit $x \in X$: par définition de X, $x \notin f(x) = X$, ce qui est une contradiction ;
— soit $x \notin X$, c'est-à-dire, $x \notin f(x)$: par définition de X, $x \in X$, ce qui est une contradiction également. □

Nous sommes maintenant armés pour prouver l'existence de langages non semi-décidables :

Théorème 10.5. *Soit Σ un alphabet. Il existe un langage $L \subset \Sigma^*$ tel que L n'est pas semi-décidable.*

L'idée de la démonstration est de montrer que l'ensemble des langages semi-décidables est infini dénombrable (car ils sont indexés par les algorithmes) tandis que l'ensemble des langages est infini non dénombrable (comme ensemble des parties d'un ensemble dénombrable).

Démonstration. Soit $\mathcal{A}$ l'ensemble de tous les algorithmes énumérant des langages sur Σ. Pour $A \in \mathcal{A}$, on note $f(A)$ le langage énuméré par A. Par définition de la semi-décidabilité, f est une surjection de $\mathcal{A}$ vers l'ensemble des langages semi-décidables. Chaque algorithme de $\mathcal{A}$ a une description finie, par exemple comme une suite d'instructions d'un langage de programmation, ou comme une suite de bits. Autrement dit, chaque algorithme de $\mathcal{A}$ est décrit par un mot d'un certain langage, sur un certain alphabet Σ' (disons, $\Sigma' = \{0, 1\}$ pour une description par suite de bits). Donc $\mathcal{A}$ peut être vu comme un sous-ensemble de Σ'^* : il y a une injection de $\mathcal{A}$ vers Σ'^* et donc une surjection g de Σ'^* vers $\mathcal{A}$. Par ailleurs, il est facile de construire une bijection h de $\mathbb{N}$ vers Σ'^* : on énumère les mots un à un par taille croissante (ϵ, 0, 1, 00, 01, 10, 11, 001...) et on affecte à chaque entier $n \in \mathbb{N}$ le $n+1$-ème mot de cette énumération. Au final, $f \circ g \circ h$ est une surjection de $\mathbb{N}$ vers l'ensemble des langages semi-décidables et ce dernier est infini dénombrable.

Considérons maintenant l'ensemble de tous les langages, 2^{Σ^*}. Supposons par l'absurde que tous les langages soient semi-décidables. Alors d'après la remarque précédente, il existe une surjection $f \circ g \circ h$ de $\mathbb{N}$ vers 2^{Σ^*}. Mais on vient de voir qu'il était possible de construire une bijection φ de $\mathbb{N}$ vers Σ^*. Alors $f \circ g \circ h \circ \varphi^{-1}$ est une surjection de Σ^* vers 2^{Σ^*}, ce qui est absurde d'après le théorème de Cantor. □

Cette preuve montre en fait qu'il existe beaucoup plus de langages non semi-décidables (une infinité non dénombrable) que de langages semi-décidables (une infinité dénombrable). Nous donnerons à la fin de la partie suivante quelques exemples de langages non semi-décidables, mais la plupart des langages non semi-décidables n'ont en fait pas de description en français puisqu'il n'y a qu'un « nombre » infini dénombrable de telles descriptions !

10.3 Langages non décidables

Les langages non semi-décidables ont peu d'intérêt en pratique : il n'existe pas d'algorithme d'énumération pour ces langages donc il est rare de s'y intéresser. Un langage qui serait semi-décidable mais non décidable est par contre beaucoup plus intéressant : on peut énumérer l'ensemble de ses mots, mais on ne peut pas en fournir d'algorithme de reconnaissance. Le mathématicien David Hilbert avait fixé comme l'un des grands défis des mathématiques du 20e siècle le *problème de décision* (*Entscheidungsproblem*) : parvenir à une méthode automatique de calcul permettant de prouver si un résultat mathématique est vrai ou faux. L'existence de langages semi-décidables (donc formellement descriptibles) mais non décidables (donc pour lequel il est impossible de décider de l'appartenance d'un mot) a pour conséquence l'impossibilité de résoudre le problème de décision de Hilbert. Certains langages correspondant à des concepts très utiles en pratique se trouvent être semi-décidables mais non décidables. Nous allons montrer que c'est le cas du langage de l'arrêt.

Le *langage de l'arrêt* est l'ensemble H des mots sur un alphabet donné décrivant les couples d'algorithmes et entrées tels que l'algorithme termine sur cette entrée. Formellement, soit $\Sigma = \{0, 1\}$. Soit $\mathcal{A}$ l'ensemble des algorithmes prenant en entrée un mot u de Σ^*. Quitte à réencoder en binaire les descriptions des algorithmes, on peut voir tout algorithme A de $\mathcal{A}$ comme un mot de Σ^*. Quitte également à ajouter un codage spécial pour les paires, on peut voir toute paire formée d'un algorithme $A \in \mathcal{A}$ et d'un mot $u \in \Sigma^*$ comme un mot $f(A, u)$ de Σ^*, f étant une injection de $\mathcal{A} \times \Sigma^*$ vers Σ^* (on peut faire en sorte que f et f^{-1} soient exprimables par un algorithme de $\mathcal{A}$). Le langage de l'arrêt est l'ensemble $H \subset \Sigma^*$ tel que $x \in H$ si et seulement si il existe $A \in \mathcal{A}$ et $u \in \Sigma^*$ avec $x = f(A, u)$ et *A s'arrêtant sur u*.

La preuve de non-décidabilité du langage de l'arrêt utilise encore une fois un argument diagonal.

Théorème 10.6 (Turing)**.** *Le langage de l'arrêt est semi-décidable mais non décidable.*

Démonstration. Montrons tout d'abord que H, le langage de l'arrêt, est semi-décidable. On construit un algorithme B qui prend en entrée un mot x de Σ^* et procède comme suit :

1. si $x \notin f(\mathcal{A}, \Sigma^*)$, l'algorithme retourne Faux ;
2. sinon, l'algorithme calcule $f^{-1}(x) = (A, u)$ et applique l'algorithme A à l'entrée u ; si ce calcul termine, l'algorithme retourne Vrai.

On a choisi B tel que si $x \in H$, B retourne Vrai sur x (et sinon, B peut soit retourner Faux, soit ne pas terminer) donc H est semi-décidable.

