Théorie des langages

Notes de cours

François Yvon et Akim Demaille

Novembre 2005

Première partie

Bases

Avertissement au lecteur :

Ces notes documentent le cours de théorie des langages enseigné dans le cadre de la BCI d'informatique.

Ces notes sont, malgré nos efforts, 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@enst.fr, akim@epita.fr).

Table des matières

Chapitre 1

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 1.2), 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.

1.1 Quelques langages « réels »

1.1.1 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 que finies 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 effectuer 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.

1.1.2 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).

A 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 sont aussi utilisées pour les protéines. En effet, la structure primaire des protéines 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.

1.1.3 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 son 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.

1.2 Terminologie

1.2.1 Bases

Étant donné un ensemble *fini* de symboles Σ, que l'on appelle l'*alphabet* , on appelle *mot* toute séquence finie (éventuellement vide) d'éléments de Σ. Par convention, le mot vide est notée ε. 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). Par convention, $|\varepsilon| = 0$. 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). εest l'élément neutre pour la concaténation : $u\varepsilon = \varepsilon u = u$. 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 de 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 $\Sigma = \{a, b, c\}$. $\Sigma^\star$ vaut alors $\Sigma^\star = \{\varepsilon, a, b, c, ab, ba, \ldots\}$

[1] Pas toutes : penser aux formes conjuguées, aux noms propres, aux emprunts, aux néologisme, aux argots...

– $\{\varepsilon, a, b, c\}$, soit tous les mots de longueur strictement inférieure à 2 ;
– $\{ab, acb, aab, acab, aabb, \ldots\}$, soit tous les mots qui commencent par un a et finissent par un b ;
– $\{\varepsilon, ab, aabb, aaabbb, \ldots\}$, soit tous les mots contenant n a suivis d'autant de b. Ce langage est noté $\{a^n b^n, n \geq 0\}$;
– $\{\varepsilon, abc, aabbcc, aaabbbccc, \ldots\}$, soit tous les mots contenant n occurrences de la lettre a, suivies d'un nombre identique d'occurrences de la lettre b, suivies d'autant de fois la lettre c. Ce langage est noté $\{a^n b^n c^n, n \geq 0\}$.
– $\{\varepsilon, a, aa, aaa, aaaaa \ldots\}$, tous les mots composées 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 non-dénombrable. 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.

1.2.2 Quelques notions de calculabilité

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

Définition 1.1 (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 1.2 (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. Dans la pratique, 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
– ...

1.3 Opérations sur les mots

1.3.1 Facteurs et sous-mots

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 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 est plus grand que $|u|$, $pref_k(u)$ 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 $u = u_1 \dots u_n$, où les u_i sont dans Σ, est défini par : $u^R = u_n \dots u_1$. 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.

1.3.2 Distances entre mots

Une famille de distances

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

[2]Attention : il y a ici désaccord entre les terminologies françaises et anglaises : *subword* signifie en fait *facteur* et c'est *scattered subword* qui est l'équivalent anglais de notre *sous-mot*.
[3]On parle aussi en linguistique de *terminaison* au lieu de suffixe.
[4]Ici, rien à voir avec la conjugaison des grammairiens.

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

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

définit une distance sur $\Sigma^\star$. 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 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). 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^{eme} symbole de u : $u(i) = u_i$. A 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 :

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

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 moins 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 Levinshtein, 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 la Table 1.1.

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

TAB. 1.1 – Métamorphose de chien en chameau
Deux mots consécutifs sont à distance 1. L'ordre des opérations élémentaires est arbitraire.

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.

1.3.3 Ordres sur les mots

Nous avons vu qu'on pouvait définir des ordres partiels sur $\Sigma^\star$ en utilisant les relations de préfixe, suffixe... Il est possible de définir des ordres *totaux* sur $\Sigma^\star$, à la condition de disposer d'un ordre total $\prec$ sur Σ. À cette condition, l'ordre *lexicographique* sur $\Sigma^\star$ noté $\prec_l$ est défini par $u \prec_l v$ 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 $\prec$.
Cet ordre est celui traditionnellement utilisé dans les dictionnaires ; il conduit toutefois à des résultats contre-intuitifs lorsque l'on manipule des langages infinis.

L'ordre *radiciel* (ou militaire) $\prec_a$ utilise également $\prec$, mais privilégie, lors des comparaisons, la longueur des chaînes : $u \prec_a v$ si et seulement si :
– $|u| < |v|$
– $|u| = |v|$ et $u \prec_l v$
Ainsi est-on assuré, lorsque l'on utilise l'ordre alphabétique que (i) il existe un nombre fini de mots plus petits qu'un mot arbitraire u ; (ii) que pour tout w, w' si $u \prec_a v$ alors $wuw' \prec_a wvw'$.

1.3.4 Quelques résultats combinatoires élémentaires

La propriété suivante est trivialement respectée :

[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.

Lemme 1.1.

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

Cette propriété est illustrée sur la figure suivante :

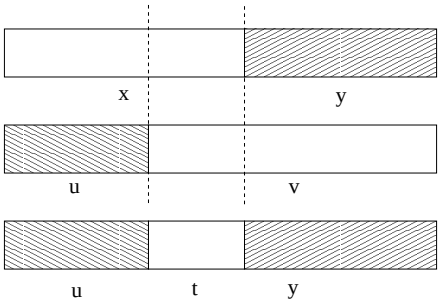


FIG. 1.1 – Illustration d'un résultat combinatoire simple

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

Théorème 1.1. *Si $xy = yz$, avec $x \neq \varepsilon$, alors $\exists u, v \in \Sigma^\star$ et un entier $k \geq 0$ tels que : $x = uv$, $y = (uv)^k u = u(vu)^k$, $z = vu$.*

Preuve : 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 $|y| > |x|$ 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 1.2. *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 $y = u^j$.*

Preuve : 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.

1.4 Opérations sur les langages

1.4.1 Opérations ensemblistes

Les langages étant des ensembles de séquences, 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 $\mathcal{P}(\Sigma^\star)$ par :

– $L_1 \cup L_2 = \{u \in \Sigma^\star, u \in L_1 \text{ ou } u \in L_2\}$; $\cup$ est une opération commutative et associative
– $L_1 \cap L_2 = \{u \in \Sigma^\star, u \in L_1 \text{ et } u \in L_2\}$; $\cap$ est une opération commutative et associative
– $\overline{L} = \{u \in \Sigma^\star, u \notin L\}$
L'opération de concaténation (on dit également le *produit* , mais ce n'est pas le produit cartésien), définie sur les mots, se généralise naturellement aux langages par :
– $L_1 L_2 = \{u, \exists (x, y) \in L_1 \times L_2 \text{ tq. } u = xy\}$
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 $\{u \in \Sigma^\star, \exists v \in L, u = v^n\}$.

L'opération de *fermeture de Kleene* (ou plus simplement *l'étoile*) d'un langage L se définit par :
– $L^\star = \bigcup_{i \geq 0} L^i$
On définit également $L^+ = \bigcup_{i \geq 1} L^i$: à la différence de $L^\star$ qui contient toujours ε, L^+ ne contient εque si L le contient. On a : $L^+ = LL^\star$.

$L^\star$ 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 fini, $\Sigma^\star$, 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.

Un des intérêts de ces notations est qu'elles permettent d'exprimer de manière formelle (et compacte) des langages complexes, éventuellement infinis, à partir de langages plus simples. Ainsi l'ensemble des suites de 0 et de 1 contenant la séquence 111 s'écrira par exemple : $\{0,1\}^*\{111\}\{0,1\}^*$, la notation $\{0,1\}^*$ permettant un nombre arbitraire de 0 et de 1 avant la séquence 111. La notion d'expression rationnelle, introduite au chapitre 2, développe cette intuition.

1.4.2 Plus d'opérations dans $\mathcal{P}(\Sigma^\star)$

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

Définition 1.3 (Ensemble des préfixes). *Soit L un langage de $\Sigma^\star$, on définit l'ensemble des préfixes de L, noté $Pref(L)$ par :*

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

On dit qu'un langage L est un langage *préfixe* si pour tout u et v dans L, on a $u \notin Pref(v)$ et $v \notin Pref(u)$. En utilisant un langage préfixe fini, il est possible définir un procédé de codage donnant lieux 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.

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

Définition 1.4 (Ensemble des suffixes). *Soit L un langage de $\Sigma^\star$, on définit l'ensemble des suffixes de L, noté $Suff(L)$ par :*

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

Définition 1.5 (Ensemble des facteurs). *Soit L un langage de $\Sigma^\star$, on définit l'ensemble des facteurs de L, noté $Fac(L)$ par :*

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

[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.

Définition 1.6 (Quotient droit). *Le quotient droit d'un langage L par le mot w est défini par :*

$$L_{/w} = \{v \in \Sigma^\star \mid wv \in L\}$$

Le quotient droit d'un langage par un mot w est donc l'ensemble des mots de $\Sigma^\star$ qui, concaténés à w, produisent un mot de L. Cette notion généralise la notion «d'inverse» d'un mot : on pourrait tout aussi bien noter cet ensemble $w^{-1}L$.

Définition 1.7 (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 1.8 (Congruence droite associée à un langage L). *Soit L un langage et soient w_1, w_2 deux mots tels que $L_{/w_1} = L_{/w_2}$. Il est clair, par définition du quotient, que l'on a alors $\forall u, L_{/w_1u} = L_{/w_2u}$. S'en déduit une congruence droite $\mathcal{R}_\mathcal{L}$ «naturellement» associée à L et définie par :*

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

1.4.3 Morphismes

Définition 1.9 (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 fonction de comptage des occurrences.

Définition 1.10 (Code). *Un code est un morphisme injectif : $\phi(u) = \phi(v)$ entraîne $u = v$.*

Chapitre 2

Langages et expressions rationnels

Une première famille de langage 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 langage 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 imposé 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 2.1), nous étudions quelques systèmes informatiques classiques mettant ces concepts en application.

2.1 Rationalité

2.1.1 Langages rationnels

Parmi les opérations définies dans $\mathcal{P}(\Sigma^\star)$ à la section 1.4, 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 2.1 (Langages rationnels). *Soit Σ un alphabet fini. 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, L_1 et L_2 sont des langages rationnels, alors $L_1 \cup L_2$, $L_1 L_2$, et $L^\star$ sont également des langages rationnels.
Est alors rationnel tout langage construit par un nombre fini d'application de la récurrence (iii).

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

[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.

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 : $\{0,1\}^*\{111\}\{0,1\}^*$, 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.

2.1.2 Expressions rationnelles

Définition 2.2 (Expressions rationnelles). *Soit Σ un alphabet fini. Les* expressions rationnelles *(RE) sur Σ sont définies inductivement par :*
(i) εet $\emptyset$ 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 $(e_1 + e_2)$, $(e_1 e_2)$, $(e_1^\star)$ et $(e_2^\star)$ sont également des expressions rationnelles.
On appelle alors expression rationnelle toute formule construite par un nombre fini d'application 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) εdénote 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
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.

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 ($+$). Ainsi, $aa^\star + b^\star$ s'interprète-t-il comme $((a(a^\star)) + (b^\star))$.

2.1.3 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 2.3 (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 de la Table 2.1, qui expriment, (par le signe $=$), un certain nombre d'équivalences élémentaires :

$\emptyset e = e\emptyset = \emptyset$	$\varepsilon e = e\varepsilon = e$
$\emptyset^\star = \varepsilon$	$\varepsilon^\star = \varepsilon$
$e + f = f + e$	$e + \emptyset = e$
$e + e = e$	$e^\star = (e^\star)^\star$
$e(f + g) = ef + eg$	$(e + f)g = eg + fg$
$(ef)^\star e = e(fe)^\star$	
$(e + f)^\star = e^\star(e + f)^\star$	$(e + f)^\star = (e^\star + f)^\star$
$(e + f)^\star = (e^\star f^\star)^\star$	$(e + f)^\star = (e^\star f)^\star e^\star$

Tab. 2.1 – Identités rationnelles

En utilisant ces identités, il devient possible d'opérer des transformations purement syntaxiques (c'est-à-dire qui ne changent pas le langage dénoté) d'expressions rationnelles, en particulier pour les simplifier. Un exemple de réduction obtenue par application de ces expressions est le suivant :

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

La conceptualisation algorithmique d'une stratégie efficace permettant de réduire les expressions rationnelles sur la base des identités de la Table 2.1 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 des automates finis, qui sera présentée dans le chapitre suivant (à la section 3.2.2).

2.2 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
```

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 le jeu de caractère étendu IS0 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 la Table 2.2.

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`, `âmes`, `assions`, `assiez`, `assent`. On trouvera donc toutes les formes du premier groupe avec un simple[2] :

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

Guère plus difficile, le deuxième groupe : `isse`, `isses`, `î`, `îmes`, `issions`, `issiez`, `issent`. D'où le nouveau motif :

```
> grep -E '([îâ]t|[âî]mes|[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 :

[2] L'option `-E` donne accès à toutes les extensions notationnelles

L'expression	dénote	remarque
.	Σ	. vaut pour n'importe quel symbole
Répétitions		
`e*`	$e^\star$	
`e+`	$ee^\star$	
`e?`	$e + \varepsilon$	
`e{n}`	(e^n)	
`e{n,m}`	$(e^n + e^{n+1} \ldots + e^m)$	à condition que $n \leq m$
Regroupements		
`[abc]`	$(a + b + c)$	a, b, c sont des caractères
`[a-z]`	$(a + b + c \ldots z)$	utilise l'ordre des caractères ASCII
`[^a-z]`	$\Sigma \backslash \{a, b, c\}$	n'inclut pas le symbole de fin de ligne `\n`
Ancres		
`\<e`	e	e doit apparaître en début de mot, ie. précédé d'un séparateur (espace, virgule...)
`e\>`	e	e doit apparaître en fin de mot, ie. suivi d'un séparateur (espace, virgule...)
`^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
...	+	

TAB. 2.2 – Définition des motifs pour `grep`

```
> grep -E '([îâû]n?t|[îâû]mes|[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|[îâû]mes|[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 pourtant pas cette notation que nous allons utiliser, mais la notation \>, qui est une notation pour εlorsque 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|[îâû]mes|[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 apparaissent sur un radical comprenant au moins trois lettres, soit finalement (en ajoutant également \< qui spécifie un début de mot) :

```
> grep "\<[a-zéèêîôûç]{3,}([îâû]n?t|[îâû]mes|[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 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 rechercher dans un fichier. Une bibliothèque équivalente existe en C++, en java...

Attention Une confusion fréquente à éviter : les shells UNIX utilisent des notations complètement différentes pour exprimer des ensembles de noms de fichiers. Ainsi, par exemple, la commande `ls nom*` liste tous les fichiers dont le nom est préfixé par `nom` ; et pas du tout l'ensemble de tous les fichiers dont le nom appartient au langage $nom^\star$.

Chapitre 3

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 and 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).

3.1 Automates finis

3.1.1 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 3.1 (Automate fini). *Un* automate fini *(DFA) 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
– $F \subset Q$ *sont les états* finaux
– δ *est une fonction* totale *de* $(Q \times \Sigma)$ *dans* Q, *appelée* fonction de transition.

La terminologie anglaise correspondante parle de *finite-state automaton* ; comme il apparaîtra plus tard, les automates finis de cette section possèdent la propriété d'être *déterministes* : d'où l'abréviation DFA pour *Deterministic Finite-state Automaton*.

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. Si $\delta(q, a) = r$, on dit que a est *l'étiquette* de la transition (q, r). Les automates admettent une représentation graphique, comme celle de la Figure 3.1.

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 se graphe s'exprime matriciellement par :

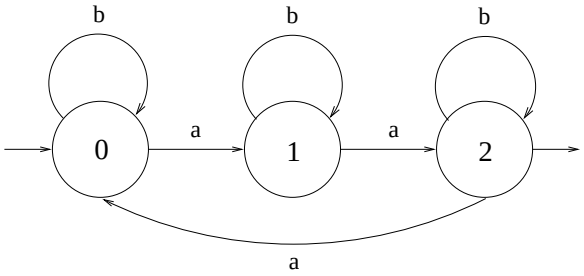


FIG. 3.1 – Un automate fini

δ	a	b
0	1	0
1	2	1
2	0	2

Un *calcul* dans A est une séquence d'états $q_1 \dots q_n$ de A, tel qu'il existe toujours au moins une transition entre deux états successifs q_i et q_{i+1}. L'*étiquette du calcul* est le mot construit par concaténation des étiquettes de chacune des transitions. Un calcul dans A est réussi si l'état d'origine est l'état initial, et si l'état terminal est 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 : 00122.

La notation $\vdash_A$ permet de formaliser cette notion. 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. Cette notation s'étend en $(q, uv) \vdash_A^\star (p, v)$ s'il existe une suite d´états $q = q_1 \dots q_n = p$ tels que $(q_1, u_1 \dots u_n v) \vdash_A (q_2, u_2 \dots u_n v) \dots \vdash_A (q_n, v)$. 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 un état 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.

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 3.2. *Un langage est* reconnaissable *s'il existe un automate fini qui le reconnaît.*

Un second exemple d'automate, très similaire au premier, est représenté à la Figure 3.2. Dans cet exemple, 0 est à la fois initial et final. A reconnaît le langage correspondant aux mots u tels que $|u|_a$ est divisible par 3 : chaque 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 :

Algorithm 1 – Reconnaissance par un DFA

// $u = u_1 \dots u_n$ est le mot à reconnaître
// $A = (Q, q_0, \delta, F)$ est le DFA
$q := q_0$
$i := 1$
while $(i \leq n)$ **do**
    $q := \delta(q, u_i)$
    $i := i + 1$
od
if $(q \in F)$ **then** *return*(**true**) **else** *return*(**false**)

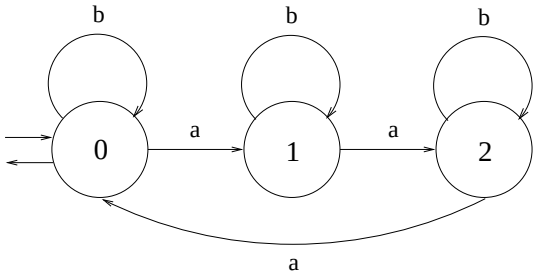


FIG. 3.2 – Un automate fini comptant les a (modulo 3)

– $\delta^\star(q, u) = \delta(q, u)$ si $|u| = 1$
– $\delta^\star(q, au) = \delta^\star(\delta(q, a), u)$
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, ie. existe et est unique. Cette nouvelle notation permet de donner une notation alternative pour le langage reconnu par un automate A :

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

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

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

Définition 3.3 (Équivalence entre automates). *Deux automates finis A_1 et A_2 sont* équivalents *si et seulement s'ils reconnaissent le même langage.*

Ainsi, par exemple, l'automate de la Figure 3.3 est-il équivalent à celui de la Figure 3.1 : tous deux reconnaissent le langage de tous les mots qui contiennent un nombre de a congru à 2 modulo 3.

Nous l'avons noté plus haut, $\delta^\star$ est définie pour tout mot de $\Sigma^\star$. Ceci implique que l'algorithme de reconnaissance (1) 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 de la Figure 3.4, 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.

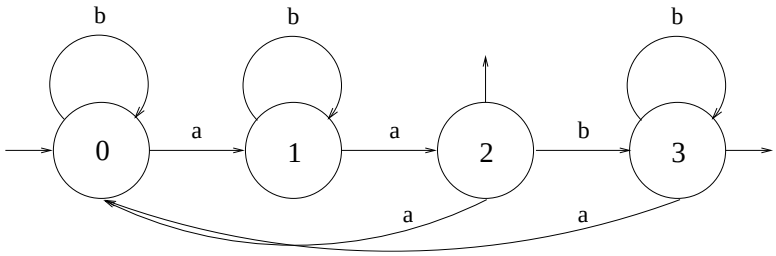


FIG. 3.3 – Un automate fini équivalent à celui de la Figure 3.1

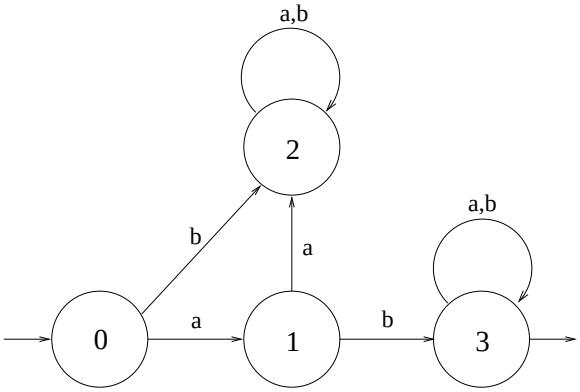


FIG. 3.4 – Un automate fini pour $ab(a+b)^\star$

3.1.2 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 3.4 (Automate fini). *Un automate fini 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*
– $F \subset Q$ *sont les états finaux*
– δ *est une fonction* partielle *de* $(Q \times \Sigma)$ *dans* Q

La différence avec la définition 3.1 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 évolue dans une configuration (q, au), telle qu'il n'existe pas de transition étiquetée par a et sortant de l'état q.

La définition (3.4) 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 de la Figure 3.5, dont le transformé est précisément l'automate de la Figure 3.4, l´état 2 jouant le rôle de puits.