Montrons maintenant que H n'est pas décidable. On procède par l'absurde : soit B un algorithme qui décide H. $\mathcal{A}$ et Σ^* sont deux langages dénombrables (cf. section précédente)

donc on peut numéroter les algorithmes de $\mathcal{A}$ et les mots de Σ^* par A_0, $A_1\ldots$ et u_0, $u_1\ldots$, respectivement. On construit un algorithme B' prenant en entrée un mot u_i de Σ^* de la manière suivante : on commence par appliquer B à $f(A_i, u_i)$. Si B renvoie Faux, on renvoie Faux ; sinon, on part dans une boucle infinie. B' étant un algorithme de $\mathcal{A}$, il existe un entier k tel que $B' = A_k$. Appliquons maintenant l'algorithme de décision B à $f(B', u_k)$. De deux choses l'une :

— B retourne Vrai sur $f(B', u_k)$. Ceci signifie que B' s'arrête sur l'entrée u_k, et donc, par définition de B', que B retourne Faux sur $f(A_k, u_k) = f(B', u_k)$, ce qui est une contradiction.
— B retourne Faux sur $f(B', u_k)$. Donc B' ne s'arrête pas sur l'entrée u_k et donc, comme B s'arrête sur toute entrée, B renvoie Vrai avec l'entrée $f(A_k, u_k) = f(B', u_k)$. Nous aboutissons de nouveau à une contradiction. □

L'argument diagonal, l'idée de coder les algorithmes par des mots du langage et le résultat lui-même rappellent la preuve du théorème d'incomplétude de Gödel, qui est un analogue pour la logique mathématique du théorème de l'arrêt de Turing en logique informatique. Le théorème de Gödel énonce que dans tout système formel permettant d'exprimer l'arithmétique, il existe des énoncés mathématiques que l'on ne peut ni prouver ni infirmer. Le théorème de l'arrêt montre que quand on se fixe un modèle de calcul suffisamment puissant, il existe des propriétés mathématiques de ce modèle de calcul qui ne peuvent être calculées.

Beaucoup de problèmes concrets dans de nombreux domaines de l'informatique sont ainsi décrits par des langages semi-décidables mais non décidables. Ici, le problème « déterminer si $x \in X$ a la propriété P » est décrit par le langage « l'ensemble des $x \in X$ qui ont la propriété P ». Dans tous les exemples ci-dessous, « l'ensemble des $x \in X$ » lui-même est toujours décidable.

— Déterminer si une grammaire hors-contexte est ambiguë.

— Déterminer si la fonction calculée par un programme d'un langage de programmation classique a n'importe quelle propriété non triviale (par exemple, elle ne lève jamais d'exceptions). Ce résultat est le *théorème de Rice*.

— Déterminer si une requête écrite dans le langage d'interrogation de bases de données SQL renvoie un résultat sur au moins une base de données (*théorème de Trakhtenbrot*).

— Étant donnés deux ensembles de mots, déterminer si une séquence de mots du premier ensemble peut être identique à une séquence de mots du second ensemble (problème de *correspondance de Post*).

— Étant données des règles de calcul dans un groupe, déterminer si deux expressions sont égales.

Un exemple de langage non semi-décidable est le complément du langage de l'arrêt. Un tel langage, dont le complément est récursivement énumérable, est dit *co-récursivement énumérable* (noté co-r.e.). On peut construire un langage qui n'est ni r.e. ni co-r.e. en combinant deux langages possédant ces propriétés : par exemple, l'ensemble des couples d'algorithmes dont le premier termine et le second ne termine pas sur une entrée donnée.

10.4 Modèles de calculs

Nous précisons maintenant ce que nous entendons par un *algorithme* dans les parties qui précèdent. Rappelons qu'un algorithme est un procédé de calcul automatique prenant en entrée des données éventuelles (un mot sur un certain alphabet) et produisant en sortie des données (un mot sur un certain alphabet). Une des méthodes les plus pratiques de formaliser cette notion d'algorithme est par le biais des *machines de Turing*, qui peuvent être vues comme des *automates finis à bande*. Nous définissons les machines de Turing dans le cas simple où la sortie est soit VRAI soit FAUX ; on peut facilement étendre la définition au cas où la machine produit une sortie non booléenne, par exemple en rajoutant une seconde bande.

Définition 10.7. *Une* machine de Turing *(déterministe) est la donnée de :*

(i) *un ensemble fini Q d'états ;*
(ii) *un alphabet de travail Γ ;*
(iii) *$b \in \Gamma$ est un symbole spécial « blanc » ;*
(iv) *$\Sigma \subset \Gamma \backslash \{b\}$ est l'alphabet d'entrée/sortie ;*
(v) *q_0 est un état initial ;*
(vi) *$F \subset Q$ est un ensemble d'états finaux ;*
(vii) *$\delta : Q \backslash F \times \Gamma \to Q \times \Gamma \times \{L, R\}$ est une fonction de transition.*

Étant donnée une machine de Turing $(Q, \Gamma, b, \Sigma, q_0, F, \delta)$, un calcul *sur l'entrée $u \in \Sigma^*$ est une séquence de configurations $\{(q_k, \varphi_k, i_k)\}_{1 \leqslant k \leqslant n}$ où $q_k \in Q$ est l'état courant, $\varphi_k : \mathbb{Z} \to \Gamma$ est une fonction associant un symbole à chaque position d'une bande virtuelle infinie, et $i_k \in \mathbb{Z}$ est la position actuelle sur la bande. On impose que la séquence de configurations respecte les propriétés suivantes :*

— *$q_1 = q_0$, $i_1 = 0$, $\varphi_1(x) = u_x$ pour $1 \leqslant x \leqslant |u|$ et $\varphi_1(x) = b$ pour $x \notin [1, |u|]$.*
— *Pour $1 \leqslant k \leqslant n-1$, $\delta(q_k, \alpha_k) = (q_{k+1}, \alpha_{k+1}, \beta)$ avec :*
  — *$i_{k+1} = i_k + 1$ si $\beta = R$, $i_{k+1} = i_k - 1$ sinon ;*
  — *Pour $x \neq i_k$, $\varphi_k(x) = \varphi_{k+1}(x)$;*
  — *Pour $x = i_k$, $\varphi_k(x) = \alpha_k$ et $\varphi_{k+1}(x) = \alpha_{k+1}$.*

Un calcul d'une machine de Turing est dit acceptant *(ou* produisant VRAI*) s'il conduit à un état final,* échouant *(ou* produisant FAUX*) s'il conduit à un état non final depuis lequel il n'existe pas de transition possible.*

De même qu'un automate fini déterministe, une machine de Turing effectue un calcul sur un mot d'entrée pour déterminer si le mot est accepté ou non. Ici, le calcul est plus complexe que pour un automate car les transitions peuvent également dépendre des symboles écrits sur une bande de papier de taille non bornée (décrite par les φ_k) ; noter cependant qu'un calcul acceptant ou échouant donné n'utilisera qu'une partie finie de cette bande.