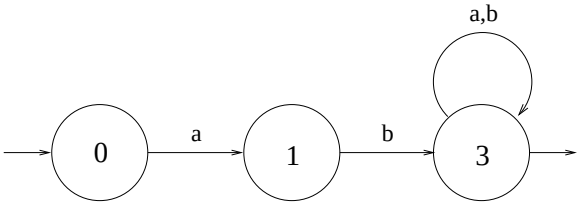


FIG. 3.5 – Un automate partiellement spécifié

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 3.4), il existe donc un automate complètement spécifié (ou automate *complet*) équivalent.

3.1.3 États utiles

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

Définition 3.5 (Accessibilité, co-accessibilité). *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 à la spécification de $L(A)$. C'est précisément ce que montre le théorème suivant.

Théorème 3.1 (Émondage). *Si $L(A) \neq \emptyset$ est un langage reconnaissable, alors il est également reconnu par un automate émondé.*

Preuve : 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'.

3.1.4 Automates non-déterministes

Dans cette section, nous augmentons le modèle d'automate défini en (3.4), 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 pas les automates déterministes.

Non-déterminisme

Définition 3.6 (Automate fini non-déterministe). *Un automate fini* non-déterministe *(NFA) 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
– *$F \subset Q$ sont les états* finaux
– *δ est une fonction (partielle) de $(\Sigma \times Q)$ dans 2^Q : l'image par δ d'un couple (q, a) est un sous-ensemble de Q.*

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 (3.4) est un cas particulier de la définition (3.6), avec pour tout (q, a), l'ensemble $\delta(q, a)$ ne contient qu'un seul élément : on parle alors d'automate déterministe.

Les notions de calcul et de calcul réussi se définissent exactement comme dans dans le cas déterministe. On définit également la fonction de transition étendue $\delta^\star$ de $Q \times \Sigma^\star$ dans 2^Q par :
– $\delta^\star(q, u) = \delta(q, u)$ si $|u| = 1$
– $\delta^\star(q, au) = \bigcup_{r \in \delta(q,a)} \delta^\star(r, u)$
Ces notions sont illustrées sur l'automate non-déterministe de la Figure 3.6 : 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 $(1, 1, 1)$, qui n'est pas un calcul réussi.

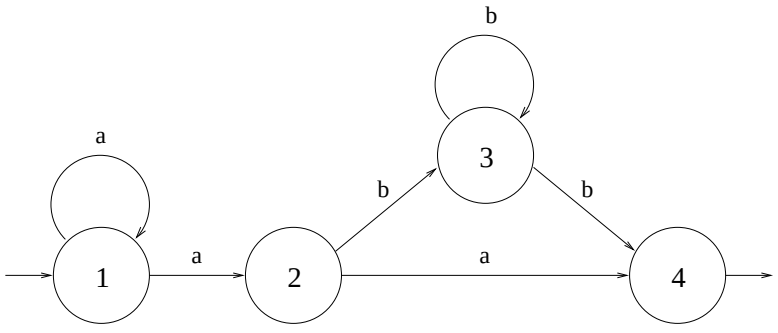


FIG. 3.6 – Un automate non-déterministe

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

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

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 q_0 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 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. Dans ce nouveau modèle, le temps de reconnaissance d'un mot n'est plus linéaire, mais proportionnel au nombre de chemins dont ce mot est l'étiquette, qui peut être exponentiel en fonction de la taille de l'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 3.2. *Pour tout NFA A défini sur Σ, il existe un DFA A' équivalent à A. Si A a n états, alors A' a au plus 2^n états.*

Preuve : on pose $A = (\Sigma, Q, q_0, F, \delta)$ et on considère A' défini par : $A' = (\Sigma, 2^Q, \{q_0\}, F', \delta')$ avec :
– $F' = \{G \subset Q, F \cap G \neq \emptyset\}$.
– $\delta'(G, a) = H$, avec $H = \cup_{q \in G} \delta(q, a)$
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 le singleton $\{q_0\}$; 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 de la Figure 3.7.

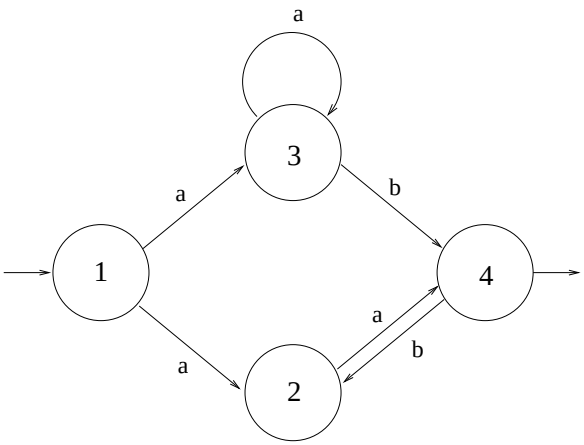


FIG. 3.7 – Un automate à déterminiser

L'automate de la Figure 3.7 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 représenté à la Figure 3.8. 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.

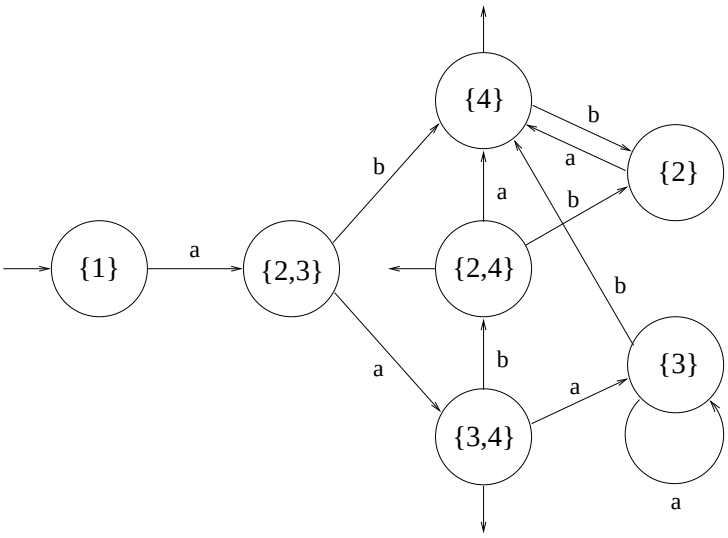


FIG. 3.8 – Le résultat de la déterminisation

Que se passerait-il si l'on ajoutait à l'automate de la Figure 3.7 une transition supplémentaire bouclant dans l'état 1 sur le symbole a ? Construisez le déterminisé de ce nouvel automate.

Démontrons maintenant le théorème 3.2 ; et pour saisir le sens de la démonstration, reportons nous à la Figure 3.7, 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 choisit 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 la Figure 3.8 : 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.

Première remarque : A' est un automate fini déterministe, puisque l'image par δ' d'un couple (H,a) est uniquement définie. Nous allons montrer que tout calcul dans A correspond à exactement un calcul dans A', soit formellement que :

$$\text{si } (q_0,u) \vdash_A^\star (p,\varepsilon) \text{ alors } \exists G, p \in G, (\{q_0\},u) \vdash_{A'}^\star (G,\varepsilon) \text{si } (\{q_0\},u) \vdash_{A'}^\star (G,\varepsilon) \text{ alors } \forall p \in G, (q_0\, u) \vdash_A^\star (G,\varepsilon)$$

Opérons une récurrence sur la longueur de u : si u est égal à εle 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 : $(\{q_0\},v) \vdash_{A'}^\star (G,\varepsilon)$, 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 $(\{q_0\},u = va) \vdash_{A'}^\star (H,\varepsilon)$, avec $q \in H$. Inversement, soit $(\{q_0\},u = va) \vdash_{A'}^\star (G,a) \vdash_{A'} (H,\varepsilon)$: pour tout p dans G il existe un calcul 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 $(\{q_0\}, u) \vdash_{A'}^{\star} (G, \varepsilon)$ 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 le NFA équivalent à un DFA A 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 réellement utiles* du déterminisé. Il suffit, pour cela, de construire de proche en proche depuis $\{q_0\}$, les états accessibles, résultant en général à des automates (complets) ayant moins que 2^n états.

Il existe toutefois des automates pour lesquels l'explosion combinatoire annoncée a lieu, comme celui qui est représenté à la Figure 3.9. Sauriez-vous expliquer d'où vient cette difficulté ? Quel est le langage reconnu par cet automate ?

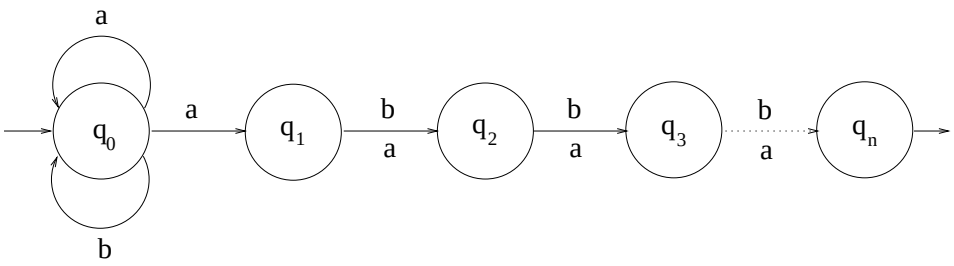


FIG. 3.9 – Un automate difficile à déterminiser

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.

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* .

Formellement, un automate non-déterministe avec transitions spontanées (en abrégé un ε-NFA) se définit comme un *NFA*, à la différence près que δ a maintenant comme domaine de définition $Q \times (\Sigma \cup \{\varepsilon\})$. Les transitions spontanées permettent d'étendre la notion de calcul : $(q, u) \vdash_A (p, v)$ si (i) $u = av$ et $p \in \delta(q, a)$ ou bien (ii) $u = v$ et $p \in \delta(q, \varepsilon)$. 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 $L(A) = \{u, (q_0, u) \vdash_A^{\star} (q, \varepsilon) \text{ avec } q \in F\}$.

La Figure 3.10 représente un exemple de ε-NFA, correspondant au langage $a^{\star}b^{\star}c^{\star}$.

Cette nouvelle extension n'ajoute rien de plus 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 de ε-fermeture d'un état q, correspondant à tous les états accessibles depuis q par une ou plusieurs transition spontanée. Formellement :

Définition 3.7. *Soit q un état de Q. On appelle ε-fermeture (en anglais closure) de q l'ensemble $\varepsilon - closure(q) = \{p, (q, \varepsilon) \vdash_A^{\star} (p, \varepsilon)\}$. Par construction, $q \in \varepsilon - closure(q)$.*

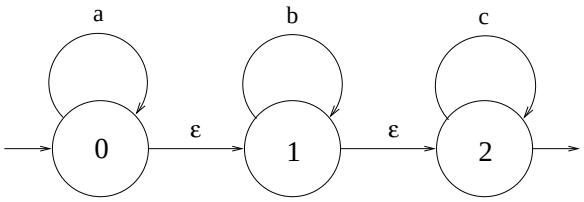


FIG. 3.10 – Un automate avec transitions spontanées

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 de la Figure 3.10 est-elle égale à $\{0, 1, 2\}$.

Théorème 3.3. *Pour tout ε-NFA A, il existe un NFA A' tel que $L(A) = L(A')$.*

Preuve. En posant $A = (\Sigma, Q, q_0, F, \delta)$, on définit A' comme suit : $A' = (\Sigma, Q, q_0, F', \delta')$ avec :
– $F' = \{q, \varepsilon - closure(q) \cap F \neq \emptyset\}$
– $\delta'(q, a) = \bigcup_{p \in \varepsilon - closure(q)} \delta(p, a)$
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 de la Figure 3.10, cet algorithme construit l'automate représenté à la Figure 3.11.

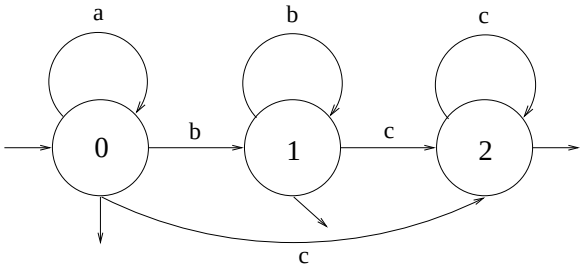


FIG. 3.11 – Un automate débarrassé de ses transitions spontanées

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.

3.2 Reconnaissables

Nous avons défini à la section 3.1 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 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 2.1.1), puis nous présentons un ensemble de résultats classiques permettant de caractériser les langages reconnaissables.

3.2.1 Opérations sur les reconnaissables

Théorème 3.4 (Clôture par complémentation). *Les langages reconnaissables sont clos par complémentation.*

Preuve : 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 $F' = Q \backslash F$. 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'.

Théorème 3.5 (Clôture par union ensembliste). *Les langages reconnaissables sont clos par union ensembliste.*

Preuve : 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 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 3.6 (Clôture par intersection ensembliste). *Les langages reconnaissables sont clos par intersection ensembliste.*

Preuve (constructive[1]) : 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 :
– $q_0 = (q_0^1, q_0^2)$
– $F = (F^1, F^2)$
– $\delta((q_1, q_2), a) = (\delta^1(q^1, a), \delta^2(q^2, a))$
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 comme : $F = (F^1, F^2)$.

Théorème 3.7 (Clôture par miroir). *Les langages reconnaissables sont clos par miroir.*

Preuve : 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.

[1] Une preuve plus directe utilise les deux résultats précédents et la loi de Morgan $L_1 \cap L_2 = \overline{(\overline{L_1} \cup \overline{L_2})}$.

Théorème 3.8 (Clôture par concaténation). *Les langages reconnaissables sont clos par concaténation*

Preuve : 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. 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 $A = (\Sigma, Q^1 \cup Q^2 \backslash \{q_0^2\}, q_0^1, F^2, \delta)$, 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 $q \in Q^2$.
– $\delta(q_F^1, a) = \delta^1(q_F, a) \cup \delta^2(q_0^2, a)$
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 3.9 (Clôture par étoile). *Les langages reconnaissables sont clos par étoile.*

Preuve : La construction de A' reconnaissant $L^\star$ à partir de A reconnaissant L est immédiate : il suffit de rajouter une transition spontanée depuis tout état final de A vers l'état initial q_0. Cette nouvelle transition permet l'itération dans A' de mots de L. Pour compléter la construction, on vérifie si εappartient à $L(A)$: si ce n'est pas le cas, alors il faudra marquer l'état initial de A 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...

3.2.2 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.

Les expressions rationnelles sont formellement définies de manière récursive à partir des «briques»

de base que sont $\emptyset$, εet les symboles de Σ. Nous commençons donc par présenter les automates finis pour les langages dénotés par ces trois expressions rationnelles à la Figure 3.12.

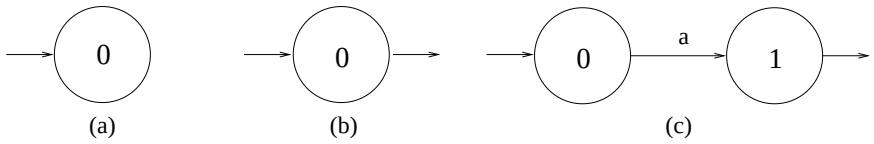


FIG. 3.12 – Machines élémentaires pour $\emptyset$, $\{\varepsilon\}$ et $\{a\}$

À partir de ces automates élémentaires, nous allons construire de manière itérative des automates pour des expressions rationnelles plus complexes. Pour cette construction, nous n'utiliserons que des automates qui ont un unique état final n'admettant aucune transition sortante, que nous noterons q_F. C'est bien le cas des automates élémentaires pour εet a ; pour $\emptyset$ il faudrait rajouter un état final distinct de (et non relié à) l'état initial. Si e_1 et e_2 dénotent les langages reconnus respectivement par A_1 et A_2, alors l'automate de la Figure 3.13 reconnaît le langage dénoté par l'expression $e_1 + e_2$.

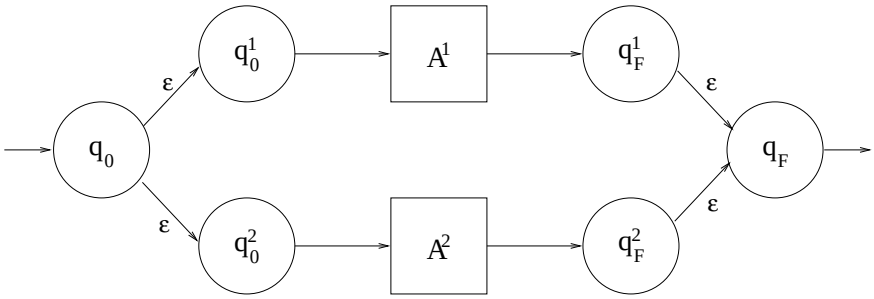


FIG. 3.13 – Machine réalisant $e_1 + e_2$

L'union correspond donc à une mise en parallèle «physique» 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. On note, par ailleurs, que la machine résultant de l'union conserve la propriété de n'avoir qu'un seul état final de degré sortant nul.

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 la Figure 3.14.

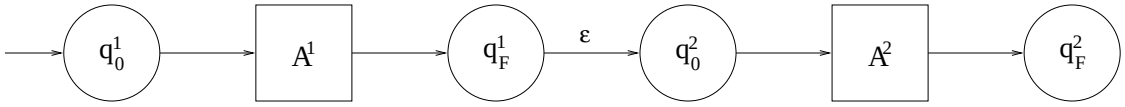


FIG. 3.14 – Machine réalisant e_1e_2.

La machine réalisant l'étoile est, comme précédemment, construite en rajoutant une possibilité de reboucler depuis l'état final vers l'état initial de la machine, ainsi qu'un arc permettant de reconnaître ε, comme représenté sur la Figure 3.15.

A 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

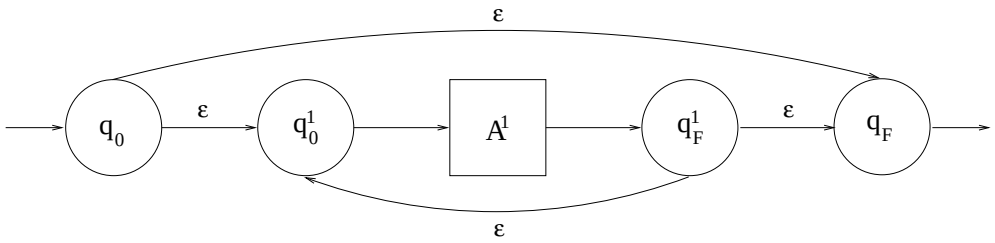


FIG. 3.15 – Machine réalisant $e_1^\star$

d'algorithme de Thompson.

En guise d'illustration de cette construction, la Figure 3.16 représente l'automate correspondant à l'expression : $e = (a + b)^\star b$.

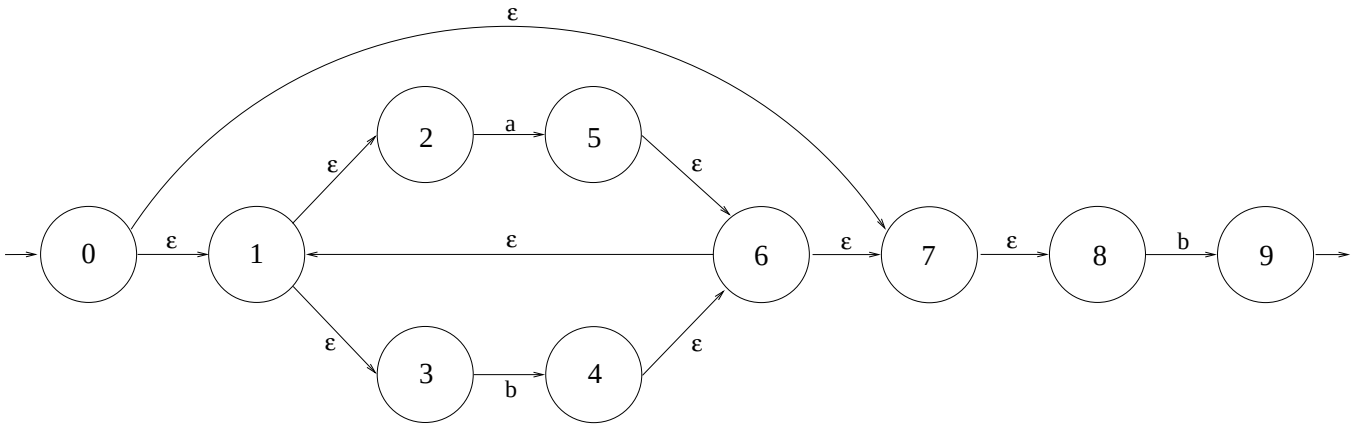


FIG. 3.16 – Machine pour $(a + b)^\star b$

Cette construction simple produit un automate qui a au plus $2n$ états pour une expression formée par n opérations rationnelles (car chaque opération rajoute 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 *Gloushkow*) ou pour construire directement un automate déterministe.

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'unique état respectivement initial et final. Ces nouveaux états sont connectés aux états initial 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_i, où q_i 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_{ii} l'étiquette de la transition de q_i vers q_i si celle ci existe ; si elle n'existe pas on a simplement $e_{ii} = \varepsilon$.

La procédure de suppression de q_i comprend alors les étapes suivantes :
- pour chaque paire d'état (q_j, q_k) avec $j \neq i, k \neq i$, tels qu'il existe une transition $q_j \to q_i$ étiquetée e_{ji} et une transition $q_i \to q_k$ étiquetée e_{ik}, ajouter la transition $q_j \to q_k$, portant l'étiquette $e_{ji}e_{ii}^{\star}e_{ik}$. Si la transition $q_j \to q_k$ existe déjà avec l'étiquette e_{jk}, alors il faut additionnellement faire : $e_{jk} = (e_{jk} + e_{ji}e_{ii}^{\star}e_{ik})$. Cette transformation doit être opérée pour chaque paire d'états (y compris pour $q_j = q_k$!) avant que q_i puisse être supprimé. Une illustration graphique de ce mécanisme est reproduit sur la Figure 3.17.
- supprimer q_i, ainsi que tous les arcs incidents

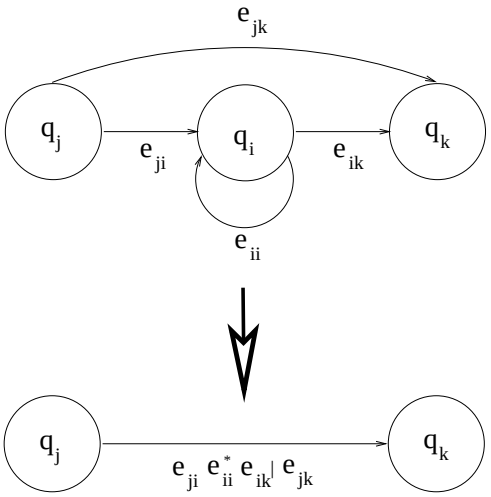


FIG. 3.17 – Illustration de BMC : élimination de l'état i

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_i avant que q_i soit supprimé contient une séquence $q_jq_iq_k$. L'étiquette de ce sous-calcul étant copiée lors de la suppression de q_i sur l'arc $q_j \to q_k$, un calcul équivalent existe dans l'automate réduit.

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 quelle est l'expression rationnelle corres-

pondant aux automates des figures 3.7 et 3.8.

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.

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 3.10 (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 de la Figure 3.2 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 3.3.2.

3.3 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 3.3.1, qui est connue sous le nom de *lemme de pompage* (en anglais *pumping lemma*). Intuitivement, cette propriété 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.

3.3.1 Lemme de pompage

Théorème 3.11 (Lemme de pompage). *Soit L un langage rationnel infini. 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) $|v| \geq 1$ (ii) $|uv| \leq k$ et (iii) pour tout $i \geq 0$, uv^iw est également un mot de L.*

Si un langage est infini, alors il contient des mots de longueur arbitrairement grande. Ce que dit ce lemme, c'est essentiellement que dès qu'un mot de L est assez long, il contient un facteur différent

de εqui peut être itéré à volonté tout en restant dans L. En d'autres termes, les mots « longs» de L sont construits par répétition d'un facteur s'insérant à l'intérieur de mots plus courts.

Preuve : Soit A un DFA de k états reconnaissant L et soit x un mot de L, de longueur supérieure ou égale à k. 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 : le lemme est aussi vérifié pour de nombreux langages non-rationnels : il n'exprime donc qu'une condition nécessaire (mais pas suffisante) de rationalité.

Ce lemme permet, par exemple, de prouver que le langage des carrés parfaits défini par $L = \{u \in \Sigma^\star, \exists v \text{ tel que } u = v^2\}$ n'est pas un langage 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 suffixe u dépend de u tout entier : reconnaître un tel langage demanderait en fait un nombre infini d'états.

3.3.2 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 1) ; cette section montre en fait que la plupart des problèmes classiques pour les langages rationnels ont des solutions algorithmiques.

Théorème 3.12. *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 $k \leq |u| < 2k$

Preuve (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 $|u| < k$.

(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 $2k$, ce qui est impossible. C'est donc que le plus petit mot de longueur au moins k a une longueur inférieure à $2k$.

Théorème 3.13. *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.

On en déduit un résultat concernant l'équivalence :

Théorème 3.14. *Soient A^1 et A^2 deux automates finis. Il existe une procédure permettant de décider si A^1 et A^2 sont équivalents.*

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 3.2.1) et de tester si le langage reconnu par cet automate est vide. Si c'est le cas, alors les deux automates sont effectivement équivalents.

3.4 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.

3.4.1 Une nouvelle caractérisation des reconnaissables

Commençons par une nouvelle définition : celle d'indistinguabilité.

Définition 3.8. *Soit L un langage de $\Sigma^\star$. Deux mots de u et v sont dits* indistinguables *dans L si pour tout w dans $\Sigma^\star$, soit uw et vw sont tous deux dans L, soit uw et vw sont tous deux dans $\overline{L}$.*

En d'autres termes, deux mots u et v sont distinguables dans L s'il existe un mot w de L tel que uw soit dans L, mais pas vw. La relation d'indistinguabilité dans L est une relation réflexive, symétrique et transitive : c'est une relation d'équivalence que nous noterons $\equiv_L$.

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 de L $x = uw$, ayant pour préfixe u, $y = vw$ est en effet un autre mot de L. $u = a$ et $v = aa$ sont par contre 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 A par :

Définition 3.9. *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 d'indistinguabilité dans A par : $u \equiv_A v$.*

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 rejoint bien la précédente, puisque tout mot w tel que $\delta^\star(q, w)$ aboutisse à 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.

Pour continuer, rappelons la notion de congruence droite, déjà introduite à la section 1.4.2 :

Définition 3.10 (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 3.15 (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'équivalences*

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

(ii) $\Rightarrow$ (iii) : Soit $\equiv$ la relation satisfaisant la propriété (*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.

(iii) $\Rightarrow$ (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, $L = \{u, \exists a \in \Sigma, i \in N \text{ tq. } u = a^{2^i}\}$ 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.

3.4.2 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 3.16. *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 appelé automate* canonique *de L.*

Preuve : soit A un automate fini déterministe reconnaissant L. $\equiv_A$ définit une relation d´équivalence satisfaisant les conditions de l'alinéa (ii) de la preuve du théorème 3.15 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'équivalences 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 préalable, nous définissons une troisième relation d'indistinguabilité, portant cette fois sur les états :

Définition 3.11. *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. δ_ν est correctement défini 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 :

$$\phi([q]_\nu) = \phi([p]_\nu) \quad \Leftrightarrow \quad \exists u, v \in \Sigma^\star, \delta^\star(q_0, u) = q, \delta^\star(q_0, u) = p, \text{ et } u \equiv_L v \tag{3.1}$$

$$\Leftrightarrow \quad \delta^\star(q_0, u) \equiv_\nu \delta^\star(q_0, u) \tag{3.2}$$

$$\Leftrightarrow \quad [q]_\nu = [p]_\nu \tag{3.3}$$

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, puis que, ces deux automates ayant les mêmes états initiaux et finaux, ils reconnaissent le même langage.

L'idée de l'algorithme de minimisation de $A = (\Sigma, Q, q_0, F, \delta)$ 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'équivalences. 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 $Q \backslash F$
- Itérer jusqu'à stabilisation :
  - pour toute paire d'état 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 reproduit à la Figure 3.18 :

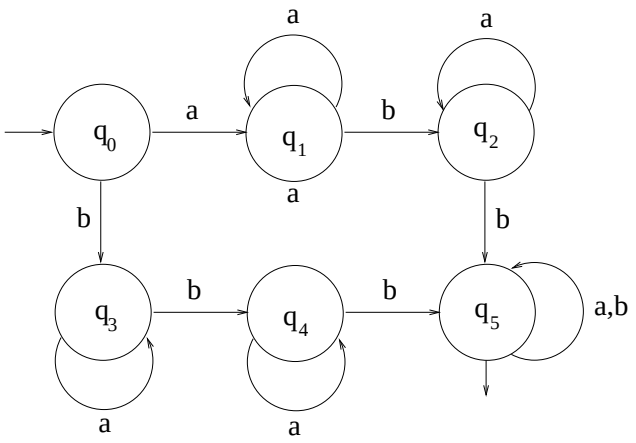


FIG. 3.18 – Un DFA à minimiser

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

– $\Pi_0 = \{\{q_0, q_1, q_2, q_3, q_4\}, \{q_5\}\}$ (car q_5 est le seul état final)
– $\Pi_1 = \{\{q_0, q_1, q_3\}, \{q_2, q_4\}, \{q_5\}\}$ (car q_2 et q_4, sur le symbole b, atteignent q_5).
– $\Pi_2 = \{\{q_0\}, \{q_1, q_3\}, \{q_2, q_4\}, \{q_5\}\}$ (car q_1 et q_3, sur le symbole b, atteignent respectivement q_2 et q_4).
– $\Pi_3 = \Pi_2$ fin de la procédure.

L'automate minimal résultant de ce calcul est reproduit à la Figure 3.19.

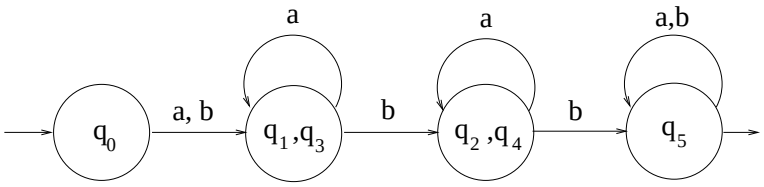


FIG. 3.19 – L'automate minimal de $(a + b)a^\star ba^\star b(a + b)^\star$

Chapitre 4

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.

4.1 Grammaires

Définition 4.1 (Grammaire). *Une* grammaire syntagmatique *G est définie par un quadruplet* (V, Σ, P, S)*, où* V*,* Σ *et* P *désignent respectivement des ensembles de* variables *(ou* non-terminaux*), de* terminaux *, et de* productions *.* V *et* Σ *sont des ensembles finis de symboles tels que* $V \cap \Sigma = \emptyset$*. Les productions sont des éléments de* $(V \cup \Sigma)^\star \times (V \cup \Sigma)^\star$*, que l'on note sous la forme* $\alpha \to \beta$*.* S *est un élément distingué de* V*, 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 *grammar* (pour eux, il n'y a pas d'ambiguïté sur le terme !).

Dans la suite, nous utiliserons les conventions suivantes pour noter les éléments de la grammaire : les variables seront notées par des symboles en majuscule ; les terminaux par des symboles minuscule ; les mots de $(V \cup \Sigma)^\star$ par des lettres de l'alphabet grec.

Illustrons ces premières définitions en examinant la grammaire définie dans la Table 4.1. 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 :	$S \to aSb$
p_2 :	$S \to ab$

Tab. 4.1 – Une grammaire pour $a^n b^n$

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,

Définition 4.2 (Dérivation Immédiate). *On définit la relation* $\Rightarrow_G$ *(lire :* dérive immédiatement*) sur l'ensemble* $(V \cup \Sigma)^\star \times (V \cup \Sigma)^\star$ *par* $\gamma\alpha\delta \Rightarrow_G \gamma\beta\delta$ *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 de la relation $\Rightarrow_G$, que l'on note $\stackrel{\star}{\Rightarrow}_G$:

Définition 4.3 (Dérivation). *On définit la relation* $\stackrel{\star}{\Rightarrow}_G$ *sur l'ensemble* $(V \cup \Sigma)^\star \times (V \cup \Sigma)^\star$ *par* $\alpha_1 \stackrel{\star}{\Rightarrow}_G \alpha_m$ *si et seulement si il existe* $\alpha_2, ..., \alpha_{m-1}$ *dans* $(V \cup \Sigma)^\star$ *tels que* $\alpha_1 \Rightarrow_G \alpha_2 \Rightarrow_G ... \Rightarrow_G \alpha_{m-1} \Rightarrow_G \alpha_m$.

Ainsi, par exemple, la dérivation suivante est une dérivation pour la grammaire de la Table 4.1 :

$$S \Rightarrow_{G_1} aSb \Rightarrow_{G_1} aaSbb \Rightarrow_{G_1} aaaSbbb \Rightarrow_{G_1} aaaaSbbbb$$

permettant de déduire que $S \stackrel{\star}{\Rightarrow}_{G_1} aaaaSbbbb$.

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

Définition 4.4 (Langage engendré par une grammaire). *On appelle langage engendré par G, noté* $L(G)$*, le sous-ensemble de* $\Sigma^\star$ *défini par* $\{w \in \Sigma^\star, S \stackrel{\star}{\Rightarrow}_G w\}$.

$L(G)$ est donc un sous-ensemble de $\Sigma^\star$, contenant précisément les mots qui se dérivent par $\stackrel{\star}{\Rightarrow}_G$ depuis S. Lorsqu'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 de la Table 4.1 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 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 8.1.2.

Comme nous l'avons fait pour les expressions rationnelles (à la section 2.1.3) et pour les automates finis (voir la section 3.1), il est en revanche possible de définir des classes d'équivalence de grammaires qui engendrent le même langage.

Définition 4.5 (Équivalence entre grammaires). *Deux grammaires* G_1 *et* G_2 *sont* équivalentes *si et seulement si elles engendrent le même langage*

[1] La terminologie est ainsi faite que lorsque l'on parle de grammaires, il est d'usage d'appeler *phrase* les séquences d'éléments terminaux, que nous appelions précédemment *mot*. Il s'ensuit parfois (et c'est fort fâcheux), que le terme de *mot* est parfois employé pour dénoter les éléments de l'alphabet $\Sigma^\star$ (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.

Les grammaires syntagmatiques décrivent donc des systèmes permettant d'engendrer, par récriture successive, 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 5.2.

4.2 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.

4.2.1 Grammaires de type 0

Définition 4.6 (Type 0). *On appelle grammaire de type 0 une grammaire syntagmatique dans laquelle la forme des production 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.

Théorème 4.1. *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 1.2.2.

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.

[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.

4.2.2 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 4.7 (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 4.2.6.

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 4.8. *On appelle grammaire* contextuelle *, ou* 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 $(\Sigma \cup V)^\star$, $\beta \neq \varepsilon$ et A est dans V.*

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

Preuve : Soit L engendré par la grammaire monotone $G = (T, N, S, P)$. Sans perte de généralité, nous supposons que les terminaux n'apparaîssent 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 $X_A \to a$. Nous allons construire une grammaire contextuelle équivalente à G et pour cela nous allons démontrer deux résultats intermédiaires.

Lemme 4.1. *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*.*

Soit, en effet, $\alpha = \alpha_1 \dots \alpha_m \to \beta = \beta_1 \dots \beta_n$ une production de G. G étant monotone, m est inférieur ou égal à n : complétons alors α avec des εde manière à écrire $\alpha = \alpha_1 \dots \alpha_n$, où les α_i sont dans $\Sigma \cup V \cup \{\varepsilon\}$. 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 :
– $\alpha_1 \alpha_2 \to \beta_1 X_1$
– $\forall i, 1 < i < n, X_{i-1} \alpha_{i+1} \to \beta_i X_i$
– $X_{n-1} \to \beta_n$
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 :

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

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 $S \overset{\star}{\Rightarrow}_G \quad v\alpha w \Rightarrow_G \quad v\beta w \overset{\star}{\Rightarrow}_G \quad u$, alors u se dérive dans G' en remplaçant l'étape de dérivation $v\alpha w \Rightarrow_G \quad v\beta w$ 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 : G est monotone) ont une longueur inférieure ou égale à 2. Le lemme suivant nous permet d'arriver à la forme désirée pour les parties gauches.

Lemme 4.2. *Si G est une grammaire monotone ne contenant que des productions de type $AB \to CD$, avec A, B, C et D dans $V \cup \{\varepsilon\}$, alors il existe une grammaire équivalente satisfaisant la propriété du théorème 4.2.*

La démonstration de ce lemme utilise le même principe que la démonstration précédente, en introduisant pour chaque production $R = AB \to CD$ le nouveau non-terminal X_{AB} et en remplaçant R par les trois productions suivantes :
– $AB \to X_{AB}B$
– $X_{AB}B \to X_{AB}D$
– $X_{AB}D \to CD$
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 4.9 (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). Une présentation plus complète des langages contextuels et des grammaires qui les reconnaissent est effectué au chapitre 13. Un représentant notable de ces langages est le langage $\{a^n b^n c^n, n \geq 1\}$, qui est engendré par la grammaire G d'axiome S et dont les productions sont représentées à la Table 4.2.

p_1 :	S	$\to aSQ$
p_2 :	S	$\to abc$
p_3 :	cQ	$\to Qc$
p_4 :	bQc	$\to bbcc$

TAB. 4.2 – Une grammaire pour $a^n b^n c^n$

La grammaire de la Table 4.2 est monotone. Dans cette grammaire, on observe par exemple les dérivations listées dans la Table 4.3.

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 :

$S \Rightarrow_G abc$ (par p_1)
$S \Rightarrow_G aSQ$ (par p_1)
$\Rightarrow_G aabcQ$ (par p_2)
$\Rightarrow_G aabQc$ (par p_3)
$\Rightarrow_G aabbcc$ (par p_4)
$S \Rightarrow_G aSQ$ (par p_1)
$\Rightarrow_G aaSQQ$ (par p_1)
$\Rightarrow_G aaabcQQ$ (par p_2)
$\Rightarrow_G aaabQcQ$ (par p_3)
$\Rightarrow_G aaabbccQ$ (par p_4)
$\Rightarrow_G aaabbcQc$ (par p_3)
$\Rightarrow_G aaabbQcc$ (par p_3)
$\Rightarrow_G aaabbbccc$ (par p_4)

TAB. 4.3 – Des dérivations pour $a^n b^n c^n$

Théorème 4.3. *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 impliquée dans la dérivation de u est au plus aussi long que u, à cause de la propriété de monotonie. Il «suffit» donc, pour décider u, de construire de proche en proche l'ensemble D de toutes 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. De nouveau, le lecteur intéressé se reportera au chapitre 13, dans lequel certains de ces algorithmes sont présentés.

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*.

4.2.3 Grammaires hors-contexte

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 4.10 (Grammaire de type 2). *On appelle grammaire de type 2 (on dit également* hors-contexte *ou* algébrique *, en abrégé* grammaire CF*) (en anglais* Context-free*) une grammaire syntagmatique dans laquelle toute production $\alpha \rightarrow \beta$ est de la forme : $A \rightarrow \beta$, avec A dans V et β dans $(V \cup \Sigma)^+$.*

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 4.11 (A-production). *On appelle A-production une production dont la partie gauche est réduit au symbole A.*

Les langages hors-contexte se définissent alors par :

Définition 4.12 (Langage hors-contexte). *On appelle langage hors-contexte (ou langage* algébrique, *en abrégé* langage CF*) un langage engendré par une grammaire hors-contexte.*

Attention : ce n'est pas parce qu'un langage est engendré par une grammaire hors-contexte qu'il est lui-même nécessairement hors-contexte ; ni parce qu'un langage est engendré par une grammaire contextuelle qu'il est lui-même contextuel. Considérez par exemple la grammaire contextuelle suivante :

p_1 :	S	$\rightarrow aSb$
p_2 :	aSb	$\rightarrow aaSbb$
p_3 :	S	$\rightarrow ab$

TAB. 4.4 – Une grammaire contextuelle pour $a^n b^n$

La grammaire (contextuelle) de la Table 4.4 engendre le langage hors-contexte $a^n b^n$. Ce langage étant un langage CF, on peut toutefois trouver une grammaire CF pour ce langage.

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 5.

Notons simplement, pour l'instant, que cette classe contient des langages non-triviaux, comme par exemple $\{a^n b^n, n \geq 1\}$, engendré par la grammaire de la Table 4.1, qui est bien une grammaire hors-contexte ; ou encore le langage des palindromes (l'écriture de cette grammaire est laissée en exercice).

4.2.4 Grammaires régulières

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 4.13 (Grammaire de type 3). *On appelle grammaire de type 3 (on dit également* régulière *, en abrégé* grammaire RG*) (en anglais* regular*) une grammaire syntagmatique dans laquelle toute production $\alpha \rightarrow \beta$ est soit de la forme : $A \rightarrow aB$, avec a dans Σ et A, B dans V, soit de la forme $A \rightarrow a$.*

Par définition, toute grammaire régulière est hors-contexte. Le trait spécifique des grammaires régulières est que chaque production récrit au moins 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 une variable. 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 :$A \rightarrow a$.

La notion de langage régulier se définit comme précédemment par :

Définition 4.14 (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.

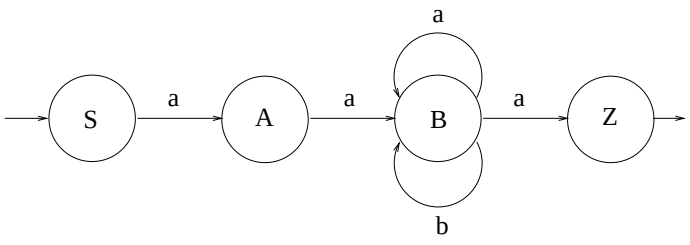
Les réguliers sont rationnels

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

Théorème 4.4. *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 4.2.6), 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 3.2) ; nous y renvoyons le lecteur.

Pour avoir l'intuition de ce théorème, considérons la grammaire engendrant le langage $aa(a \mid b)^\star a$ et l'automate reconnaissant ce langage, tous deux reproduits dans la Table 4.5.

G	A
p_1 : $S \rightarrow aA$ p_2 : $A \rightarrow aB$ $p_{3,4}$: $B \rightarrow aB \mid bB$ p_5 : $B \rightarrow a$	

Tab. 4.5 – Grammaire et automate pour $aa(a \mid b)^\star$

Dans G, la dérivation de l'entrée $aaba$ est $S \Rightarrow_G aA \Rightarrow_G aaB \Rightarrow_G aabB \Rightarrow_G aaba$, 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.