Pour un exemple de machine de Turing et de son exécution, voir http://ironphoenix.org/tril/tm/ (Palindrome Detector).

Les machines de Turing peuvent être utilisées comme modèle de calcul pour les notions de semi-calculabilité et calculabilité, ce qui donne un cadre formel aux résultats présentés dans les parties précédentes. Elles sont également utilisées pour formaliser les notions de *complexité d'un problème* : la complexité est la longueur du calcul que doit faire une machine de Turing pour résoudre ce problème en fonction de la taille de l'entrée.

De manière très intéressante, un grand nombre d'autres formalismes ont le même *pouvoir d'expression* que les machines de Turing, c'est-à-dire qu'ils permettent de décider exactement les mêmes langages.

λ-calcul. Introduit par Church parallèlement à Turing, ce formalisme, à la base des langages de programmation fonctionnels, est basé sur la définition et l'application de fonctions récursives, décrites dans un langage logique muni de règles de *réduction* permettant de réécrire ces formules logiques.

Fonctions récursives. La théorie des fonctions récursives a été élaborée par Gödel et Kleene comme modèle de fonction calculable basé sur un ensemble de fonctions et primitives de base : fonctions constantes, composition, récursion, etc.

Machines à registres. Une alternative aux machines de Turing plus proche des ordinateurs moderne est celle des machines à registres : ici, la mémoire n'est plus représentée comme une bande infinie, mais comme un ensemble de registres mémoire (ou *variables*) qui peuvent être lus et écrits dans un ordre quelconque. L'*architecture de von Neumann*, utilisée pour concevoir les premiers ordinateurs, est un exemple de ces machines à registres.

Langages de programmation Turing-complets. La plupart des langages utilisés pour développer des logiciels [1] sont *Turing-complets* : si on suppose qu'ils sont exécutés dans un environnement sans limite de mémoire, ils permettent de décider n'importe quel langage décidable par une machine de Turing.

Les grammaires génératives de type 0, introduites par Chomsky, ne sont pas un modèle de calcul à proprement parler, mais engendrent exactement les langages semi-décidables.

Cette remarquable équivalence de modèles formels aussi divers de ce que peut être un processus automatique de calcul a conduit Church et Turing à formuler ce qu'on appelle la *thèse de Church–Turing* : ce qui est intuitivement calculable (ou ce qui est *humainement* calculable), c'est ce qui est calculable par une machine de Turing (ou par le λ-calcul, etc.).

1. Les exceptions sont quelques rares langages qui sont utilisés pour des buts spécifiques, comme SQL sans les fonctions utilisateurs pour l'interrogation de bases de données relationnelles, ou XPath pour la navigation dans les arbres XML.

Deuxième partie

Annexes

Chapitre 11

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.

11.1 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)

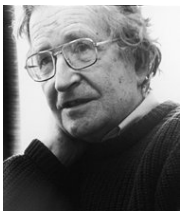
est un chercheur en théorie des automates, qui fut l'étudiant de Edward J. McCluskey. Il a apporté de nombreuses contributions dans ce domaine, et est en particulier connu pour l'*algorithme de minimisation de Brzozowski* et pour l'algorithme de Brzozowski et McCluskey (section 4.2.2).

Chapitre rédigé par Pierre Senellart.

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 λ-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 (λ-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)

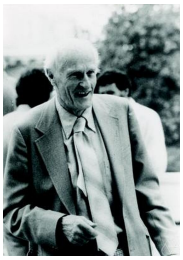
est une chercheuse en langages formels. Elle est en particulier connue pour la *forme normale de Greibach* d'une grammaire hors-contexte et ses conséquences sur l'équivalence entre grammaires hors-contexte et automates à pile.

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 20e 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'épitaphe de David Hilbert résume sa vision de la science :

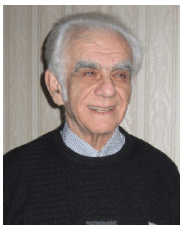
Wir müssen wissen.	*(Nous devons savoir.)*
Wir werden wissen.	*(Nous saurons.)*

Stephen C. Kleene (Américain, 1909 – 1994)



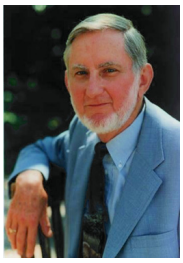
est l'inventeur des expressions rationnelles et a apporté des contributions majeures à la théorie des langages récursifs, un modèle de calculabilité. L'opérateur d'*étoile*, caractéristique des expressions rationnelles, porte son nom, de même que le théorème d'équivalence entre expressions rationnelles 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.

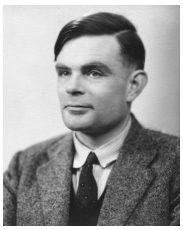
John von Neumann (Hongro-Américain, 1903 – 1957)

, 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.

11.2 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 : Eriktj (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 12

Correction des exercices

12.1 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 $L_1 = (I_1\Sigma^\star \cap \Sigma^\star F_1) \setminus \Sigma^\star T_1 \Sigma^\star$.
   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 $aa^\star + ab(ab)^\star$.

3. Pour l'intersection, il suffit d'utiliser le fait que $A \setminus B = A \cap \overline{B}$. $L_1 \cap L_2$ 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 :
   $$L_1 \cap L_2 = ((I_1 \cap I_2)\Sigma^\star \cap \Sigma^\star(F_1 \cap F_2)) \setminus (\Sigma^\star(T_1 \cup T_2)\Sigma^\star)$$
   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, $\Sigma^\star$ également ; les rationnels sont stables par concaténation, intersection, complément. Ceci suffit pour conclure que tout langage de la forme $(I\Sigma^\star \cap \Sigma^\star F) \setminus \Sigma^\star T \Sigma^\star$ 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 $A = (\Sigma, Q, q_o, F, \delta)$ un automate fini déterministe reconnaissant L. On définit L' sur l'alphabet $\Sigma' = Q \times \Sigma \times Q$ par les trois ensembles :

   — $I = \{(q_0, a, \delta(q_o, a)), a \in \Sigma\}$

   — $F = \{(q, a, r), q \in Q, a\Sigma, r \in \delta(a, q) \cap F\}$

   — $\Sigma' \times \Sigma' \setminus \{(q, a, r)(r, b, p), q \in Q, a, b\Sigma, r \in \delta(a, q), b \in \delta(r, b)\}$.

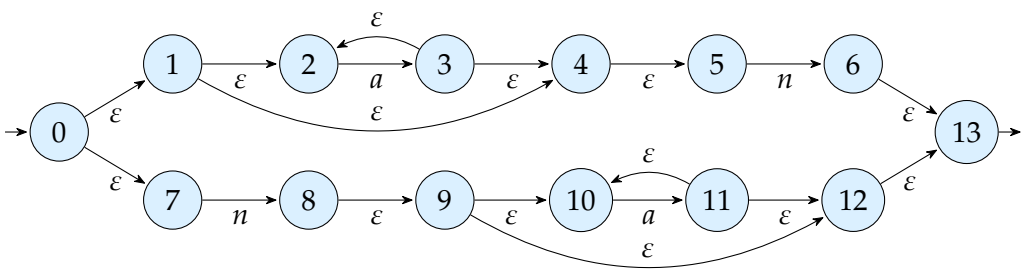
   Soit maintenant un mot u de L. Son calcul dans A est par construction un mot w de L' qui vérifie de plus $\phi(w) = u$, donc $L \subset \phi(L')$. Réciproquement, si $u = \phi(w)$, avec w dans L', alors on déduit immédiatement un calcul pour u dans A, et donc $u \in L$.

12.2 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 $b(c + a)^\star d$, soit $\Sigma^\star b(c + a)^\star d$.
3. À faire...

12.3 Correction de l'exercice 4.14

1. La construction de Thompson conduit à l'automate 12.1.



Automate 12.1 – Automate d'origine pour $(a^\star n \mid na^\star)$

État	Fermeture
0	0, 1, 2, 4, 5, 7
1	1, 2, 4, 5
2	2
3	2, 3, 4, 5
4	4, 5
5	5
6	6, 13
7	7
8	8, 9, 10, 12, 13
9	9, 10, 12, 13
10	10
11	10, 11, 12, 13
12	12, 13
13	13

TABLE 12.2 – ε-fermetures (avant) des états de l'automate 12.1

Pour chaque état, l'ensemble des états accessibles par transitions spontanées.

2. Le calcul de l'ε-fermeture conduit aux résultats du tableau 12.2.
3. La suppression des transitions ε 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 12.2), ajouter une transition $\delta(p, a) = q'$ pour tout état p tel que q appartienne à l'ε-fermeture avant de p. Marquer ensuite comme final tous les états dont la fermeture avant contient un état final.

État	Fermeture
0	0
1	0, 1
2	0, 1, 2, 3
3	3
4	0, 1, 3, 4
5	0, 1, 3, 4, 5
6	6
7	0, 7
8	8
9	8, 9
10	8, 9, 10, 11
11	11
12	8, 9, 11, 12
13	6, 8, 9, 11, 12, 13

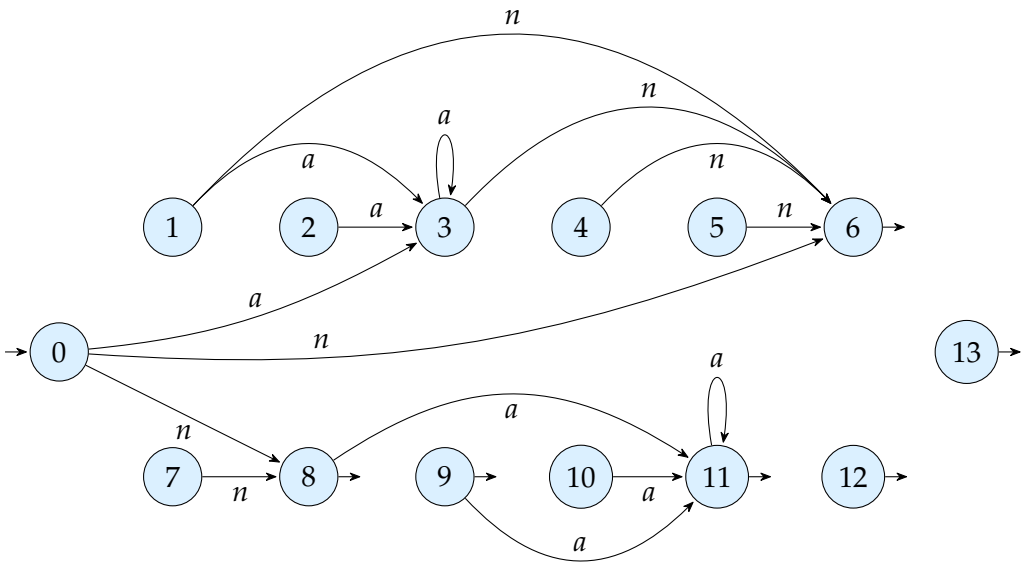
TABLE 12.3 – ε-fermetures arrières des états de l'automate 12.1

Pour chaque état, l'ensemble des états co-accessibles par transitions spontanées.

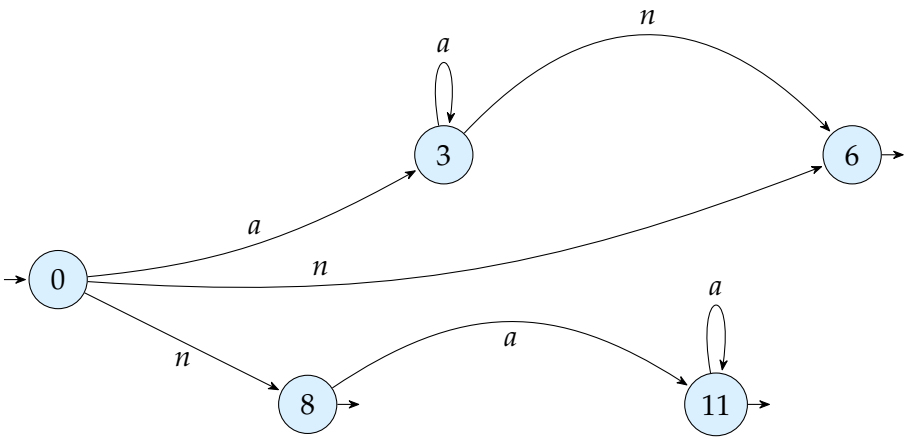
   Alternativement, en utilisant le tableau des clôtures arrières (tableau 12.3), il faut donc ajouter une transition $\delta(p, a) = q'$ pour tout état p dans l'ε-fermeture arrière de q. On marque ensuite comme final tous les états dans la fermeture arrière des états final.