Soient $G = (V, \Sigma, P, S)$ une grammaire régulière et A l'automate dérivé de G suivant $A = (\Sigma, V \cup \{Z\}, S, \{Z\}, \delta)$, avec la fonction δ est définie selon :
– $\delta(A, a) = B \Leftrightarrow (A \rightarrow aB) \in P$
– $\delta(A, a) = Z \Leftrightarrow (A \rightarrow a) \in P$
Montrons maintenant que $L(G) = L(A)$ et pour cela, montrons par induction l´équivalence suivante : $S \Rightarrow_G^\star uB$ 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 $S \Rightarrow_G^\star avA \Rightarrow_G avbB$. 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 4.2.3) 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 3.3.1), il ne peut être reconnu par un automate fini. La distinction introduite entre langages réguliers et langages hors-contexte n'est pas de pure forme : il existe des langages CF qui ne sont pas rationnels.

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 4.15 (Grammaire de type 3). *On appelle grammaire de type 3 (ici* 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 V, 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 langage 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 de la Table 4.1.

4.2.5 Grammaires à choix finis

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

4.2.6 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 8.1.4).

Pour ce qui concerne les grammaires régulières, notons simplement que si l'on accepte des productions de type $A \to \varepsilon$, 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 4.2.4 en marquant comme états finaux de A tous les états correspondant à une variable pour laquelle il existe une règle $A \to \varepsilon$. La construction inverse en est également simplifiée : pour chaque état final A de l'automate, on rajoute la production $A \to \varepsilon$.

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 $A \to \varepsilon$; qu'elle est simplement hors-contexte sinon. De même on parlera de langage strictement hors-contexte pour un langage CF ne contenant pas εet de langage hors-contexte sinon.

4.2.7 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 (voir en particulier au chapitre 9 pour les langages CF et la chapitre 13 pour les langages CS)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 de également de décider ces langages. En particulier, le problème de la reconnaissance pour les grammaires CF sera étudié en détail dans les chapitre 6 et chapitre 7.

Chapitre 5

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 4.2.3 comme étant des grammaires syntagmatiques dont toutes les productions sont de la forme $A \rightarrow u$, 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 5.1, 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 5.2). La section 5.3 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 6.

5.1 Quelques exemples

5.1.1 La grammaire des déjeuners du dimanche

Un exemple de grammaire hors-contexte est présenté à la Table 5.1, 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 majuscule, les terminaux sont des mots du vocabulaire usuel. On utilise également dans cette grammaire le symbole | pour exprimer une alternative : $A \rightarrow u \mid v$ vaut pour les deux règles $A \rightarrow u$ et $A \rightarrow v$.

Première remarque sur la grammaire de la Table 5.1 : elle contient de nombreuses règles de type $A \rightarrow a$, qui servent simplement à introduire les symboles terminaux de la grammaire : les symboles apparaissant en partie gauche de ces production 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

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

TAB. 5.1 – La grammaire G_D des repas dominicaux

obtenus resteraient identiques à ceux de $L(G_D)$.

En utilisant la grammaire de la Table 5.1, on construit une première dérivation pour l'énoncé *Louis boude*, représentée dans la Table 5.2 :

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

TAB. 5.2 – 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 : $S \Rightarrow_{G_D} GN\ GV \Rightarrow_{G_D} GN\ V \Rightarrow_{G_D} GN\ boude \Rightarrow_{G_D} NP\ boude \Rightarrow_{G_D} Louis\ boude$. 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 : $GN \Rightarrow_{G_D} GN\ GNP \Rightarrow_{G_D} GN\ GNP\ GNP \Rightarrow_{G_D} GN\ GNP\ GNP\ GNP \ldots$.

Dernière remarque concernant G_D : cette grammaire engendre des énoncés qui sont (du point de vue du sens) ambigu. 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.

5.1.2 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.

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 Table 5.3, les non-terminaux figurent entre crochets (<>) ; 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 `|`, chaque alternative figurant ici sur une ligne distincte.

Les éléments de base de la syntaxe sont les mots et les chiffres, définis dans la Table 5.3.

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

[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`.

```
<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>
```

TAB. 5.3 – Constructions élémentaires du bash

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 de la Table 5.4. On reconnaît dans la dérivation de la Table 5.4 une dérivation

$$\begin{aligned} \texttt{<number>} &\Rightarrow_{G_B} \texttt{<number> 0} \\ &\Rightarrow_{G_B} \texttt{<number> 10} \\ &\Rightarrow_{G_B} \texttt{<number> 510} \\ &\Rightarrow_{G_B} \texttt{3510} \end{aligned}$$

TAB. 5.4 – Dérivation du nombre 3510

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 de la Table 5.5

```
<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>
```

TAB. 5.5 – 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 4.2.4).

5.2 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.

5.2.1 Dérivation gauche

Définition 5.1 (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 la Table 5.2 : 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 la précédente.

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

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

Définition 5.2 (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 5.1. *Soit G une grammaire CF d'axiome S, et u dans* $\Sigma^\star$ *:* $S \overset{\star}{\Rightarrow}_G u$ *si et seulement si* $S \overset{L}{\Rightarrow}_G u$ *(idem pour* $S \overset{R}{\Rightarrow}_G u$*).*

Preuve 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 :

$$S \Rightarrow_G^\star \quad u_1 \dots u_i A u_j \dots u_k B \gamma \Rightarrow_G \quad u_1 \dots u_i A u_j \dots u_k \beta \gamma \Rightarrow_G^\star \quad u$$

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 sont utilisés dans aucune dérivation d'un mot 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 dérivation, 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*.

5.2.2 Arbre de dérivation

Définition 5.3 (Arbre de dérivation). *Un arbre A est un* arbre de dérivation *dans G si et seulement si :*
- *tous les nœuds de A sont étiquetés par un symbole de $V \cup \Sigma$*
- *la racine est étiquetée par S*
- *si un nœud n n'est pas une feuille et porte l'étiquette X, alors $X \in V$*
- *si $n_1, n_2 ... n_k$ sont les fils de n dans A, d'étiquettes respectives $X_1 ... X_k$, alors $X \to X_1 ... X_k$ 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 une 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 5.1. 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.

[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]Pas un ensemble au sens traditionnel : le nombre d'applications de chaque production importe.

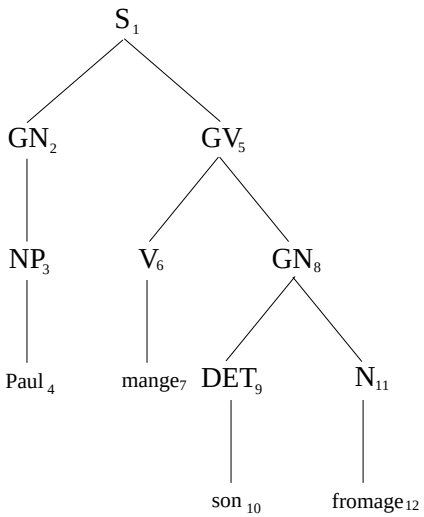


FIG. 5.1 – L'arbre de dérivation de *Paul mange son fromage*

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, 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 Table 5.6.

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

TAB. 5.6 – Une grammaire pour les sommes

Pour cette grammaire, l'expression 3 + 5 + 1 correspond à deux arbres d'analyse, représentés à la Figure 5.2.

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.

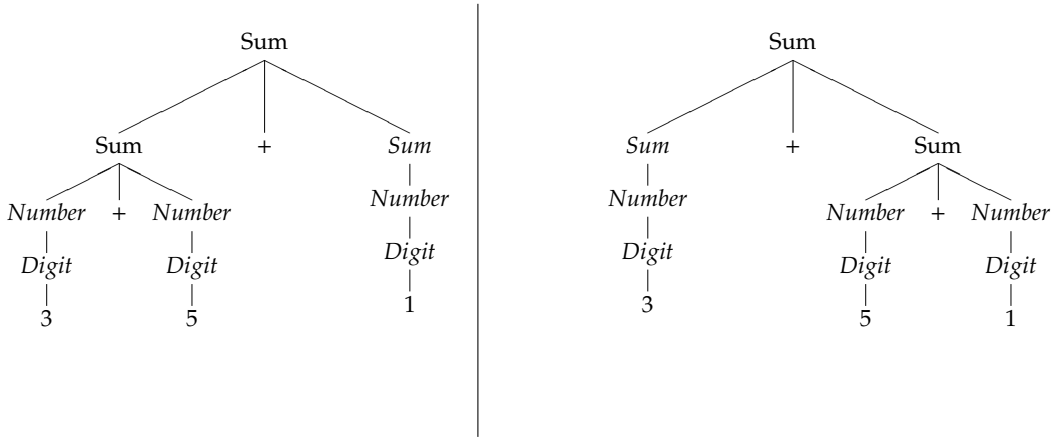


FIG. 5.2 – Deux arbres de dérivation d'un même calcul

5.2.3 Ambiguïté

Définition 5.4 (Ambiguïté). *Une grammaire est ambiguë s'il existe un mot admettant plusieurs dérivations gauches différentes dans la grammaire.*

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

La grammaire de la Table 5.7 est un exemple de grammaire ambiguë.

$$S \to ASB \mid AB$$
$$A \to aA \mid a$$
$$B \to bB \mid b$$

TAB. 5.7 – Une grammaire ambiguë

Dans la grammaire de la Table 5.7, 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 donc une propriété des grammaires.

Cette propriété contamine pourtant parfois même les langages :

Définition 5.5 (Ambiguïté (d'un langage)). *Un langage hors-contexte est (intrinsèquement) ambigu si toutes les grammaires 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. Des techniques de parsage adaptées à la gestion et à la représentation compacte des ambiguïtés seront présentées dans le chapitre consacré à la tabulation 12

5.2.4 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 5.6 (Équivalence). *Deux grammaires G_1 et G_2 sont* équivalentes *si et seulement si elles engendrent le même langage. 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*. A 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.

5.3 Les langages hors-contexte

5.3.1 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 pour le langage $a^n b^n$ (présentée page 46) et une dérivation de *aaabbb* dans cette grammaire, la lignée du second a de *aaabbb* correspond à la séquence $(S \rightarrow aSb, 2)$, $(S \rightarrow aSb, 1)$.

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 $(S \rightarrow aSb, 2)$, $(S \rightarrow aSb, 2)$, $(S \rightarrow ab, 1)$. 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 a, qui, étant non-original, contient deux fois le même ancêtre (A, i). La dérivation complète de s pourra donc s'écrire :

$$S \Rightarrow_G^\star \quad uAy \Rightarrow_G^\star \quad uvAxy \Rightarrow_G^\star \quad uvwxy$$

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 5.2 (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.*

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 lemme précédent. 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. Les deux conditions supplémentaires dérivent pour l'une ($vx \neq \varepsilon$) de la possibilité de choisir G tel qu'aucun terminal autre que S ne dérive ε(voir chapitre 8) : en considérant une dérivation minimale, on s'assure ainsi qu'au moins un terminal est produit durant la dérivation $A \stackrel{\star}{\Rightarrow} \quad vAx$. La seconde condition, ($vwx < k$), dérive de la possibilité de choisir pour vwx un facteur original et donc de taille strictement inférieure à n.

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 de ce langage. Décomposons z en $uvwxy$, et notons $z_n = uv^nwx^ny$. 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 t_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.

5.3.2 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 3.2.1.

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

Théorème 5.3 (Clôture). *Les langages hors-contextes sont clos pour les opérations rationnelles.*

Pour les trois opérations, une construction simple permet d'établir ce résultat. Si, en effet, G_1 et G_2 sont définies par : $G_1 = (V_1, \Sigma_1, S_1, P_1)$ et $G_2 = (V_2, \Sigma_2, S_2, P_2)$, on vérifie simplement que :

– $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, S, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\})$ engendre exactement $L_1 \cup L_2$.
– $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, S, P_1 \cup P_2 \cup \{S \to S_1 S_2\})$ engendre exactement $L_1 L_2$.
– $G = (V_1 \cup \{S\}, \Sigma_1, S, P_1 \cup \{S \to SS_1, S \to \varepsilon\})$ 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 leur 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 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 (mais à la section 9.3)

Théorème 5.4. *L'intersection d'un langage régulier et d'un langage hors-contexte est un langage hors-contexte.*

5.3.3 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 6.

Nous commençons par un résultat positif, qui s'énonce comme :

Théorème 5.5. *Il existe un algorithme permettant de déterminer si une grammaire hors-contexte engendre un langage vide.*

La preuve exploite en fait un argument analogue à celui utilisé dans notre démonstration du lemme de pompage (section 5.3.1) : 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 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 5.6. *Il existe un algorithme permettant de déterminer si une grammaire hors-contexte engendre un langage infini.*

L'idée de la démonstration repose sur l'observation suivante : après élimination des productions inutiles, des productions epsilon et des cycles (voir la section 8.1), 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 :

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

– 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 G_1 est inclus dans le langage engendré une grammaire CF G_2 ;
– 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.

Les preuves de ces assertions, ainsi que des résultats complémentaires concernant l'indécidabilité de problèmes algorithmiques pour les grammaires et langages CF sont présentés au chapitre 11. 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 6

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-contextes étant une sous classes 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 6.1, nous présentons l'espace de recherche du parsage ; nous présentons ensuite dans les sections 6.2 et 6.3 les stratégies ascendantes et descendantes, ainsi que les problèmes que pose la mise en œuvre de telles stratégies. Cette étude des analyseurs se poursuit au chapitre 7, où nous présentons des analyseurs déterministes, utilisables pour certains types de grammaires ; ainsi qu'au chapitre 8, où nous étudions des techniques de normalisation des grammaires, visant à simplifier l'analyse, où à se prémunir contre des configurations indésirables. L'approfondissement de ces techniques fait également l'objet des chapitres9, consacré plus particulièrement au parsage des grammaires de langages informatiques et au chapitre 12, consacré à l'analyse de langages «naturels».

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

6.1 Graphe de recherche

Un point de vue général sur la question de la reconnaissance est donné par la considération suivante. $\Rightarrow_G$ définit une relation binaire sur les séquences de $(V \cup \Sigma)^\star$; comme toute relation binaire, cette relation se représente par un graphe dont les sommets sont les séquences de $(V \cup \Sigma)^\star$

et dans lequel un arc de α vers β indique que $\alpha \Rightarrow_G \beta$. On appelle ce graphe *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 noeud 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 approches, ascendantes et descendantes, sont explicitées dans les sections qui suivent, dans lesquelles on essayera également de suggérer que ce problème admet une implantation algorithmique polynomiale : dans tous les cas, c'est-à-dire que la reconnaissance de u demande un nombre d'étapes borné par $k\,|u|^{\,p}$.

Le second problème intéressant est celui de l'analyse, c'est-à-dire de la construction de tous les arbres de dérivations 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 demandera nécessairement un temps de traitement exponentiel.

6.2 Reconnaissance ascendante

Supposons qu'étant donnée la grammaire des repas dominicaux (cf. la page 58), 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) ?
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] En renversant l'orientation des arcs.
[2] En fait, seulement ceux qui correspondent à une dérivation gauche.

1. on remplace *le* par la partie gauche de la règle *le* → *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 2.

La procédure détaillée dans l'algorithme 2 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 2 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 *burlp* 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 6.1).

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

Algorithm 2 – Parsage ascendant en profondeur d'abord

```
// la fonction principale est bulrp : bottom-up left-right parsing
// α contient le proto-mot courant
bulrp(α)
begin
  if (α = S) then return(true) fi
  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) fi
          fi
      od
  od
  return(false)
end
```

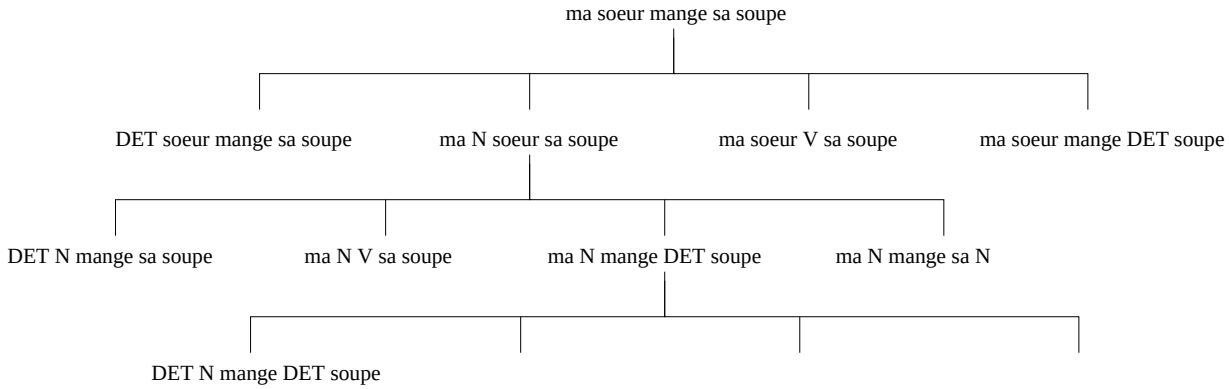


FIG. 6.1 – Recherche ascendante

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 7.2.

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. Ces techniques sont en particulier présentées dans le chapitre 12.

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 $X \rightarrow X$, ou par des cycles de production de type $X \rightarrow Y$, $Y \rightarrow X$, 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 8.1), 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.

6.3 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 3.

Algorithm 3 – Parsage descendant en profondeur d'abord

// la fonction principale est *tdlrp* : top-down left-right parsing
// α est le proto-mot courant, u l'entrée à reconnaître
$tdlrp(\alpha, u)$
begin
  if $(\alpha = u)$ **then** *return*(**true**) **fi**
  $\alpha = u_1 \dots u_k A\gamma$
  while $(\exists A \rightarrow \beta)$ **do**
    $(\beta = u_{k+1}...u_{k+l}\delta)$ avec $\delta = \varepsilon$ ou $\delta = A \dots$
    if $(tdlrp(u_1 \dots u_{k+l}\delta\gamma)$ = **true**) **then** *return*(**true**) **fi**
  od
  return(**false**)
end

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 dominicale de la Table 5.1 : 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 7.1.

Dans l'optique d'une stratégie de reconnaissance descendante, notre grammaire possède un vice de forme manifeste, à savoir la production $GN \rightarrow GN \;\; GNP$, 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 8 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.

6.4 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 document 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 7.
– 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. Ces algorithmes seront présentés dans le chapitre consacré au parsage tabulaire 12

Quel que soit le contexte, les algorithmes de parsage bénéficient toujours d'une étape de pré-traitement et de normalisation de la grammaire, visant en particulier à éviter les configurations problématiques (cycles et productions *epsilon* pour les analyseurs ascendants ; récursions gauches pour les analyseurs descendants). Ces pré-traitement sont présentés dans le chapitre 8.

Chapitre 7

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 6, 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 6.1). 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'il conduisent à une complexité d'analyse *linéaire* par rapport à la longueur de l'entrée.

La section 7.1 présente la mise en application de ce programme pour les stratégies d'analyse descendantes, conduisant à la famille d'analyseurs prédictifs LL. La section 7.2 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.

7.1 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.

7.1.1 Une intuition simple

Comme expliqué à la section 6.3, 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.
Un première manière de simplifier la tâche des analyseurs consiste à éviter d'utiliser dans la grammaire les productions de type (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 (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 de grammaire présenté dans la Table 7.1. Cette grammaire présente la particularité de ne contenir que des productions de type (i) ; de plus, les coins gauches des productions associées à ce terminal sont toutes différentes.

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

Tab. 7.1 – 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 Table 7.1, 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 G de la Table 7.2 ; la Table 7.3 décrit pas-à-pas les étapes de l'analyse de *aacddcbb* par cette grammaire.

$$S \to aSb$$
$$S \to cC$$
$$C \to dC$$
$$C \to c$$

TAB. 7.2 – 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*	

TAB. 7.3 – Étapes de l'analyse de aacddcbb
A 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.

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 de la Table 7.2, cette matrice est reproduite à la Table 7.4.

	a	*b*	*c*	*d*
S	aSb		cC	
C			c	dC

TAB. 7.4 – Table d'analyse prédictive
Si S est prédit et a le premier terminal non apparié, alors récrire S par aSb. Les cases vides sont des situations d'échec.

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.

[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 4.2.4 aboutit en fait à un automate *déterministe*.

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

Définition 7.1 (grammaires SLL(1)). *Une grammaire $G = (V, \Sigma, S, P)$ est une grammaire SLL(1) si et seulement si :*
– *(i)* $\forall (A \to \alpha) \in P, \exists a \in \Sigma, \alpha = a\beta$
– *(ii)* $\forall A \in V, (A \to a_1\alpha_1) \in P$ *et* $(A \to a_2\alpha_2) \in P \Rightarrow a_1 \neq a_2$

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 (*Left*) 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 5.2.3) 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 8.2.2). 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.

7.1.2 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 Table 7.5, qui engendre des formules arithmétiques.

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

Tab. 7.5 – Une grammaire pour les expressions arithmétiques

Le non-terminal F de la grammaire de la Table 7.5, 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 :

$$F \Rightarrow_A F * T \stackrel{\star}{\Rightarrow}_A F * T * T * \ldots * T \Rightarrow_A T * T \ldots T$$

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 (α. 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 $T \rightarrow (S)$; sinon si c'est un chiffre, il faut choisir $T \rightarrow D$.