   Cette procédure aboutit à l'automate 12.4.

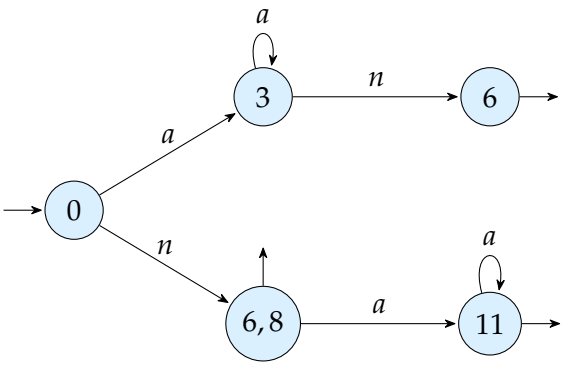
   Les états 1, 2, 4, 5, 7, 9, 10, 12 et 13 sont devenus inutiles dans l'automate 12.4, si on les élimine, on obtient l'automate 12.5. Cet automate n'est pas déterministe, l'état 0 ayant deux transitions sortantes pour le symbole n.
4. Après déterminisation par la méthode des sous-ensembles, on aboutit finalement à l'automate 12.6.



Automate 12.4 – Automate sans transition spontanée pour $(a^\star n \mid na^\star)$



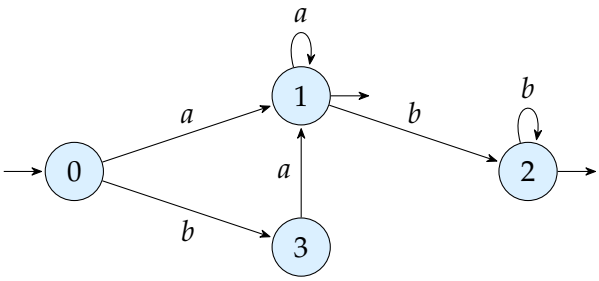
Automate 12.5 – L'automate 12.4, émondé



Automate 12.6 – Automate déterministe pour $(a^\star n \mid na^\star)$

12.4 Correction de l'exercice 4.15

1. Après élimination de la transition spontanée, la méthode de construction des sous-ensembles permet de construire l'automate 12.7.



Automate 12.7 – Déterminisation de l'automate 4.14

12.5 Correction de l'exercice 4.19

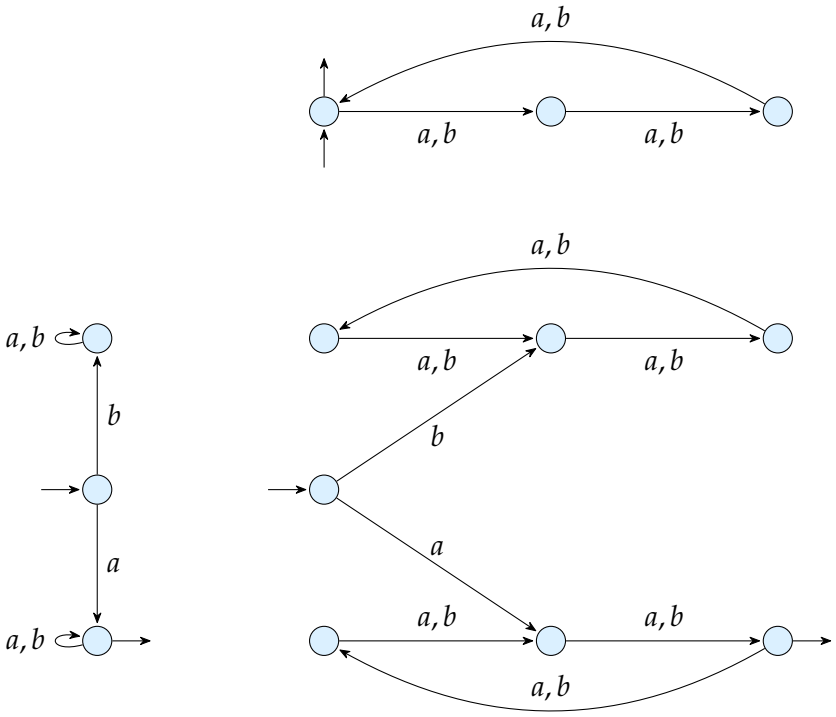
1. Les automates A_1 et A_2 sont représentés respectivement verticalement et horizontalement dans la représentation de l'automate 12.8.
2. L'application de l'algorithme construisant l'intersection conduit à l'automate 12.8.
3. a. L'algorithme d'élimination des transitions ε demande de calculer dans un premier temps l'ε-fermeture de chaque état. Puisque les seules transitions ε sont celles ajoutées par la construction de l'union, on a :
   — $\varepsilon - closure(q_0) = \{q_0^1, q_0^2\}$
   — $\forall q \in Q^1 \cup Q^2, \varepsilon - closure(q) = \{q\}$

   L'élimination des deux transitions ε 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.

   b. 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 $\overline{A}$ sont identiques aux transitions de la construction directe de l'intersection.



Automate 12.8 – 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 $\overline{A_2}$), 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 finals 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 ε, 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 ;

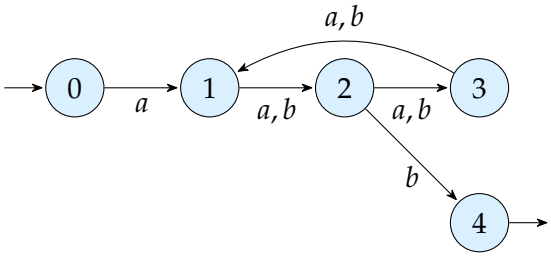
   — soit elle est construite et on peut alors rendre cette partie comme état initial et supprimer q_0 : tous les calculs réussis depuis q_0 seront des calculs réussis depuis $\{q_0^1, q_2^0\}$, et réciproquement, puisque ces deux états ont les mêmes transitions sortantes.

12.6 Correction de l'exercice 4.20

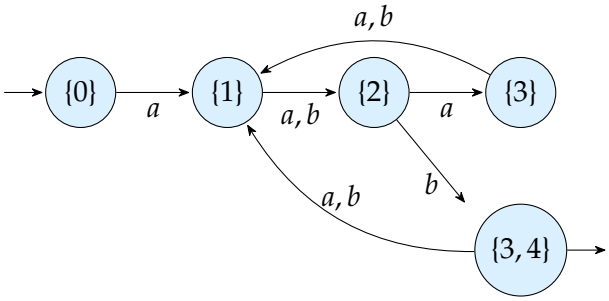
Le langage contenant les mots dont la longueur est divisible par 3 est reconnu par un automate à trois états, chaque état correspondant à un reste de la division par 3. De 0 on transite vers 1 (indépendamment de l'entrée) ; de 1 on transite vers 2, et de 2 vers 0, qui est à la fois initial et final.

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 12.9 non-déterministe, qui correspond formellement au produit des automates intersectés. L'application de la méthode des sous-ensembles conduit à A_2, l'automate 12.10, déterministe.



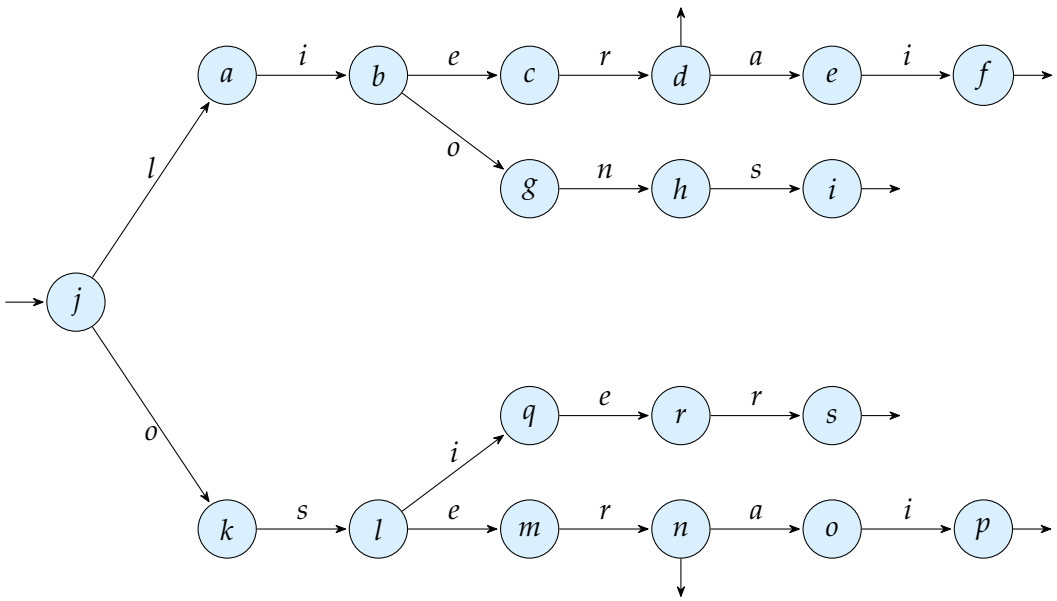
Automate 12.9 – L'automate A_1 de la question 1



Automate 12.10 – L'automate A_2 de la question 1

12.7 Correction de l'exercice 4.35

1. En utilisant les méthodes du cours, on obtient l'automate 12.11, 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 ε permet de les distinguer (i.e. qu'il conduit à un calcul réussi depuis (f) mais pas depuis (e)).



Automate 12.11 – 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.

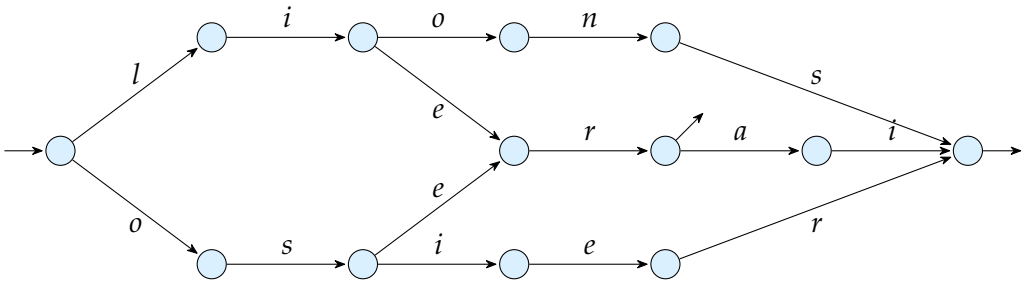
   Commençons par l'initialisation : $\{d, f, i, n, s, p\}$ sont distingués de tous les autres états par leur « finalitude ». On distingue deux sous-ensembles : $\{f, i, s, p\}$ sont distingués de d et de n par le mot ai, mais sont indistinguables entre eux. À ce stade, on ne peut conclure sur d et n.

   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 12.12.



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 12.12 – Le dictionnaire déterminisé et minimisé

12.8 Correction de l'exercice 5.21

2 L'application de la méthode de transformation d'automate en grammaire présentée en section 5.2.4 conduit à la grammaire 12.13, ou bien encore à la grammaire 12.14 si l'on part de l'automate déterministe.

$$
\begin{aligned}
S &\rightarrow A \mid bA \\
A &\rightarrow aA \mid aB \\
B &\rightarrow bB \mid \varepsilon
\end{aligned}
$$

Grammaire 12.13 – Grammaire régulière pour l'automate (non-déterministe)

$$
\begin{aligned}
S &\rightarrow aA \mid bB \\
A &\rightarrow aA \mid bC \\
B &\rightarrow aA \\
C &\rightarrow bC \mid \varepsilon
\end{aligned}
$$

Grammaire 12.14 – Grammaire régulière pour l'automate (déterministe)