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.

7.1.3 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 7.2 (FIRST). *Soit $G = (V, \Sigma, S, P)$ une grammaire CF et A un élément de V. On appelle $FIRST(A)$ le sous-ensemble de Σ défini par :*

$$FIRST(A) = \{a \in \Sigma, \exists \alpha \in (V \cup \Sigma), A \Rightarrow_G^\star a\alpha\}$$

Ainsi, dans la grammaire de la Table 7.5, on a :

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

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

$$FIRST(A) = \cup_{X, A \rightarrow X\alpha \in P} FIRST(X)$$

Cette approche se heurte toutefois à deux difficultés :

- les productions récursives à gauche, qui sont du type du type $A \rightarrow A\alpha$; ces productions sont hautement nocives pour les analyseurs descendants (voir la section 6.3) et il faudra dans tous les cas s'en débarrasser. Des techniques idoines pour ce faire sont présentées à la section 8.2.2 ; 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 $A \rightarrow X\alpha$ pour lesquelles le non-terminal X est tel que $X \Rightarrow_G^\star \varepsilon$: 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 $A \rightarrow \varepsilon$, et considérons l'état d'un analyseur descendant tentant de faire des prédictions à partir d'un proto-mot de la forme $uA\alpha$. A pouvant ne dériver aucun symbole, il paraît difficile de contrôler les applications erronées de la production $A \rightarrow \varepsilon$ en

regardant simplement $FIRST(A)$ et le symbole à apparier. Notons qu'il en irait de même si l'on avait $A \Rightarrow^{\star}_{G} \varepsilon$. 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 7.3 (NULL). *Soit $G = (V, \Sigma, S, P)$ une grammaire CF. On appelle NULL le sous-ensemble de V défini par :*

$$NULL = \{A \in V, A \Rightarrow^{\star}_{G} \varepsilon\}$$

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*, 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 $|V|$). Illustrons son fonctionnement sur la grammaire de la Table 7.6.

$$\begin{array}{l} S \to c \mid ABS \\ A \to B \mid a \\ B \to b \mid \varepsilon \end{array}$$

Tab. 7.6 – Une grammaire (ambiguë) pour $(a \mid b)^{\star}c$

Un premier examen de l'ensemble des productions conduit à insérer B dans *NULL* (à cause de $B \to \varepsilon$) ; la seconde itération conduit à ajouter A, puisque $A \to B$. Une troisième et dernière itération n'ajoute aucun autre élément dans *NULL* : en particulier S ne dérive pas εpuisque 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 4.

Illustrons le fonctionnement de cet algorithme sur la grammaire de la Table 7.6 : 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 :

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

FOLLOW *FOLLOW* est nécessaire pour contrôler par anticipation la validité de prédictions de la forme $A \to \varepsilon$, ou, plus généralement $A \Rightarrow^{\star}_{G} \varepsilon$: pour valider un tel choix, il faut en effet connaître

Algorithm 4 – Calcul de FIRST

```
// initialisation
foreach A ∈ V do
   FIRST^0(A) := ∅
od
k := 0
cont := true
while (cont = true) do
        k := k + 1
        foreach A ∈ V do
           FIRST^k(A) := FIRST^{k−1}(A)
        od
        foreach (A → X_1 ... X_k) ∈ P do
           j := 0
           do
              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)
           while (X_j ∈ NULL)
        od
        // Vérifie si un des FIRST a changé ; sinon stop
        cont := false
        foreach A ∈ V do
           if (FIRST^k(A) ≠ FIRST^{k−1}(A)) then cont := true fi
        od
end
```

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

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

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 εcomplique un peu le calcul, et requiert d'avoir au préalable calculé $NULL$ et $FIRST$. Ce calcul se formalise par l'algorithme 5.

Illustrons, de nouveau, le fonctionnement de cette procédure sur la grammaire de la Table 7.6. 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

Algorithm 5 – Calcul de FOLLOW

```
// initialisation
foreach A ∈ V do
   FOLLOW^0(A) := ∅
od
foreach (A → X_1 ... X_n) ∈ P do
   for j = 1 to n − 1 do
      if X_j ∈ V 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})
                     fi
                  od
      fi
   od
od
k := 0
cont := true
while (cont = true) do
      k := k + 1
      foreach A ∈ V do
         FOLLOW^k(A) := FOLLOW^{k−1}(A)
      od
      foreach (A → X_1 ... X_n) ∈ P do
         j := n + 1
         do
            j := j − 1
            FOLLOW^k(X_j) := FOLLOW^k(X_j) ∪ FOLLOW^k(A)
         while (X_j ∈ NULL)
      od
      // Vérifie si un des FOLLOW a changé ; sinon stop
      cont := false
      foreach A ∈ V do
         if (FOLLOW^k(A) ≠ FOLLOW^{k−1}(A)) then cont := true fi
      od
od
```

$FIRST(B)$, 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.

7.1.4 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 $V \times \Sigma$) 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 $(V \cup \Sigma)^\star$ de la manière suivante[2].

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

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 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 $FIRST(\alpha)$, 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 $NULL$ ($\alpha \in NULL^\star$). Ceci conduit à la seconde règle de remplissage de M :

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

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

Considérons alors une dernière fois la grammaire de la Table 7.6. 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 de la Table 7.7.

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

TAB. 7.7 – Table d'analyse prédictive pour la grammaire de la Table 7.6

L'examen de la Table 7.7 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 $M(S, b)$ de la Table 7.7, 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 $A \to B$. A 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 la Table 7.7 ne permet pas de mettre en œuvre une analyse déterministe. Ceci est dû à l'ambiguïté de la grammaire de la Table 7.6, 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).

7.1.5 Analyseurs LL(1)

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

Définition 7.5 (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 $(V \cup \Sigma)^\star$ pour chaque valeur (A, a) de $V \times \Sigma$.*

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 l'algorithme suivant (on suppose que la construction de M est une donnée de l'algorithme) :

Algorithm 6 – Analyseur pour grammaire LL(1)

// le mot en entrée est : $u = u_1 \dots u_n$
initialisation
$matched := \varepsilon$
$tomatch := u_1$
$left := S$
$i := 1$
while $(left \neq \varepsilon \ \wedge \ i \leq |u|\ )$ **do**
    $left = X\beta$
    if $X \in V$ **then**
        do
            if *undefined*$(M(X, tomatch))$ **then** *return*(**false**) **fi**
            $\alpha := M(X, tomatch)$
            // Remplace A par α
            $left := \alpha\beta$
        od
    else
        // le symbole de tête est terminal : on apparie simplement
        if $(X \neq u_i)$ **then** *return*(**false**) **fi**
    fi
    $matched := matched \ . \ u_i$
    $tomatch := u_{i+1}$
    $i := i + 1$
od
if $(left = \varepsilon \ \wedge \ tomatch = \varepsilon)$ **then** *return*(**true**) **else** *return*(**false**) **fi**

7.1.6 LL1-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 8.2.2. Une seconde transformation bien utile consiste à *factoriser* à gauche la grammaire.

Pour mesurer l'utilité de cette démarche, considérons la grammaire reproduite à la Table 7.8.

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 une nouvelle grammaire, présentée dans la Table 7.9.

$$
\begin{array}{l}
S \rightarrow if\ B\ then\ S \\
S \rightarrow if\ B\ then\ S\ else\ S \\
\ldots
\end{array}
$$

TAB. 7.8 – Une grammaire non-LL(1) pour *if-then-else*

$$
\begin{array}{l}
S \rightarrow if\ B\ then\ S\ S' \\
S' \rightarrow else\ S \mid \varepsilon \\
\ldots
\end{array}
$$

TAB. 7.9 – Une grammaire factorisée à gauche pour *if-then-else*

Le mot clé *if* sélectionne maintenant une production unique ; c'est aussi vrai pour *else*. Le fragment décrit dans la Table 7.9 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 7.1 (Factorisation gauche). *Soit $G = (V, \Sigma, S, P)$ une grammaire hors-contexte, alors il existe une grammaire équivalente qui est telle que si $A \rightarrow X_1 \ldots X_k$ et $A \rightarrow Y_1 \ldots Y_l$ sont deux A-productions de G', alors $X_1 \neq Y_1$.*

Preuve : 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 $A \rightarrow X\alpha_i$ par l'ensemble $\{A \rightarrow XA' , A' \rightarrow \alpha_i\}$, 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.

7.1.7 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 du 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.

7.1.8 Un exemple complet commenté

Nous détaillons dans cette section les constructions d'un analyseur LL pour la grammaire de la Table 7.5, qui est rappelée dans la Table 7.10, sous une forme légèrement modifiée. L'axiome est la variable Z.

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

TAB. 7.10 – Une grammaire (simplifiée) pour les expressions arithmétiques

La première étape de la construction consiste à se débarrasser des productions récursives à gauches, puis à factoriser à gauche la grammaire résultante. Au terme de ces deux étapes, on aboutit à la grammaire transformée 7.11, qui comprend deux nouvelles variables récursives à droite, S' et F', et dont nous allons montrer qu'elle est effectivement LL(1).

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 la Table 7.12.

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

TAB. 7.11 – Une grammaire LL pour les expressions arithmétiques

	FIRST						*FOLLOW*		
It.	1	2	3	4	5	It.	0	1	2
Z				(	(12	Z			
S			(	(12	(12	S	#)	#)	#)
S'	+	+	+	+	+	S'		#)	#)
T		(	(12	(12	(12	F	+	+#)	+#)
T'	*	*	*	*	*	F'		+#)	+#)
F	(	(12	(12	(12	(12	T	*	*+#)	*+#)
D	12	12	12	12	12	D		*+#)	*+#)

TAB. 7.12 – Calcul de *FIRST* et *FOLLOW*

Il est alors possible de déterminer la table d'analyse prédictive, représentée dans la Table 7.13. Cette table est sans conflit, puisque chaque case contient au plus une production.

	#	+	*	(	)	1	2
Z				$Z \rightarrow S\#$		$Z \rightarrow S\#$	$Z \rightarrow S\#$
S				$S \rightarrow FS'$		$S \rightarrow FS'$	$S \rightarrow FS'$
S'	$S' \rightarrow \varepsilon$	$S' \rightarrow +FS'$			$S' \rightarrow \varepsilon$		
T				$T \rightarrow TF'$		$T \rightarrow TF'$	$T \rightarrow TF'$
T'	$F' \rightarrow \varepsilon$	$F' \rightarrow \varepsilon$	$F' \rightarrow *F'T$	$F' \rightarrow \varepsilon$			
F				$T \rightarrow (S)$		$T \rightarrow D$	$T \rightarrow D$
D					$D \rightarrow 1$	$D \rightarrow 2$	

TAB. 7.13 – Table d'analyse prédictive pour la grammaire de l'arithmétique

Cette table permet effectivement d'analyser déterministiquement des expressions arithmétiques simples, comme le montre la trace d'exécution de la Table 7.14.

7.2 Analyseurs LR

7.2.1 Concepts

Comme évoqué à la section 6.2, 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 $A \rightarrow \alpha$ 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*.

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

Tab. 7.14 – Trace de l'analyse déterministe de $2 * (1 + 2)$

(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 2, 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.

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 de la Table 7.15.

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

Tab. 7.15 – Une grammaire pour $a^\star bb^\star a$

Soit alors $u = abba$; une analyse ascendante de ce mot est reproduite à la Table 7.16.

En considérant les réductions opérées, on obtient la dérivation : $S \Rightarrow_G AB \Rightarrow_G AbB \Rightarrow_G Aba \Rightarrow_G aAba \Rightarrow_G abba$, 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 de la Table 7.16

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

TAB. 7.16 – Analyse ascendante de $u = abba$

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 la Table 7.16. Ce processus contient quelques 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.

7.2.2 Analyseurs LR(0)

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

Définition 7.6. *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 :

$$L_{A\to\alpha} = \{\gamma = \beta\alpha \in (\Sigma \cup V)^\star \ tq.\ \exists v \in \Sigma^\star, S \Rightarrow^\star_G \ \beta A v \Rightarrow_G \ \beta\alpha v\}$$

En d'autres termes, tout mot γ de $L_{A\to\alpha}$ contient un suffixe α et est tel qu'il existe une réduction de 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 de la Table 7.15 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 de la Figure 7.1, 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 de la Figure 7.1 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 7.

L'implantation décrite dans l'algorithme 7 est un peu naïve : 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 implantation plus efficace consiste à mémoriser, pour chaque symbole de la pile, l'état q atteint dans A. A 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.

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

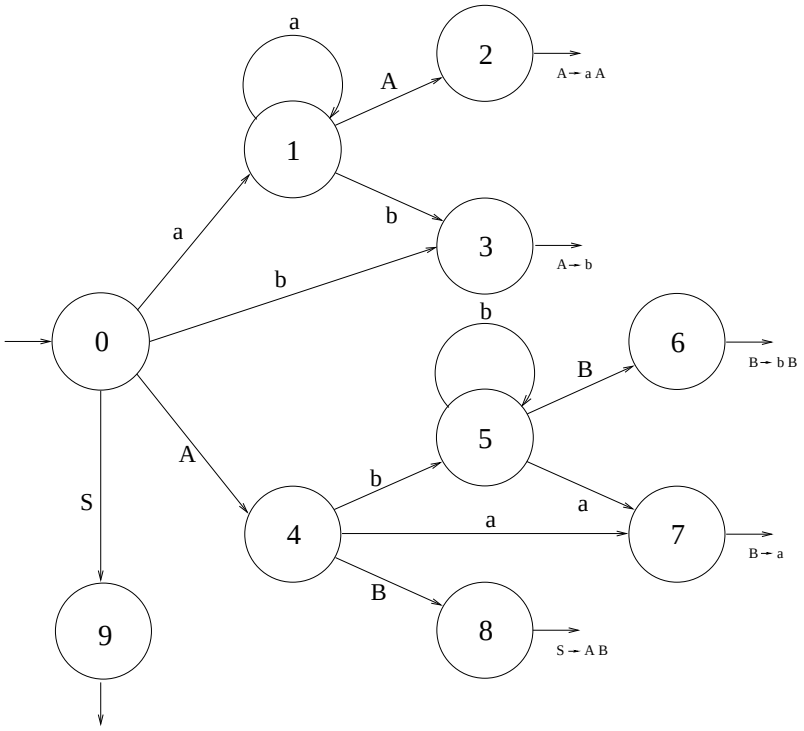


FIG. 7.1 – L'automate des réductions licites

Algorithm 7 – Reconnaissance ascendante guidée par un automate $A = (\Sigma \cup V, Q, q_0, F, \delta)$. Version 1

// initialisation
$\beta := \varepsilon$ // Initialisation de la pile
$q := q_0$ // Se positionner dans l'état initial
$i := j := 0$
while ($i \leq |u|$) **do**
    while ($j \leq |\beta|$) **do**
        $j := j + 1$
        $q := \delta(q, \beta_j)$
    od
    while ($q \notin F$) **do**
        $\beta := \beta u_i$ // Empilage de u_i
        if ($\delta(q, u_i)$ existe) **then** $q := \delta(q, u_i)$ **else** *return*(**false**) **fi**
        $i := i + 1$
    od
    // Réduction de $p : A \rightarrow \gamma$
    $\beta := \beta\gamma^{-1}A$
    $j := 0$
    $q := q_0$
od
if ($\beta = S \ \wedge \ q \in F$) **then** *return*(**true**) **else** *return*(**false**) **fi**

Algorithm 8 – Reconnaissance ascendante guidée par un automate $A = (\Sigma \cup V, Q, q_0, F, \delta)$. Version 2

```
// initialisations
β := (ε, q0) // Initialisation de la pile
q := q0 // Se positionner dans l'état initial
i := 0
while (i ≤ |u| ) do
        while (q ∉ F) do
                if (δ(q, ui) existe ) then do
                                        q := δ(q, ui) // Progression dans A
                                        push(β, (ui, q)) // Empilage de (ui, q)
                                        i := i + 1
                                od
                        else return(false)
                fi
        od
        // On atteint un état final : réduction de p : A → δ
        j := 0
        // Dépilage de δ
        while (j <|δ| ) do
                pop(β)
                j := j + 1
        od
        // (x, q) est sur le sommet de la pile : repositionnement
        push(β, (A, δ(q, A)))
        q := δ(q, A)
od
if (β = (S, q) ∧ q ∈ F) then return(true) else return(false) fi
```

l'automate de la Figure 7.1 à la Table 7.17. Cette table résume les différentes actions à effectuer en fonction de l'état courant de l'automate.

La Table 7.17 contient trois types d'entrées :
- une entrée de type (s, i) en colonne x signifie : effectuer un décalage, lire le symbole en tête de pile, si c'est un x transiter dans l'état i. *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 ;
- une entrée de type (g, j) en colonne X signifie : lire le symbole en tête de pile, si c'est un X transiter dans l'état j.
- une entrée de type $(r, A \to \alpha)$ en colonne $*$ signifie : réduire la pile selon $A \to \alpha$.

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, 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 $u\#$;

	A	B	a	b	*
0	g,4		s,1	s,3	
1	g,2		s,1	s,3	
2					r, $A \rightarrow aA$
3					r, $A \rightarrow b$
4		g,8	s,7	s,5	
5		g,6	s,7	s,5	
6					r, $B \rightarrow bB$
7					r, $B \rightarrow a$
8					r, $S \rightarrow AB$
	A	B	a	b	

TAB. 7.17 – Une table d'analyse LR(0)

– l'état (final) correspondant à la réduction $Z \rightarrow S\#$ 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 de la Table 7.17, on saura analyser la grammaire de la Table 7.15 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 7.2.3. 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 $S \rightarrow \alpha$.

Supposons, pour l'instant, qu'il n'existe qu'une seule S-production : $S \rightarrow X_1 \dots X_k$: 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 $S \rightarrow X_1 \dots X_k$, 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 7.2.

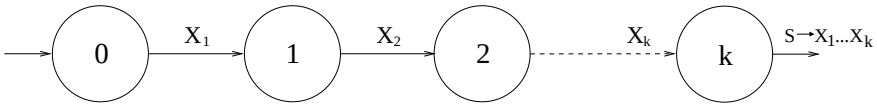


FIG. 7.2 – Une branche de l'automate

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 7.7. *Une production pointée d'une grammaire hors-contexte G est un triplet (A, α, β) de $V \times (V \cup \Sigma)^\star \times (V \cup \Sigma)^\star$, 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 εentre les deux états initiaux, indiquant que l'empilage de X_1 se résoudra en commençant l'empilage de Y_1 (voir la Figure 7.3). S'il existe plusieurs X_1-productions, on aura une branche (et une transition ε) par production.

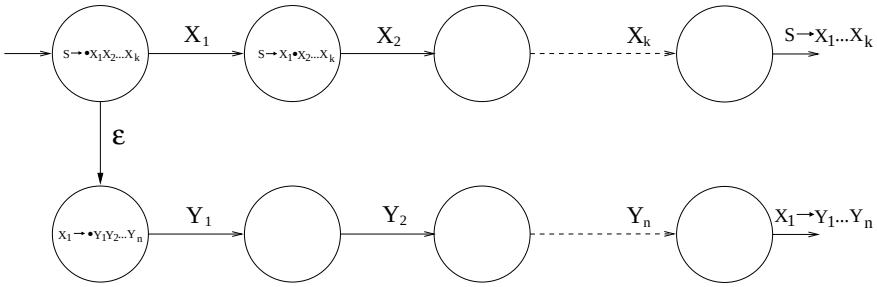


FIG. 7.3 – Deux branches de l'automate

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.

De ces remarques, découle un procédé systématique pour construire un ε-NFA $(V \cup \Sigma, Q, q_0, F, \delta)$ permettant de guider une analyse ascendante à partir d'une grammaire $G = (\Sigma, V, Z, P)$, telle que Z est non-récursif et ne figure en partie gauche que dans l'unique règle $Z \to \alpha$:
- $Q = \{[A \to \alpha \bullet \beta]$ avec $A \to \alpha\beta \in P\}$
- $q_0 = [Z \to \bullet\alpha]$
- $F = \{[A \to \alpha\bullet]$ 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. $B \in V, \forall q' = [B \to \bullet\gamma], \delta(q, \varepsilon) = q'$.

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

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

Théorème 7.2. *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) $\exists \delta, \gamma = \delta\alpha$*
– *(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 page 33), puis la construction des sous-ensembles décrite à la page 30 ; 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 de la Table 7.15 un automate ressemblant fortement à celui de la Figure 7.1. On déduit finalement de cet automate déterministe, par le procédé utilisé pour construire la Table 7.17, la *table d'analyse LR(0)*.

Définition 7.8. *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 $x \in V \cup \Sigma$ 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.

7.2.3 Analyseurs LR(1), LR(k)...

Regard 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 de la Table 7.18 :

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 :
– soit immédiatement réduire T en E
– soit opérer un décalage et continuer de préparer une réduction par $E \to T + E$.

[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).

$$S \rightarrow E\#$$
$$E \rightarrow T + E \mid T$$
$$T \rightarrow x$$

Tab. 7.18 – Une grammaire non-LR(0)

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 notre analyseur ?