Troisième partie

Références

Liste des Algorithmes

4.2 Reconnaissance par un DFA 33

7.1 Parsage ascendant en profondeur d'abord 94

7.3 Parsage descendant en profondeur d'abord 96

8.7 Calcul de FIRST 105

8.8 Calcul de FOLLOW 107

8.10 Analyseur pour grammaire LL(1) 110

8.21 Reconnaissance ascendante guidée par un automate $A = (\Sigma \cup N, Q, q_0, F, \delta)$. Version 1 117

8.22 Reconnaissance ascendante guidée par un automate $A = (\Sigma \cup N, Q, q_0, F, \delta)$. Version 2 118

8.30 Construction de l'automate d'analyse LR(1) 127

Chapitre 13

Liste des automates

4.1 Un automate fini (déterministe) 32
4.3 Un automate fini déterministe comptant les a (modulo 3) 34
4.4 Un automate fini déterministe équivalent à l'automate 4.1 34
4.5 Un automate fini déterministe pour $ab(a+b)^\star$ 35
4.6 Un automate partiellement spécifié 36
4.7 Un automate non-déterministe 39
4.8 Un automate à déterminiser 40
4.9 Le résultat de la déterminisation de l'automate 4.8 40
4.10 Un automate difficile à déterminiser 41
4.11 Un automate potentiellement à déterminiser 42
4.12 Un automate avec transitions spontanées correspondant à $a^\star b^\star c^\star$ 43
4.13 L'automate 4.12 débarrassé de ses transitions spontanées 43
4.14 Automate non-déterministe A 44
4.15 Automate de Thompson pour $\emptyset$ 49
4.16 Automate de Thompson pour ε 49
4.17 Automate de Thompson pour a 49
4.18 Automate de Thompson pour $e_1 + e_2$ 49
4.19 Automate de Thompson pour $e_1 e_2$ 50
4.20 Automate de Thompson pour $e_1{}^\star$ 50
4.21 Automate de Thompson pour $(a+b)^\star b$ 50
4.22 Illustration de BMC : élimination de l'état q_j 52

4.23 Un DFA à minimiser 59
4.24 L'automate minimal de $(a + b)a^\star ba^\star b(a + b)^\star$ 59
4.25 Un petit dictionnaire 60

8.20 L'automate des réductions licites 117
8.29 Les réductions licites pour la grammaire 8.28 126

12.1 Automate d'origine pour $(a^\star n \mid na^\star)$ 162
12.4 Automate sans transition spontanée pour $(a^\star n \mid na^\star)$ 164
12.5 L'automate 12.4, émondé 164
12.6 Automate déterministe pour $(a^\star n \mid na^\star)$ 164
12.7 Déterminisation de l'automate 4.14 165
12.8 Les automates pour A_1 (à gauche), A_2 (en haut) et leur intersection (au milieu) 166
12.9 L'automate A_1 de la question 1 167
12.10 L'automate A_2 de la question 1 167
12.11 Le dictionnaire déterminisé 168
12.12 Le dictionnaire déterminisé et minimisé 169

Chapitre 14

Liste des grammaires

5.1 G_1, une grammaire pour $a^n b^n$ 64
5.2 Une grammaire pour $a^n b^n c^n$ 68
5.4 Une grammaire contextuelle pour $a^n b^n$ 70

6.1 La grammaire G_D des repas dominicaux 78
6.3 Constructions élémentaires de Bash 80
6.7 Une grammaire pour les sommes 84
6.9 Une grammaire ambiguë 85

8.1 Fragments d'un langage de commande 100
8.2 Une grammaire LL(1) simple 101
8.5 Une grammaire pour les expressions arithmétiques 103
8.6 Une grammaire (ambiguë) pour $(a \mid b)^\star c$ 105
8.11 Une grammaire non-LL(1) pour *if-then-else* 110
8.12 Une grammaire factorisée à gauche pour *if-then-else* 110
8.13 Une grammaire (simplifiée) pour les expressions arithmétiques 112
8.14 Une grammaire LL pour les expressions arithmétiques 112
8.18 Une grammaire pour $a^\star bb^\star a$ 114
8.26 Une grammaire non-LR(0) 122
8.28 Une grammaire pour les manipulations de pointeurs 125

9.1 Élimination des productions inutiles : l'ordre importe 132

9.2 Une grammaire contenant des productions non-génératives 133
9.3 Une grammaire débarrassée de ses productions non-génératives 134
9.4 Une grammaire avant et après élimination des productions ε 135
9.5 Une grammaire pour les expressions arithmétiques 136
9.6 Une grammaire à Chomsky-normaliser 138
9.7 Une grammaire Chomsky-normalisée 138

12.13 Grammaire régulière pour l'automate (non-déterministe) 170
12.14 Grammaire régulière pour l'automate (déterministe) 170

Chapitre 15

Liste des tableaux

2.1 Métamorphose de chien en chameau 19

3.1 Identités rationnelles 26
3.2 Définition des motifs pour `grep` 28

5.3 Des dérivations pour $a^n b^n c^n$ 69
5.5 Grammaire et automate pour $aa(a + b)^\star a$ 71

6.2 Louis boude 78
6.4 Dérivation du nombre `3510` 80
6.5 Constructions parenthésées 81

8.3 Étapes de l'analyse de aacddcbb 101
8.4 Table d'analyse prédictive 102
8.9 Table d'analyse prédictive pour la grammaire 8.6 108
8.15 Calcul de FIRST et FOLLOW 113
8.16 Table d'analyse prédictive pour la grammaire de l'arithmétique 113
8.17 Trace de l'analyse déterministe de $2 * (1 + 2)$ 114
8.19 Analyse ascendante de $u = abba$ 115
8.23 Une table d'analyse LR(0) 119
8.27 Tables d'analyse LR(0) et SLR(1) de la grammaire 8.26 124

9.8 Une grammaire en cours de Greibach-normalisation 141

12.2 ε-fermetures (avant) des états de l'automate 12.1 163
12.3 ε-fermetures arrières des états de l'automate 12.1 163

Chapitre 16

Table des figures

6.6 L'arbre de dérivation de *Paul mange son fromage* 84
6.8 Deux arbres de dérivation d'un même calcul 85
7.2 Recherche ascendante 95
8.24 Une branche de l'automate 120
8.25 Deux branches de l'automate 121

Chapitre 17

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/. 11.2

Conway, M. E. (1963). Design of a separable transition-diagram compiler. *Commun. ACM*, 6:396–408. 8.2.4

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 progès récents en preuve automatique. Grand prix de philosophie 2007 de l'Académie française. 11.2

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 20e siecle. 11.2

Grune, D. et Jacob, C. J. (1990). *Parsing Techniques : a practical Guide*. Ellis Horwood. 7

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*. 11.2

Hopcroft, J. E. et Ullman, J. D. (1979). *Introduction to automata theory, languages and computation*. Addison-Wesley. 4

Knuth, D. E. (1965). On the translation of languages from left to right. 8.2.4

Lewis, II, P. M. et Stearns, R. E. (1968). Syntax-directed transduction. *J. ACM*, 15:465–488. 8.2.4

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. 11.2

Sakarovitch, J. (2003). *Eléments de théorie des automates*. Vuibert, Paris. 4

Salomaa, A. (1973). *Formal Languages*. Academic Press. 9.2.2

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. 11.2

Chapitre 18

Index

Symbols

Σ, 13
Σ^+, 14
$\Sigma^\star$, 14
$|u|$, 13
$|u|_a$, 14
ε, 13
ε-NFA, 42
ε-fermeture
- arrière
  - d'un automate, 43
- avant
  - d'un automate, 43
- d'un état, 43