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

- $Q = \{[A \rightarrow \alpha \bullet \beta, a]$ avec $A \rightarrow \alpha\beta \in P$ et $a \in \Sigma\}$
- $q_0 = [Z \rightarrow \bullet\alpha, ?]$, où ? est un symbole «joker» qui vaut pour n'importe quel symbole terminal ;
- $F = \{[A \rightarrow \alpha\bullet, a]$, avec $A \rightarrow \alpha \in P\}$; à chaque état final est associée la production $A \rightarrow \alpha$, correspondant à la réduction à opérer ;
- $\forall q = [A \rightarrow \alpha \bullet X\beta, a] \in Q, \delta(q, X) = [A \rightarrow \alpha X \bullet \beta, a]$; comme précédemment, ces transitions signifient la progression de la résolution du but courant par empilement de X ;
- $\forall q = [A \rightarrow \alpha \bullet B\beta, a] \in Q$ tq. $B \in V, \forall q' = [B \rightarrow \bullet\gamma, b]$ tq. $b \in FIRST(\beta a), \delta(q, \varepsilon) = q'$.
  La condition supplémentaire $b \in FIRST(\beta a)$ (voir la section 7.1.3) 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 précédente 7.18. L'état initial de l'automate LR(1) correspondant est $[S \rightarrow \bullet E\#]$; lequel a deux transitions ε, correspondant aux deux façons d'empiler un E, atteignant respectivement les états $q_1 = [E \rightarrow \bullet T + E, \#]$ et $q_2 = [E \rightarrow \bullet T, \#]$. Ces deux états se distinguent par leurs transitions εsortantes : dans le cas de q_1, vers $[T \rightarrow \bullet\, x, +]$; dans le cas de q_2, vers $[T \rightarrow \bullet\, x, \#]$. 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 $E \rightarrow T + E$; si c'est un '#', il faut réduire selon $E \rightarrow T$.

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 $q = [A \rightarrow \alpha\bullet, a]$, la case $T(q, a)$ contient l'unique action (réduire la pile selon $A \rightarrow \alpha$) : la décision de réduction (ainsi que la production à appliquer) est maintenant

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

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 Table 7.18 et d'en déduire un analyseur déterministe pour cette grammaire. Idem pour la grammaire de la Table 7.19, qui engendre des mots tels que $x = * * x$.

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

Tab. 7.19 – Une grammaire pour les manipulations de pointeurs

Pour vous aider dans votre construction, la Figure 7.4 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 :
– 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 3.1.4)
– 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 9, qui utilise deux fonctions auxiliaires pour effectuer directement la déterminisation.

[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-contextes susceptibles d'être analysés par une procédure déterministe ! Nous aurons l'occasion de revenir en détail sur cette question au chapitre 9. 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 atteintes des humains. C'est ainsi que le mot `if cnd1 then if cond2 then inst1 else inst2` sera plus naturellement interprétée `if cnd1 then (if cond2 then inst1 else inst2)` que `if cnd1 then (if cond2 then inst1) else inst2` en laissant simplement la priorité au décalage du `else` qu'à la réduction de `if cond2 then inst1`. b

Algorithm 9 – Construction de l'automate d'analyse LR(1)

```
// Programme principal
begin
   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)}
            od
         od
         // test de stabilisation
         if (Q = Q_i ∧ T = T_i) then break fi
   od
end
// Procédures auxiliaires
Closure(q) // Construction directe de la ε-fermeture
begin
   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]
               od
            od
         od
         // test de stabilisation
         if (q = q_i) then break fi
   od
   return(q)
end
Successor(q, X) // Développement des branches
begin
   r := ∅
   foreach [A → β • Xγ, a] ∈ q do
      r := r ∪ [A → βX • γ, a]
   od
   return(Closure(r))
end
```

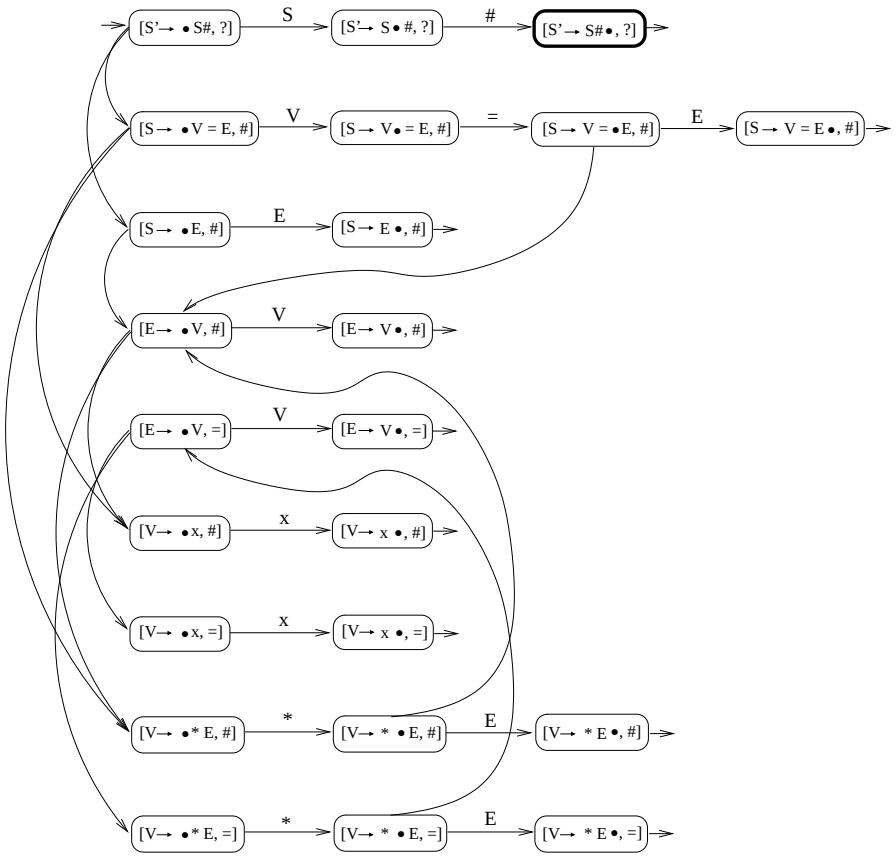


FIG. 7.4 – Les réductions licites pour la Table 7.19
Pour améliorer la lisibilité, les εne sont pas représentés. L'état d'acceptation est représenté en gras.

En guise d'application, vous vérifierez que la mise en œuvre de l'algorithme 9 construit directement le déterminisé de l'automate de la Figure 7.4.

7.2.4 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 3.1.4) 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'analyseur 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. Nous aurons l'occasion de revenir beaucoup plus en détail sur les analyseurs (*LA*)*LR* et leurs multiples variantes au chapitre 9.

Chapitre 8

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 6 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.

8.1 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.

8.1.1 Quelques préliminaires

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

Définition 8.1 (Sous-grammaire). *Soit $G = (V, \Sigma, S, P)$ une grammaire CF. On appelle* sous-grammaire *toute grammaire $G' = (V, \Sigma, S, P')$, 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.

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

Définition 8.2 (Langage engendré par un non-terminal). *Soit $G = (V, \Sigma, S, P)$ une grammaire CF. On appelle langage engendré par le non-terminal A le langage $L_A(G)$ engendré par la grammaire $G' = (V, \Sigma, A, P)$.*

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

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

Preuve : 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 $A \Rightarrow_G^\star \alpha$.

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

Lemme 8.2 («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 $\{A \to \alpha_1\beta_1\alpha_2, \ldots, A \to \alpha_1\beta_n\alpha_2\}$ engendre le même langage que G.*

La démonstration de ce second résultat est laissée en exercice.

8.1.2 Non-terminaux inutiles

La seule contrainte définitoire posée sur les productions des CFG est que leur partie gauche soit réduite à un non-terminal unique. 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, ne peuvent apparaître dans 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 8.3 (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 tel que $S \Rightarrow_G^\star xAy \Rightarrow_G^\star x\alpha y \Rightarrow_G^\star w$. 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 $\{G_A = (V, \Sigma, A, P), A \in V\}$. $L(G_A)$ contient donc l'ensemble des mots qui se dérivent depuis le non-terminal A. Il existe un algorithme (cf. la section 5.3) 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 :
  - $L(G') \subset G$, 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.
  - $L(G) \subset G'$: s'il existe un mot u tel que $S \overset{\star}{\Rightarrow}_G u$ et $S \overset{\star}{\mapsto} G'u$, 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 V_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, V_U contient A, alors on ajoute à P_U toutes les règles dont A est partie gauche, et à V_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 non-terminal ne peut-être ajouté à V_U. Par construction, pour toute production $A \to \alpha$ de P_U, il existe α_1 et α_2 tels que $S \overset{\star}{\Rightarrow}_G \alpha_1 A \alpha_2 \Rightarrow_G \alpha_1 \alpha \alpha_2$.
  En supprimant de G tous les terminaux qui ne sont pas dans V_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 de la Table 8.1.

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

Tab. 8.1 – Élimination des productions inutiles : l'ordre importe

8.1.3 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 (autres que ceux engendrés par la partie droite de la règle).

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

[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.

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 8.1 (Élimination des cycles). *Soit G une CFG, alors il existe une CFG G', équivalente à G, qui ne contient aucune production de la forme $A \to B$, où A et B sont des non-terminaux.*

Preuve. 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'autre 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.

Définition 8.4. *$B \in V$ dérive immédiatement non-génèrativement de A dans G si et seulement si $A \to B$ est une production de G. On notera $A \mapsto_G B$.*

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

$$A \mapsto_G X_1 \dots \mapsto_G X_n \mapsto_G B$$

Pour chaque variable A, il est possible, par un algorithme de parcours du graphe de la relation $\mapsto_G$, de déterminer de proche en proche $C_A = \{A\} \cup \{X \in V, A \overset{\star}{\mapsto}_G X\}$.

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 $\alpha \notin V$. 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 $X_1 \dots X_k$ 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 $A \to ...X \to \alpha$ 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 de la Table 8.2. Le calcul de la clôture transitive de $\mapsto_G$ 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 G' contient alors les productions listées dans la Table 8.3, qui correspondent aux quatre seules productions génératives : $a \to aB$, $B \to bb$, $C \to aAa$, $C \to Aa$.

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

TAB. 8.2 – Une grammaire contenant des productions non-génératives

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

TAB. 8.3 – Une grammaire débarrassée de ses productions non-génératives

8.1.4 Productions ε

Si l'on accepte la version libérale de la définition des grammaires CF (cf. la discussion de la section 4.2.6), 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 εpeut 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 8.2 (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 $G' = (V \cup \{S'\}, , \Sigma, S', P')$ 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 production εsauf 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 (ie. 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.

Preuve : 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 V_ε des variables A de G telles que le langage engendré par A, $L_A(G)$, contient ε. Quels sont-ils ? Un cas évident correspond aux productions $A \to \varepsilon$. Mais εpeut également se déduire depuis A par deux règles $A \to \alpha \to \varepsilon$, à condition que tous les symboles de α soient eux-mêmes dans V_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 7.1.3).

Une fois V_ε calculé, la transformation suivante de G conduit à une grammaire G' faiblement équivalente : G' contient les mêmes axiome, non-terminaux et terminaux que G. De surcroît, G' contient toutes les productions de G n'impliquant aucune variable de V_ε. Si maintenant G contient une production $A \to \alpha$ et α inclut des éléments de V_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 V_ε. Finalement, si S est dans V_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 V_ε se déroule à l'identique dans G'. Soit maintenant une dérivation impliquant un symbole de V_ε : soit il s'agit de $S \stackrel{\star}{\Rightarrow}_G \varepsilon$ et la production $S \to \varepsilon$ permet une dérivation équivalente dans G' ; soit il s'agit d'une dérivation $S \stackrel{\star}{\Rightarrow}_G \alpha \Rightarrow_G \beta \stackrel{\star}{\Rightarrow}_G u$, avec $u \neq \varepsilon$ et β contient au moins un symbole X de V_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 $\alpha \Rightarrow_G \beta$; 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 les grammaires de la Table 8.4 : G contenant deux terminaux qui dérivent le mot vide ($V_\varepsilon = \{A, B\}$), les productions de G' se déduisent de celles de G en considérant toutes les manières possibles d'éviter d'avoir à utiliser ces terminaux.

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

Tab. 8.4 – Une grammaire avant et après élimination des productions ε

8.1.5 Élimination des récursions gauches directes

On appelle *directement récursifs* les terminaux A d'une grammaire G qui sont tels que $A \stackrel{\star}{\Rightarrow} A\alpha$ (récursion gauche) ou $A \stackrel{\star}{\Rightarrow} \alpha A$ (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 6.3 et 7.1). 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 8.2.2).

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 :

$$A \Rightarrow_G A\alpha_{i_1} \Rightarrow_G A\alpha_{i_2}\alpha_{i_1} \ldots \stackrel{\star}{\Rightarrow}_G A\alpha_{i_n} \ldots \alpha_{i_1}$$

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 x_j. Le principe de la transformation consiste à produire x_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 :

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

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 (eg. 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 des expressions arithmétiques, entrevue à la section 7.1.2 lors de notre présentation des analyseurs LL(1). Cette grammaire, qui contient deux terminaux directement récursifs à gauche, est rappelée[2] dans la Table 8.5.

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

TAB. 8.5 – 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.

8.2 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 d'un algorithme de parsage, ou encore pour accélérer l'analyse. Les principales formes normales (de Chomsky et

[2] A 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 !

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.

8.2.1 Forme normale de Chomsky

Théorème 8.3 (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, $S \Rightarrow^{\star}_{G} \varepsilon$, 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).*

Preuve : Les résultats de simplification obtenus à la section 8.1 nous permettent de faire en toute généralité l'hypothèse que G ne contient pas de production εautre que éventuellement que $S \to \varepsilon$ 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 $G' = (V', \Sigma, S, P')$ selon :
- V' contient tous les symboles de V ;
- 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 $A \to X_1 \dots X_k$, avec tous les X_i dans V est copiée dans P' ;
- soit $A \to X_1 ... X_m$ 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, $A \Rightarrow^{\star}_{G} u$ si et seulement si $A \Rightarrow^{\star}_{G'} u$, puis l'égalité des langages engendrés par ces deux grammaires. En effet, si $A \to u = u_1 \dots u_l$ est une production de G, on a dans G' :

$$A \to A_{u_1} \dots A_{u_l} \to u_1 A_{u_2} \dots A_{u_l} \dots \Rightarrow^{\star}_{G'} u$$

Supposons que cette propriété soit vraie pour toutes les dérivations de longueur n et soit A et u tels que $A \Rightarrow^{\star} u$ en $n+1$ étapes. La première production est de la forme : $A \to x_1 A_1 x_2 A_2 \dots A_k$, où chaque x_i ne contient que des terminaux. Par hypothèse de récurrence, les portions de u engendrés 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_{x_i}, donc $A \Rightarrow^{\star}_{G'} u$. 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 transformer de 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) : $\{A \to B_1\ D_1, D_1 \to B_2\ D_2, \dots, D_{m-2} \to B_{m-1}\ B_m\}$. 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 décrite à la Table 8.6. Le résultat auquel vous devez aboutir est donné dans la Table 8.7.

$S \to bA$	$S \to aB$
$A \to a$	$B \to b$
$A \to aS$	$B \to bS$
$A \to bAA$	$B \to aBB$

Tab. 8.6 – Une grammaire à Chomsky-normaliser

$S \to A_aA$	$S \to A_bB$
$A \to a$	$B \to b$
$A \to A_aS$	$B \to A_bS$
$A \to A_aD_1$	$B \to A_aD_2$
$D_1 \to AA$	$D_2 \to BB$
$A_a \to a$	$A_b \to b$

Tab. 8.7 – 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 ascendant.
– 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.

8.2.2 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 section 8.1.5), 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 8.4 (Forme normale de Greibach). *Tout langage hors-contexte L ne contenant pas εpeut 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 $S \rightarrow \varepsilon$, qui dérive εdepuis l'axiome, comme démontré à la section 8.1.4.

Preuve : 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 de deux procédés déjà présentés, qui transforment une grammaire CF en une grammaire CF faiblement équivalente.

Le premier procédé est celui décrit au lemme 8.1 et consiste à ajouter une production $A \rightarrow \alpha$ à G si l'on observe dans G la dérivation $A \stackrel{\star}{\Rightarrow}_G \alpha$. Le second procédé est décrit dans la section 8.1.5 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 $V = \{A_0 \dots A_n\}$, l'axiome recevant conventionnellement le numéro 0. On montre alors que :

Lemme 8.3. *Soit G une grammaire CF. Il existe une grammaire équivalente G' dont toutes les productions sont de la forme :*
– $A_i \rightarrow a\alpha$, *avec* $a \in \Sigma$
– $A_i \rightarrow A_j\beta$, *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 8.1.4), alors toutes les productions dont $S = A_0$ est partie gauche satisfont par avance la propriété du lemme 8.3. Supposons que l'on a déjà traité les non-terminaux numérotés de 0 à $i-1$, et considérons le non-terminal A_i. Soit $A_i \rightarrow \alpha$ une production dont A_i est partie gauche, telle que α débute par une variable A_j. Trois cas de figures 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 8.2 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 8.3 : 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 8.1.5, 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 (récusions 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 8.3, permettant à la récurrence de se poursuivre.

Notons qu'à l'issue de cette étape 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 7.1).

Cette première partie de la construction est illustrée par la Table 8.8.

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	$S \to A_2A_2$ $A_1 \to aA_1 \mid aA_1A_3$ $A_2 \to aA_1A_1 \mid aA_1A_3A_1 \mid b \mid aA_1A_1A_4 \mid aA_1A_3A_1A_4 \mid bA_4$ $A_3 \to aA_1 \mid aA_1A_3 \mid aA_1A_3A_3$ $A_4 \to aA_1 \mid aA_1A_4 \mid aA_1A_3 \mid aA_1A_3A_4$	

Tab. 8.8 – Une grammaire en cours de Greibach-normalisation

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 : $A_{n+p} \to a\alpha$, 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 8.2, 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 la grammaire de départ et qui, de plus, est elle 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 8.4, 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).

Deuxième partie

Compléments : Algorithmes d'analyse

Cette partie correspond au cours “Compléments sur les grammaires” de la brique MFI.

Chapitre 9

Automates à pile et langages déterministes

Dans ce chapitre, nous introduisons un nouveau type d'automate fini, les *automates à pile*. Ces automates constituent une généralisation des automates finis étudiés au chapitre 3. Après avoir donné différentes définitions de ces nouvelles machines, et précisé leur fonctionnement, nous montrons le résultat principal de ce chapitre : les automates à pile (non-déterministes) reconnaissent exactement les langages hors-contexte. Nous complétons ce chapitre par une caractérisation des langages déterministes, et par un énoncé de quelques résultats importants concernant ces langages.

9.1 Automates à piles

9.1.1 Concepts

Un exemple introductif

Le modèle des automates finis souffre d'une limitation manifeste : sa mémoire du calcul est limitée et intégralement résumée par l'état courant de l'automate. En conséquence, un automate fini ne peut prendre en compte qu'un nombre fini de configurations différentes (cf. la discussion de la section 3.3.1). Un moyen simple de contourner cette limitation est de doter l'automate fini d'une mémoire infinie, que nous allons modéliser par une pile[1]. Cette pile permet de garder une mémoire (non-bornée en taille) des étapes de calculs passées, mémoire qui peut conditionner les étapes de calcul à venir.

Le calcul d'un automate fini étant principalement défini à partir de sa fonction de transition, le modèle d'automate à pile enrichit cette fonction de transition de la manière suivante :
– (i) il existe un nouvel alphabet fini contenant les symboles qui peuvent être empilés et dépilés ;
– (ii) les transitions sont conditionnées par le symbole trouvé en haut de la pile ;
– (iii) lors d'une transition dans l'automate, il est de plus possible de modifier (par un *push* ou un *pop*) la pile ;
Avant d'introduire formellement ces machines, considérons l'automate fini A de la Figure 9.1, qui reconnaît le langage $\{a^n b^m, n, m > 0\}$ et essayons d'étendre ce modèle pour en dériver un automate à pile capable de reconnaître $\{a^n b^m, m = n > 0\}$. L'automate A est incapable de "compter" le nombre

[1] Rappel : une pile est une structure de données sur laquelle n'agissent que deux opérations : l'empilage d'un nouvel élément en haut de la pile (*push*) et le dépilage de l'élément positionné au sommet de la pile (*pop*).

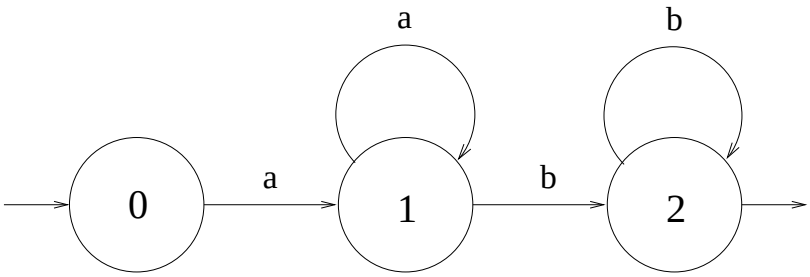


FIG. 9.1 – Un automate pour $a^n b^m$

de a déjà vus, ce qui interdit d'imposer la contrainte $n = m$. Confions alors cette tâche au mécanisme de pile, en empilant un symbole Z lors de chaque passage dans la boucle de l'état 1 et en dépilant un symbole lors de chaque passage dans la boucle de l'état 2. Si l'on impose, de plus, qu'un calcul réussi dans A doit correspondre à une pile intégralement vide, alors il apparaît que chaque mot reconnu par l'automate fini A augmenté de la pile est bien tel que le nombre de a est exactement égal au nombre de b. Toute autre situation correspond ou bien à un calcul terminant avec une pile non vide (si $n > m$), ou bien à une commande de dépilage d'une pile vide (si $m > n$). La Table 9.1 donne la trace[2]du calcul pour l'entrée *aaabbb* :

Input	État	Pile
aaabbb	0	ε
aabbb	1	ε
abbb	1	Z
bbb	1	ZZ
bb	2	ZZ
b	2	Z
	2	

TAB. 9.1 – Trace du calcul de *aaabbb*

L'automate à pile

Nous pouvons maintenant définir formellement un automate à pile :

Définition 9.1 (Automate à pile). *Un* automate à pile *(en anglais pushdown automaton, en abrégé PDA) est un septuplet* $M = (\Sigma, \Gamma, Z_0, Q, q_0, F, \delta)$ *où* Σ *et* Γ *sont deux alphabets finis,* Z_0 *un symbole distingué de* Γ, Q *un ensemble fini d'états contenant un état initial* q_0 *et un sous-ensemble d'états finaux* F, *et* δ *est une fonction de* $Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\})$ *vers l'ensemble des parties de* $Q \times (\Gamma \cup \{\varepsilon\})$.

De manière sous-jacente, un automate à pile est essentiellement un automate fini *non-déterministe*, à la différence près que la fonction de transition δ comporte trois arguments : l'état courant, le symbole d'entrée courant et le symbole courant en haut de la pile, appartenant à un inventaire fini Γ (l'alphabet de pile). Si (r, Z) est un élément de $\delta(q, a, Y)$, alors l'utilisation de cette transition conduira à :
– dépiler Y ; si $Y = \varepsilon$ la transition a lieu indépendamment du symbole en haut de pile, qui reste inchangée
– transiter dans l'état r

[2]Attention : la pile est représentée "latéralement" : le haut de la pile est à droite de la page.

– empiler Z ; si $Z = \varepsilon$, aucun symbole n'est empilé.
Compte-tenu de cette définition, un automate fini "traditionnel" est un automate à pile particulier, défini sur un alphabet de pile vide ($\Gamma = \emptyset$) et dont toutes les transitions laissent la pile inchangée. Comme un automate fini, un automate à pile admet une représentation graphique sur laquelle les modifications de la pile sont représentées sous la forme Y/Z. La Figure 9.2 représente ainsi un automate à pile reconnaissant le langage $\{a^n b^n\}$.

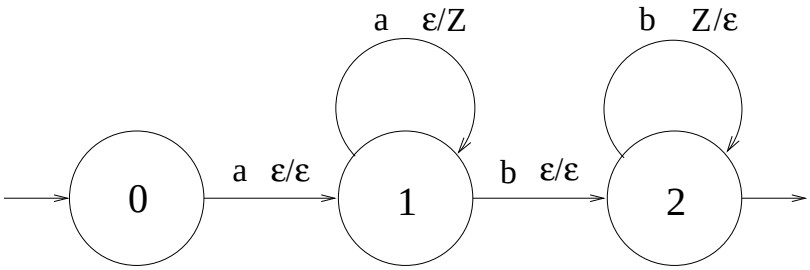


FIG. 9.2 – Un automate à pile pour $a^n b^n$

Configuration, calculs, langages

Si l'on appelle *configuration* de l'automate une description instantanée consistant en l'indication de l'état courant, du contenu de la pile, et du ruban d'entrée, alors le calcul dans un PDA est défini comme une succession de configurations $c_1 \dots c_n$ telles que le passage d'une configuration c_i à la suivante c_{i+1} corresponde à l'application d'une transition étiquetée par le symbole courant du ruban d'entrée a. La possibilité de passage de la configuration c_i à la configuration c_{i+1}, qui définit une relation binaire sur l'ensemble des configurations, est notée par le symbole $\vdash_M$.

La clôture transitive de $\vdash_M$, notée $\vdash_M^\star$, est définie par : $c \vdash_M^\star c'$ s'il existe un ensemble de configurations intermédiaires $c = c_1 \dots c_n = c'$ telles que $c_1 \vdash_M c_2 \dots \vdash_M c_n$.

L'étiquette d'un calcul est obtenue par concaténation des étiquettes le long du calcul ; un calcul réussi est un calcul débutant dans une configuration initialisée en (q_0, Z_0, u) et s'achevant dans un état final.

Ces préliminaires permettent de définir le langage associé à un automate à pile.

Définition 9.2. *Soit M un automate à pile, on note $L(M)$ le langage reconnu par M. $L(M)$ est l'ensemble des mots de $\Sigma^\star$ étiquetant un calcul réussi. Formellement,*

$$L(M) = \{u \in \Sigma^\star \mid (q_0, Z_0, u) \vdash_M^\star (q, \alpha, \varepsilon), q \in F\}$$

Variantes

La définition précédente admet de nombreuses variantes, qui ne changent en rien l'expressivité du modèle. En particulier :
– il est possible de définir δ comme une fonction de $Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\})$ vers l'ensemble des parties finies de $Q \times \Gamma^\star$: selon cette variante, il est possible de remplacer d'un coup le symbole en haut de la pile par un mot de taille quelconque.
La définition 9.1 apparaît comme un cas particulier de ce modèle, auquel on se ramène simplement en rajoutant partout où c'est nécessaire des transitions spontanées qui permettent de

décomposer en une série de transitions élémentaires les transitions empilant des mots de longueur supérieure à 1.
Sauf mention explicite du contraire, c'est cette version un peu plus libérale qui sera utilisée par la suite.
– la condition d'arrêt qui caractérise un calcul réussi peut également être modifiée. Il est en particulier possible de définir un calcul réussi comme : un calcul se terminant avec une pile vide (ce qui revient implicitement à considérer que tout état est final) : on parle *d'acceptation par pile vide* ; par opposition à l'*acceptation par état final*, qui est ici le mode d'acceptation par défaut.
En revanche, pour un PDA donné, définir un calcul réussi d'une manière ou d'une autre change naturellement le langage reconnu. Vous pourrez vous en persuader en faisant varier le mode d'acceptation pour l'automate à pile dont les transitions sont dans la Table 9.2, et en considérant pour les deux modes possibles d'acceptation les langages correspondants.

$$\Sigma = \{a, b\}, Q = \{q_0, q_1, q_2, q_3\}, \Gamma = \{X_1, X_2\}, Z_0 = X_1, F = \{q_3\}$$

$\delta(a, q_0, X_1) = (q_1, X_2X_1)$	$\delta(a, q_0, X_2) = (q_2, X_2X_2X_1)$
$\delta(a, q_1, X_2) = (q_1, X_2X_1)$	$\delta(\varepsilon, q_1, X_1) = (\varepsilon, q_3)$
$\delta(a, q_2, X_1) = (q_2, X_2X_2X_1)$	$\delta(a, q_0, X_1) = (q_1, X_2X_1)$
$\delta(\varepsilon, q_2, X_1) = (q_3, \varepsilon)$	$\delta(b, q_3, X_2) = (q_3, \varepsilon)$

TAB. 9.2 – Un automate à pile reconnaissant des langages variables

La démonstration de l'équivalence entre PDAs reconnaissant par pile vide et PDA reconnaissant par état final s'esquisse comme suit. Si M reconnaît L en acceptant par état final, alors il suffit d'ajouter à M :
– (i) un nouvel état initial q'_0 et un nouveau symbole initial de pile Z'_0 ; ainsi qu'une transition εde q'_0 vers q_0 empilant Z_0 ;
– (ii) un état de "vidange de pile" q_v, vers lequel il existe une transition εdepuis tout état final de M. q_v est le seul état dans lequel Z'_0 est dépilé.
– (iii) pour le reste, M' simule le comportement de M ;
On obtient ainsi un automate qui clairement reconnaît le même langage que L en acceptant par pile vide.

Inversement, si M reconnaît L par pile vide, alors on construit un PDA équivalent M' en faisant en sorte que :
– (i) d'empiler initialement un nouveau symbole, Z'_0, absent de l'alphabet de pile de M ;
– (ii) tout état de M admette des transitions spontanées vers un nouvel état F, qui est le seul état final de M'. Ces transitions ne sont utilisables que lorsque Z'_0 figure en haut de la pile.
– (iii) pour le reste, M' simule le comportement de M ;
La condition (iii) permet de garantir que, lorsqu'un calcul réussit (par vidage de la pile) dans M, la pile de M' est réduite au seul symbole Z'_0. Il est alors possible de transiter vers l'état final de M' et donc de réussir le calcul également dans M'.

9.2 Les langages des automates à pile

9.2.1 PDA et langages hors contexte

Le résultat principal de cette section s'énonce par le théorème 9.1.

Théorème 9.1. *Les langages reconnus par les automates à pile sont exactement les langages hors-contexte.*

La démonstration de ce résultat se fait en deux temps. Montrons tout d'abord que si L est engendré

par une CFG $G = (\Sigma, V, S, P)$, alors il existe un PDA M qui reconnaît L. Sans perte de généralité, on peut supposer que G est sous forme normale de Greibach (voir section 8.2.2), garantissant que toute production de G est de la forme $A \to a\alpha$, avec $a \in \Sigma$ et $\alpha \in (\Sigma \cup V)^*$. Construisons alors le PDA mono-état $M = (\Sigma, V, S, \{q_0\}, q_0, F, \delta)$, dont les transitions sont définies de la manière suivante (rappelons que γ^R dénote le miroir de γ) :

$$(q_0, \gamma^R) \in \delta(q_0, a, A) \text{ si et seulement si } (A \to a\gamma) \in P$$

Essentiellement, cette construction très simple fait porter tout le poids de la modélisation sur la pile, qui est utilisée pour simuler les dérivations dans G. Le langage reconnu (avec acceptation par pile vide) par la machine M est exactement L. En effet, on montre par récurrence sur la longueur de u, l'équivalence suivante :

$$S \Rightarrow_G^\star u\gamma \text{ si et seulement si } (q_0, S, u) \vdash_M^\star (q_0, \gamma^R, \varepsilon)$$

Cette équivalence est vérifiée pour $|u| = 1$ par définition de M. Supposons qu'elle est vraie pour tous les mots jusqu'à la longueur n et considérons le mot u de longueur $n+1$. Posons alors $u = va$ et décomposons une dérivation gauche de $w = ux$ en : $S \Rightarrow_G^L vA\beta \Rightarrow_G va\alpha\gamma$. L'hypothèse de récurrence nous garantit qu'il existe, pour l'entrée v, un calcul dans M aboutissant dans l'état q_0, et tel que la pile soit égale à $\beta^R A$. La production $A \to a\alpha$ de P implique l'existence dans M d'une transition $\delta(q_0, a, A)$ vers q_0 et empilant α^R. Le calcul dans M peut donc se poursuivre, aboutissant à une configuration $(q_0, \beta^R\alpha^R, \varepsilon)$.

On déduit immédiatement que les mots de $L(G)$, ie. qui sont tels que $S \Rightarrow^\star u$ sont précisément les mots reconnus par pile vide dans $L(M)$ et réciproquement.

Montrons maintenant la propriété réciproque, à savoir que si un langage est reconnu par un PDA M, alors il existe une grammaire G qui l'engendre. La preuve repose sur la construction suivante. Soit $M = (\Sigma, \Gamma, Z_0, Q, q_0, F, \delta)$ un PDA reconnaissant L, on construit sur Σ la grammaire G définie par :

- les non-terminaux de G sont tous les triplets $[q, X, r]$ de $Q \times \Gamma \times Q$, auxquels on adjoint un axiome S. Il y a donc bien un nombre fini de non-terminaux.
- P, l'ensemble des productions de G contient $S \to [q_0, Z_0, q]$ pour tous les états $q \in Q$.
- pour chaque transition $\delta(q, a, X)$ contenant $(p, Y_1 \dots Y_l)$ de M, on ajoute à P toutes les productions de la forme $[q, X, p] \to a[q_1, Y_1, q_2][q_1, Y_2, q_2] \dots [q_l, Y_l, p]$ pour toutes les valeurs possibles de p et de $q_2 \dots q_l$.
  Lorsque $Y_1 \dots Y_l = \varepsilon$, on ajoute dans P la seule production $[q, A, p] \to a$.

L'intuition de cette construction est la suivante : elle vise à ce que les dérivations gauches dans G d'un analyseur descendant lisant le mot u simulent (dans la pile de l'analyseur) le comportement du PDA lisant cette même entrée.

Formellement, nous allons alors montrer que pour tout symbole de pile X :

$$(q, X, u) \vdash_M^\star (p, \varepsilon, \varepsilon) \text{ si et seulement si } [q, A, p] \Rightarrow_G^\star u$$

Commençons par montrer l'implication directe, qui repose sur une induction sur la longueur des calculs. Si $(q, X, u) \vdash (p, \varepsilon, \varepsilon)$, c'est donc que (p, ε) appartient à $\delta(q, X, u)$, et que donc il existe une production $[q, X, p] \to u$ dans P. Considérons donc un calcul de $k+1$ étapes dans M, débutant par $(q, X, au) \vdash (p, Y_1 \dots Y_l, u)$, et qui au final conduit à $(r, \varepsilon, \varepsilon)$. Deux remarques permettent de conclure :

- u se factorise en $u = u_1 \dots u_l$, avec : $\forall q_i, \exists q_{i+1}, (q_i, Y_i, u_i) \vdash_M^\star (q_{i+1}, \varepsilon, \varepsilon)$; puisque chacun de ces calculs comprend au plus k étapes, l'hypothèse de récurrence nous assure que : $\forall q_i, \exists q_{i+1}, [q_i, Y_i, q_{i+1}] \Rightarrow_G^\star u_i$;

– la première étape de calcul implique l'existence d'une transition $\delta(q, a, X)$ contenant $(p, Y_1 \dots Y_l)$ et donc de productions $[q, X, p] \rightarrow a[q_1, B_1, q_2][q_1, B_1, q_2] \dots [q_l, B_l, p]$ dans P

Il est alors assuré que $[q, X, p] \stackrel{\star}{\Rightarrow}_G au$. On déduit que tout mot du langage reconnu par M donnant lieu à un calcul $[q_0, Z_0, u] \vdash^\star [q, \varepsilon, \varepsilon]$ appartient également au langage de G.

La réciproque se montre de manière identique par une induction sur la longueur des dérivations. Cette partie de la démonstration est laissée en exercice.

9.3 Les langages déterministes

Un résultat bien connu concernant les automates finis est que tout automate non-déterministe admet un automate déterministe équivalent, dont la construction peut toutefois entraîner une explosion combinatoire du nombre d'états (voir la section 3.1.4). Nous allons le voir dans cette section, cette propriété ne s'étend pas aux PDA. La contrainte de déterminisme permet alors de définir ce qui est une sous-classe propre des langages hors-contexte, à savoir les langages déterministes.

9.3.1 Automate à pile déterministe

Définition 9.3 (Automate à pile déterministe (DPDA)). *Un automate à pile M est déterministe s'il vérifie les trois conditions suivantes :*
– $\forall q \in Q, X \in \Gamma \cup \{\varepsilon\}, a \in \Sigma$: $\delta(q, a, X)$ *contient au plus un élément*
– $\forall q \in Q, X \in \Gamma \cup \{\varepsilon\}$: $\delta(q, \varepsilon, X)$ *contient au plus un élément*
– $\forall q \in Q, X \in \Gamma \cup \{\varepsilon\}, a \in \Sigma$: *soit* $\delta(q, a, X)$ *est vide, soit* $\delta(q, \varepsilon, X)$ *est vide*

Ces conditions visent à garantir que les calculs dans le DPDA s'effectuent de manière déterministe : pour chaque configuration et contenu du ruban d'entrée, il n'existe qu'une seule possibilité de développer le calcul, empruntant soit une transition étiquetée par un symbole, soit une transition ε. Une conséquence directe est que la reconnaissance d'un mot s'effectue en un temps linéaire en fonction de la taille de l'entrée. Ces préliminaires permettent de définir le langage associé à un automate à pile.

Définition 9.4 (Langage d'un DPDA). *Si M un automate à pile déterministe, le langage reconnu par M est :*

$$\{u \in \Sigma^\star \mid (q_0, Z_0, u) \vdash^\star_M (q, \gamma, \varepsilon) \text{ avec } q \in F\}$$

Le point important de cette définition est que, pour les automates à pile déterministes, *le seul mode de reconnaissance est la reconnaissance par état final*. Dans ce cas, il n'y pas d'équivalence avec la reconnaissance par pile vide, comme l'exemple suivant nous le montre.

Soit en effet L le langage suivant : $L = a^* \cup \{a^n b^n\}$. Il existe un DPDA qui reconnaît L par état final et nous allons voir qu'il ne peut en exister qui le reconnaisse par pile vide. Partons d'une première observation : si u et v sont deux mots de $L(M)$, avec v un préfixe propre de u (par exemple $u = aaabbb$ et $v = aa$), alors le calcul reconnaissant u débute comme le calcul reconnaissant v (à cause du déterminisme) et ces deux calculs aboutissent à une configuration dans laquelle la pile est vide (c'est la condition d'arrêt). De manière générale, c'est le cas dans notre langage de tous les mots de la forme $v = a^p$, qui sont tous reconnus avec une pile vide. Si l'on choisit que $p > n$, avec n le nombre d'états du D, alors le calcul reconnaissant a^p passe deux fois par le même état q_i, aux

j-èmes et k-ièmes étapes de calcul, dans une configuration où la pile est toujours vide. Le calcul reconnaissant $a^p b^p$ est donc de la forme :

$$(q_0, Z_0, \varepsilon) \vdash_M \ldots (q_i, \varepsilon, a^p b^p) \vdash_M (q_i, \varepsilon, a^{p-j} b^p) \ldots (q_i, \varepsilon, a^{p-k} b^p) \ldots \vdash \ldots (a, \varepsilon, \varepsilon), q \in F$$

Il existe alors un calcul court-circuitant la boucle $q_i \ldots q_i$, qui reconnaît par pile vide $a^{p-k+j} b^p$, alors que ce n'est pas un mot de L. Dans le cas déterministe, la condition de reconnaissance par pile vide ne permet pas de définir les mêmes langages que la condition de reconnaissance par état final ; et c'est donc à cette dernière définition que nous nous tiendrons.

Il est alors possible de définir la classe des langages (hors-contexte) déterministes :

Définition 9.5. *L est un langage déterministe s'il existe un automate à pile déterministe qui le reconnaît.*

9.3.2 Propriétés des langages déterministes

Qui sont les déterministes ?

Commençons par rassembler les éléments épars dont nous disposons pour l'instant :
- tous les rationnels sont certainement déterministes, un automate fini déterministe (sans pile) étant une forme simplifiée de DPDA ;
- la section précédente nous a fait rencontrer au moins un langage CF non rationnel et pourtant déterministe, $L = a^* \cup \{a^n b^n\}$;

La question suivante consiste alors à se demander cette classe ne se confond pas finalement avec celle des langages CF. L'existence de langages CF intrinsèquement ambigus nous assure qu'il n'en est rien et que cette classe ne contient certainement pas tous les langages CF.

Il est toutefois important de réaliser que la notion de déterminisme ne recouvre en rien celle d'ambiguïté et qu'il existe donc des langages qui sont non déterministes sans pour autant être ambigus. C'est le cas, en particulier, du langage $\{a^n b^n, n > 0\} \cup \{a^n b^{2n}, n > 0\}$, pour lequel il est impossible de décider de manière déterministe si les b doivent se lire par paire ou isolément. Il faudrait, pour pouvoir faire le bon choix, avoir lu au moins autant de b que de a, soit se doter d'une capacité de regard avant non bornée. Pourtant, ce langage n'est pas ambigu, car il est engendré par la grammaire (non-ambiguë) dont les productions sont $\{S \to aTb, T \to \varepsilon, S \to aUbb, U \to \varepsilon\}$. La propriété de déterministe porte donc sur "l'analysabilité" d'un langage, alors la propriété d'ambiguïté caractérise l'association (bi-univoque ou non) d'un mot et d'une structure (un arbre de dérivation).

Clôture

La classe des langages déterministes est donc une sous-classe propre des langages CF. Il est de surcroît possible de montrer que :

1. le complémentaire d'un langage déterministe est également déterministe
2. l'intersection entre un langage déterministe et un langage rationnel est déterministe
3. si L est déterministe, alors $Min(L)$, défini par $Min(L) = \{u \in L \text{ tq. } Pref(u) \cap L = \{u\}\}$, est également déterministe ;
4. si L est déterministe, alors $Max(L)$, défini par $Max(L) = \{u \in L \text{ tq. } u \in Pref(x) \Rightarrow x \notin L\}$, est également déterministe.

En revanche, la classe des langages déterministes n'est pas stable pour les opérations d'intersection, d'union, de concaténation et d'étoile.

9.3.3 Déterministes et LR

Commençons par revenir sur la notion de langage LR(k), en introduisant une définition qui s'exprime directement par une condition sur les dérivations d'une grammaire engendrant G, indépendamment donc de toute stratégie d'analyse particulière. Dans cette section, comme dans toute la discussion sur les analyseurs LR, # dénote un symbole terminal supplémentaire distinct de tous les symboles de Σ et qui sert à marquer la fin de l'entrée.