A

accepter, 114
Alan Turing, 158
algorithme
- d'élimination des états, 52
- de Brzozowski et McCluskey, 52
- de Glushkov, 50
- de Moore, 59
- de Thompson, 50

Alonzo Church, 154
alphabet, 13
ambiguïté, 84
analyse
- lexicale, 11
- syntaxique, 11

appariement, 100
arbre
- d'analyse, 83
- de dérivation, 83

Augustus De Morgan, 155
automate
- canonique, 57
- complet, 36
- déterminisé, 39
- fini déterministe, 35
- à transitions spontanées, 42
- émondé, 37
- équivalent, 34
- fini déterministe (complet), 31
- généralisé, 51
- non-déterministe, 38

axiome, 63

B

Backus, John, 153
Backus-Naur Form, 80
bijective, 145
Brzozowski, Janusz, 153

C

calcul, 148
- acceptant, 148
- dans un automate, 32
- réussi, 32
- échouant, 148

Cantor, Georg, 154
CFG, 69
Chomsky, Noam, 154
Church, Alonzo, 154
co-récursivement énumérable, 147
coin gauche, 79
complémentation
- d'automate, 45

complexité d'un problème, 148
concaténation
- d'automates, 47
- de langages, 21
- de mots, 14
congruence
- droite, 22, 56
construction
- des sous-ensembles, 41
contexte LR(0), 116

D

David Hilbert, 156
De Morgan, Augustus, 155
destination
- d'une transition, 31
DFA, 31
distance
- d'édition, 18
- de Levenshtein, 18
- préfixe, 17
décalage, 113
dérivation, 64
- droite, 82
- gauche, 82
- immédiate, 64
- non-générative, 133
déterministe
- analyseur, 99

E

Edward F. Moore, 157
Edward J. McCluskey, 157
Entscheidungsproblem, 146
état, 31
- accessible, 37
- co-accessible, 37
-s distinguables, 57
- final, 31
-s indistinguables, 57
- initial, 31
- puits, 35
- utile, 37
étiquette
- d'un calcul, 32
- d'une transition, 31
étoile
- d'automate, 48
expression rationnelle, 24
- équivalente, 26

F

facteur
- propre, 16
factoriser à gauche, 109
fonction
- de transition, 31
forme normale
- de Chomsky, 137
- de Greibach, 139

G

Georg Cantor, 154
Glushkov, Victor, 155
grammaire
- CS, 66
- RG, 70
- à choix finis, 73
- algébrique, 69
- ambiguë, 84
- contextuelle, 66
- équivalente, 65, 86
- faiblement équivalente, 86
- FC, 73
- fortement équivalente, 86
- générative, 65
- hors-contexte, 69
- linéaire, 73
- linéaire à droite, 72
- linéaire à gauche, 73
- LL(1), 109
- LL(k), 111
- LR(0), 122
- LR(1), 125
- LR(k), 128
- monotone, 66
- régulière, 70
- sensible au contexte, 66
- SLL(1), 102
- SLR(1), 123
sous-grammaire, 130
- syntagmatique, 63
graphe
- d'une grammaire, 92

Greibach, Sheila, 155
Gödel, Kurt, 155

H
Hilbert, David, 156

I
infini dénombrable, 145
injective, 145
intersection
- d'automates, 46
item, 120

J
Janusz Brzozowski, 153
John Backus, 153
John von Neumann, 157

K
Ken Thompson, 158
Kleene, Stephen C., 156
Kurt Gödel, 155

L
langage, 14
- ambigu, 85
- contextuel, 68
- de l'arrêt, 146
- des facteurs, 22
- des préfixes, 21
- des suffixes, 22
- engendré
  - par un non-terminal, 129
  - par une grammaire, 64
- hors contexte, 70
- préfixe, 22
- rationnel, 24
- reconnaissable, 33
- reconnu, 32
- récursif, 15
- récursivement énumérable, 14
- régulier, 71

lemme
- de l'étoile, 53
- de pompage
  hors-contexte, 87
  rationnel, 53

Levenshtein, Vladimir, 156
longueur
- d'un mot, 13

M
machine de Turing, 148
McCluskey, Edward J., 157
monoïde, 14
- libre, 14

Moore, Edward F., 157
morphisme, 22
mot, 13
- conjugué, 16
-s distinguables d'un langage, 55
- facteur, 15
-s indistinguables d'un automate, 56
-s indistinguables d'un langage, 55
- miroir, 16
- préfixe, 15
- primitif, 16
- suffixe, 15
- transposé, 16
- vide, 13

N
Naur, Peter, 157
NFA, 38
Noam Chomsky, 154
non-terminal, 63
- utile, 131
-directement récursif, 135

O
ordre
- alphabétique, 17
- bien fondé, 17
- lexicographique, 17
- militaire, 17
- partiel, 16
- radiciel, 17
- total, 17

origine
- d'une transition, 31

P
palindrome, 16
parsage, 83
partie
- droite, 63

- gauche, 63
Peter Naur, 157
Post
Correspondance de -, 147
problème de décision, 146
production, 63
- inutile, 131
- non-générative, 132
- pointée, 120
- récursive, 79
- utile, 131
produit
- d'automates, 46
- de langages, 21
proto-mot, 64
pumping lemma
context-free, 87
rational, 53

Q

quotient
- droit
- d'un langage, 22
- d'un mot, 16

R

relation
- d'ordre, 16
- d'équivalence invariante à droite, 56
récursivement énumérable, 143
réduction, 113

S

semi-calculable, 143
semi-décidable, 143
semi-groupe, 14
Sheila Greibach, 155
Simple LL, 102
sous-mot, 15
star lemma, 53
Stephen C. Kleene, 156
surjective, 145
symbole
- initial, 63
- pré-terminal, 78
sémantique, 84

T

table d'analyse
- LR(0), 122
- prédictive, 101
terminal, 63
théorème
- de Kleene, 52
- de Rice, 147
- de Trakhtenbrot, 147
Thompson, Ken, 158
thèse de Church–Turing, 149
transformation syntaxique, 27
transition, 31
- spontanée, 42
transposition
- d'automate, 47
Turing, Alan, 158
Turing-complet, 149

U

union
- d'automates, 45

V

variable, 63
Victor Glushkov, 155
Vladimir Levenshtein, 156
vocabulaire, 63
von Neumann, John, 157