Définition 9.6 (Langages LR(k)). *Un langage L est LR(k) s'il existe une grammaire G et un entier k tels que la condition suivante est vraie pour tout proto-mot $\alpha\gamma uv$, avec α, γ deux proto-mots et u et v deux mots de $\Sigma^\star$ tels que $|u| > k$*[3] *:*

$$\begin{cases} S\#^k \overset{L}{\Rightarrow} \alpha A u v \overset{L}{\Rightarrow} \alpha\gamma uv \; et \\ S\#^k \overset{L}{\Rightarrow} \beta B w \overset{L}{\Rightarrow} \alpha\gamma ut \end{cases} \Rightarrow (\alpha = \beta, A = B, w = ut)$$

On retrouve, dans cette définition, l'idée principale de l'analyse (déterministe, ascendante) LR présentée à la section 7.2 : dans une dérivation droite, la connaissance de la pile ($\alpha\beta$) et des k symboles terminaux (non-encore analysés) qui la suivent (u) suffit à déterminer sans ambiguïté possible l'opération à appliquer dans une dérivation droite qui réussira (ici $A \to \gamma$).

Par exemple, la grammaire de la **??** n'est pas LR(1) (elle n'est même LR(k) pour aucune valeur de k) puisque dans cette grammaire on observe simultanément :

$$\begin{cases} S\# \overset{L}{\Rightarrow} aaSbbb\# \overset{L}{\Rightarrow} aaaSbbbb\# \text{ et} \\ S\# \overset{L}{\Rightarrow} aaaSbbbb\# \overset{L}{\Rightarrow} aaaSbbbb\# \end{cases}$$

qui viole la condition de la définition 9.6 pour $A = B = S$, $\alpha = aa$, $\gamma = aSb$, $\beta = aaa$ et $u = b$. En d'autres termes, avec cette grammaire, après avoir empilé n a et un S, on se sait pas si le b qui suit doit être apparié avec un des a (réduction $S \to aSb$) ou bien réduit "tout seul" (réduction $S \to Sb$).

Il est alors possible de démontrer les deux résultats fondamentaux suivants, qui expriment l'équivalence entre langages déterministes et langages LR.
– si L est reconnu par une grammaire LR(k) alors L est déterministe
– si L est déterministe, alors il existe une grammaire LR(1) qui engendre L

[3] Le symbole #, non-inclus dans Σ, marque la fin de l'input.

Chapitre 10

Compléments sur les langages hors-contextes

Ce chapitre introduit un certain nombre de nouveaux résultats concernant les langages CF, qui complètent les résultats et caractérisations énoncés au chapitre 5. Dans un premier temps, nous revenons sur la question de la démarcation entre langages rationnels et langages hors-contextes, en étudiant la question des récursions centrales et de l'enchâssement. Nous y verrons en particulier que l'existence de dérivations contenant des récursions centrales ne suffit pas à caractériser les langages CF. Nous donnons ensuite un autre éclairage sur les classes de langage, en étudiant les lois de croissance des nombres d'occurrences de symboles terminaux dans les mots d'un langage CF ; nous verrons que ce point de vue ne permet pas non plus de démarquer rationnels et hors-contexte. Nous examinons enfin de plus près une troisième propriété des langages CF, à savoir leur capacité à modéliser des appariements entre paires de symboles, et montrons, en étudiant le théorème de Chomsky et Schützenberger, que c'est cette propriété qui fondamentalement confère aux langages CF un surcroît d'expressivité par rapport aux langages rationnels.

10.1 Compléments sur les langages CF

Nous revenons dans cette section sur la caractérisation de la frontière entre langage rationnel et langage CF, en présentant principalement deux résultats. Le premier est que l'intersection d'un langage CF et d'un langage rationnel reste CF. Le second présente une propriété nécessaire pour qu'un langage soit strictement hors-contexte, portant sur le concept d'enchâssement d'un non-terminal.

10.1.1 Langages hors contextes et rationnels

Tout d'abord une nouvelle preuve d'un résultat déjà connu et qui se démontre généralement en faisant appel à la notion d'automate à pile (voir le chapitre 9). La preuve que nous développons ici repose intégralement sur le formalisme de grammaire.

Théorème 10.1. *L'intersection d'un langage rationnel L_1 et d'un langage hors-contexte L_2 est un langage hors-contexte. Si de plus L_2 n'est pas ambigu, alors $L_1 \cap L_2$ n'est pas ambigu.*

Preuve. Soit L_1 un langage rationnel et $A_1 = (\Sigma, Q_1, q_1^0, F_1, \delta_1)$ l'automate canonique (déterministe minimal) de L_1 ; soit L_2 un langage hors-contexte et $G_2 = (\Sigma, V_2, S_2, P_2)$ une grammaire CF sous

forme normale de Chomsky (voir la section 8.2.1) pour L_2. Sans perte de généralité, nous supposons que ni L_1 ni L_2 ne contiennent ε. Nous supposons également que A_1 contient un unique état final q_1^F ; si cela n'est pas le cas, on traitera séparément chacun des langages obtenus en fixant tour à tour comme unique état final chaque élément F_1 ; l'union de ces langages étant égal à L_1.

Nous construisons pour l'intersection $L_1 \cap L_2$ la grammaire G suivante :
- les non-terminaux de G sont tous les triplets de la forme (q, r, X), où q et r sont des états de Q_1, et X un symbole (terminal ou non-terminal) de G_2.
- le symbole initial de G est (q_1^0, q_1^F, S_2)
- les productions de G sont de trois sortes :

  (i) $(p, q, A) \to (p, r, B)(r, q, C)$ pour tout r dans Q_1, et toute production $A \to B\ C$ de G_2 ;

  (ii) $(p, q, A) \to (p, q, a)$ pour toute production $A \to a$ de G_2 ;

  (iii) $(p, q, a) \to a$ si et seulement si $\delta_1(p, a) = q$

L'idée de cette construction est la suivante : les productions de type (i) et (ii) simulent exactement le fonctionnement de G_2, à la différence près qu'elles engendrent des séquences de triplets (p, q, a), qui sont des non-terminaux pour G. La seule possibilité pour éliminer ces non-terminaux consiste à appliquer des productions de type (iii), qui simulent elles le comportement de l'automate A_1. Une séquence de non-terminaux ne peut alors être produite que si elle est de prime abord engendrée par la grammaire, associée à une suite quelconque d'états ; et s'il existe une suite d'états simulant le comportement de l'automate. La spécification du symbole initial de la grammaire garantit de surcroît que cette suite d'états commence dans q_1^0 et s'achève dans q_1^F.

Pour montrer formellement ce résultat, soit $u = u_1 \dots u_n$ un élément de l'intersection $L_1 \cap L_2$. Il existe une dérivation gauche de u dans G_2, ce qui implique que :

$$(q_1^0, q_1^F, S) \Rightarrow_G^\star \quad (q_1^0, q_1, u_1) \dots (q_i, q_{i+1}, u_{i+1}) \dots (q_{n-1}, q_1^F, u_n)$$

Ce résultat vaut pour toute séquence d'états $(q_1^0 \dots q_i \dots q_1^F)$. u étant également dans L_1, il existe un calcul de (q_1^0, u) aboutissant à q_1^F, ce qui entraîne directement que u est dans $L(G)$. La réciproque se démontre de manière similaire.

La seconde partie du théorème découle directement de (i) l'unicité de la dérivation permettant d'aboutir dans G à la suite de pré-terminaux $(p_0, q_0, a_0 \dots (p_n, q_n, a_n)$ et de (ii) l'unicité de la suite d'états calculant un mot u dans A_1. (i) permet de déduire que chaque séquence de symboles pré-terminaux admet dans G une dérivation gauche unique ; (ii) permet d'affirmer que, pour tout mot u, il existe une unique suite de pré-terminaux qui permet de le dériver, correspondant à la séquence d'états calculant u dans A_1.

10.1.2 Enchâssement

Comme nous l'enseigne le lemme de pompage pour les langages CF (voir la section 5.3.1), les mots des langages hors-contexte se caractérisent par la possibilité d'itération simultanée de facteurs situés de part et d'autre d'un facteur central. Pour tout langage CF infini, il existe un entier k tel que, pour mot z de L plus long que k se décompose en $uvwxy$ avec $vx \neq \varepsilon$ et, pour tout i, uv^iwx^iy est également dans L.

Les grammaires linéaires à gauche ou à droite, introduites à la section 4.2.4, sont telles que la dérivation ne se développe que «d'un seul côté» : chaque proto-phrase ne contient qu'un seul non-terminal, qui est soit le plus à gauche (grammaires linéaires à gauche), soit le plus à droite (linéarité droite). En particulier, une grammaire linéaire à gauche ou à droite ne peut contenir de

sous-dérivation de la forme $A \stackrel{\star}{\Rightarrow} \alpha A \beta$ avec simultanément α et β non vides. Dans la mesure où ces grammaires reconnaissent exactement les langages rationnels (voir la section 4.2.4), il est tentant de chercher du côté des récursions centrales la propriété caractéristique des grammaires qui engendrent des langages strictement CF. C'est ce que nous chercherons à faire dans cette section, en introduisant la notion d'enchâssement.

Définition 10.1 (Enchâssement). *Une grammaire G est dite* self-embedding *s'il existe un terminal utile A tel que : $A \stackrel{\star}{\Rightarrow} \alpha A \beta$, où α et β sont des séquences non-vides de terminaux*[1].

S'il est clair qu'aucune grammaire linéaire à gauche ou à droite n'est self-embedding, cette propriété ne suffit pourtant pas à garantir qu'une grammaire hors-contexte n'engendre pas un langage rationnel. Ainsi, par exemple, la Table 10.1, bien que contenant une dérivation $S \stackrel{\star}{\Rightarrow} aSb$, engendre-t-elle $\{a^n b^m, n > 0, m > 0\}$, qui est un langage parfaitement rationnel.

$$S \rightarrow aS$$
$$S \rightarrow Sb$$
$$S \rightarrow ab$$

Tab. 10.1 – Une grammaire "self-embedding" pour $a^n b^m$

Le fait, pour une grammaire, d'être self-embedding ne garantit pas qu'elle qu'elle n'engendre un langage strictement CF. Nous allons toutefois montrer que cette propriété est nécessaire, c'est-à-dire précisément que si une grammaire G ne possède pas cette propriété, alors elle engendre un langage qui est rationnel.

Théorème 10.2. *Soit G une grammaire non self-embedding, alors il existe une grammaire G' linéaire à droite qui est (fortement) équivalente à G, impliquant que $L(G)$ est un langage rationnel.*

La première partie de la démonstration consiste à noter que si G n'est pas self-embedding, alors la mise en forme normale de Greibach de G conduit à une grammaire qui n'est pas non plus self-embedding. L'algorithme de mise sous forme normale de Greibach (voir la section 8.2.2) repose en effet sur l'application itérée de deux transformations (élimination des récursions gauches, application de la "distributivité" à gauche) qui ni l'une ni l'autre n'introduisent des non-terminaux self-embedding.

Sans perte de généralité, nous supposons donc que G est sous-forme normale de Greibach, c'est-à-dire que toute production est soit de la forme $A \rightarrow a\alpha$, avec $a \in \Sigma$, soit $S \rightarrow \varepsilon$. Nous supposerons additionnellement que α ne contient que des éléments non-terminaux : si cela n'était pas le cas, il suffirait de rajouter un nouveau terminal X_a pour chaque terminal a, ainsi que des productions $X_a \rightarrow a$.

Considérons alors les l premières étapes d'une dérivation gauche dans G : chaque étape récrit le non-terminal le plus à gauche en insérant un nouveau terminal, suivi du nouveau non-terminal le plus à gauche, suivi d'un «reste», qui s'ajoute au suffixe de la proto-phrase courante (voir la Table 10.2).

Notons n le nombre de non-terminaux de G, et m la longueur maximale d'une partie droite non-terminale d'une production de G : $\forall A \rightarrow a\alpha, |\alpha| \leq m$. La preuve que nous allons maintenant

[1] Si G ne contient que des terminaux utiles, alors cette définition peut être affaiblie en posant que α et β sont simplement des proto-mots non-vides.

$$
\begin{aligned}
S = A_0 &\Rightarrow a_1 A_1 \beta_1 \\
&\Rightarrow a_1 a_2 A_2 \beta_2 \\
&\Rightarrow a_1 a_2 a_3 A_3 \beta_3 \\
&\Rightarrow \dots \\
&\Rightarrow a_1 \dots a_l A_l \beta_l
\end{aligned}
$$

TAB. 10.2 – Une dérivation gauche

développer repose sur le fait que si G n'est pas self-embedding, alors le suffixe $A_l\beta_l$ des proto-mots a une longueur bornée par mn.

Remarquons d'abord que, pour tout l, $|A_l\beta_l| \leq 1 + l.(m-1)$: chaque étape de la dérivation récrit A_i en $a_{i+1}A_{i+1}\alpha_{i+1}$, soit une croissance nette du reste $A_i\beta_i$ d'un facteur $|\alpha_{i+1}|$, dont la longueur est bornée par $(m-1)$. Supposons alors $|A_l\beta_l| > mn$: ceci n'est possible que si $l > n$ et si au moins n étapes de la dérivation ont conduit *à une croissance stricte* du reste $A_l\beta_l$ par application d'une production $A \to a\alpha$ telle que $|\alpha| \geq 2$. Comme il n'existe que n non-terminaux, ceci signifie qu'au moins une telle production a été utilisée deux fois, donnant lieu à une sous-dérivation : $A \to aA\alpha \stackrel{\star}{\Rightarrow} a \dots A\beta$ *avec* β *non vide* (car $|\alpha| > 0$). Or, puisque G n'est pas self-embedding, il ne peut exister de tel non-terminal. Cette contradiction prouve que l'on a nécessairement $|A_l\alpha_l| \leq mn$. Un corollaire immédiat : les ensembles de terminaux et de non-terminaux étant finis, les séquences susceptibles d'apparaître à droite du non-terminal le plus à gauche dans une dérivation gauche sont en nombre fini.

Partant de ce résultat, nous allons construire une grammaire linéaire à droite G' équivalente à G, dont les non-terminaux $[A\beta]$ sont définis par : $[A\beta]$ est un non-terminal de G' si et seulement si $S \stackrel{\star}{\Rightarrow}_G uA\beta$ avec u dans $\Sigma^\star$, auxquels on ajoute l'axiome $[S]$. Le résultat précédent nous assure que ces non-terminaux sont en nombre fini. Les productions de G' sont définies par :
– $\forall\beta, [A\beta] \to a[\beta]$ dans G' si et seulement si $A \to a$ dans G.
– $\forall\beta, [A\beta] \to a[\alpha\beta]$ dans G' si et seulement si $A \to a\alpha$ dans G.
G' est par construction linéaire à droite et équivalente à G. Toute dérivation dans G de la forme :

$$S \Rightarrow_G a_1A_1\beta_1 \Rightarrow_G a_1a_2A_2\beta_2 \Rightarrow_G \dots a_1a_2A_l\beta_l$$

correspond, de manière bi-univoque, à une dérivation dans G' :

$$[S] \Rightarrow_{G'} a_1[A_1\beta_1] \Rightarrow_{G'} a_1a_2[A_2\beta_2] \Rightarrow_{G'} \dots a_1a_2[A_l\beta_l]$$

dans G'.

Ce résultat nous conduit donc à une nouvelle caractérisation non pas des langages hors-contexte mais plutôt des langages rationnels : un langage est rationnel s'il existe une grammaire CF non-self embedding qui l'engendre.

10.2 Correspondance de Parikh

Dans cette section, nous étudions les langages CF du point de vue de la longueur des mots qu'ils engendrent et des distributions d'occurrences de symboles dans les mots du langage.

Un premier résultat, qui dérive directement du lemme de pompage (cf. la section 5.3.1), est que lorsque l'on range les mots d'un langage CF par longueur croissante, la différence de longueur

entre deux mots successifs est bornée par une constante, suggérant que l'allongement des mots suit une loi de progression majorée par une fonction linéaire. Cette constatation permet d'exclure de la famille des langages CF des langages tels que $\{a^{2^n}, n > 0\}$. Il est alors tentant de chercher dans les contraintes sur les longueurs et les nombres d'occurrences de symboles des caractérisations nouvelles de l'expressivité des langages CFs, toujours par opposition aux langages rationnels. Nous verrons pourtant que de ce point de vue là, ces deux familles de langages sont identiques. Plus précisément, si l'on appelle *lettre-équivalents* deux langages qui contiennent les mêmes mots à des permutations des symboles près, alors nous montrerons que pour chaque langage hors-contexte, il existe un langage rationnel qui lui est lettre-équivalent.

10.2.1 Correspondance de Parikh

Commençons par une définition :

Définition 10.2 (Correspondance de Parikh). *Soit $\Sigma = \{a_1, \ldots, a_n\}$ un alphabet fini et L un langage de $\Sigma^\star$, on appelle correspondance de Parikh l'application de L dans N^n qui à un mot w associe le n-uplet d'entiers $\psi(w) = (|w|_{a_1}, |w|_{a_2} \ldots |w|_{a_n})$ dénombrant le nombre d'occurrences de chaque symbole de Σ dans le mot w.*

Soit, par exemple, pour $\Sigma = \{a, b\}$: $\psi(\varepsilon) = (0, 0)$; $\psi(aabb) = (2, 2)$. Pour continuer, nous aurons également besoin de la définition suivante, qui permet de caractériser des ensembles de n-uplets.

Définition 10.3 (Semi-linéarité). *Un ensemble S de n-uplets d'entiers de N^n est dit* linéaire *s'il existe un ensemble fini $(t_0, \ldots, t_k)$ de n-uplets tel que S est l'ensemble de tous les n-uplets de la forme : $t_0 + \sum_{k>0} x_i.t_i$, avec les x_i parcourant N.*

Un ensemble S de n-uplets est semi-linéaire *s'il s'écrit comme l'union finie d'ensembles linéaires.*

Par exemple, l'ensemble des couples d'entiers dont la différence, en valeur absolue, est égale à 1 est un ensemble semi-linéaire. C'est l'union de deux ensembles linéaires, l'un engendré par $t_0 = (0, 1)$ et $t_1 = (1, 1)$; l'autre par $t_0 = (1, 0)$ et $t_1 = (1, 1)$.

Ces deux définitions préliminaires permettent de poser le résultat majeur de cette section.

Théorème 10.3. *Si L est un langage hors-contexte, alors $\{\psi(w), w \in L\}$ est un ensemble semi-linéaire.*

Preuve. Sans perte de généralité, nous supposons que L ne contient pas ε(dont la présence ou l'absence ne changera rien à la propriété de semi-linéarité) et que $G = (\Sigma, V, S, P)$ est une grammaire *CF* pour L sous forme normale de Chomsky.

Pour chaque partie U de V, on note L_U l'ensemble des mots de L dont la dérivation gauche contient *au moins une occurrence* de chaque variable de U. L est clairement l'union des L_U et, puisque les L_U sont en nombre fini, il nous suffit de montrer que chaque L_U est lui-même semi-linéaire.

Pour chaque partie U de V, nous identifions alors deux ensembles finis d'arbres de dérivation et de mots apparaissant à leur frontière. Rappelons (cf. la section 5.2.2) que l'arbre de dérivation d'un mot w de L est un arbre étiqueté par des symboles de la grammaire, dont la racine est étiquetée par l'axiome, dont tout sous-arbre de profondeur 1 correspond à une production, et dont les feuilles sont étiquetées par les symboles de w. On définit alors :

– T_I est l'ensemble des arbres de dérivation τ tels que : (i) τ a pour racine S ; (ii) chaque variable de U étiquette au moins un nœud interne de τ ; (iii) chaque variable X de U apparaît au plus $|U|$ fois sur un chemin dans τ. Cette dernière condition assure que T_R est un ensemble fini, ainsi que l'ensemble des mots apparaissant à la frontière d'un arbre de T_R.

T_I constitue donc un ensemble d'arbres de dérivation de G contenant les dérivations "minimales" (en nombre d'étapes de dérivation) de L_U.

– pour chaque variable X de U, avec $U \neq S$, on considère l'ensemble $T_A(X)$ des arbres τ possédant les propriétés suivantes : (i) la racine de τ est étiquetée par X ; (ii) X n'étiquette aucun nœud interne de τ ; (iii) X étiquette une feuille unique de τ ; (iv) chaque variable X de U apparaît au plus $|U|$ fois sur un chemin dans τ. Cette dernière condition garantissant, comme précédemment, la finitude de l'ensemble $T_A(X)$.

  Intuitivement, $T_A(X)$ est l'ensemble des arbres de dérivations minimales de la forme $X \stackrel{\star}{\Rightarrow} uXv$, qui correspondent aux dérivations impliquant récursivement le non-terminal X. T_A est l'union des $T_A(X)$ et est donc également un ensemble fini.

L'observation principale qui permet d'éclairer la suite de la démonstration est la suivante : les mots de L_U s'obtiennent en "insérant" (éventuellement de manière répétitive) dans un arbre de T_I des arbres de T_A (voir la Figure 10.1). De cette observation, découle constructivement le résultat principal qui nous intéresse, à savoir la semi-linéarité des n-uplets images par la correspondance de Parikh des langages L_U.

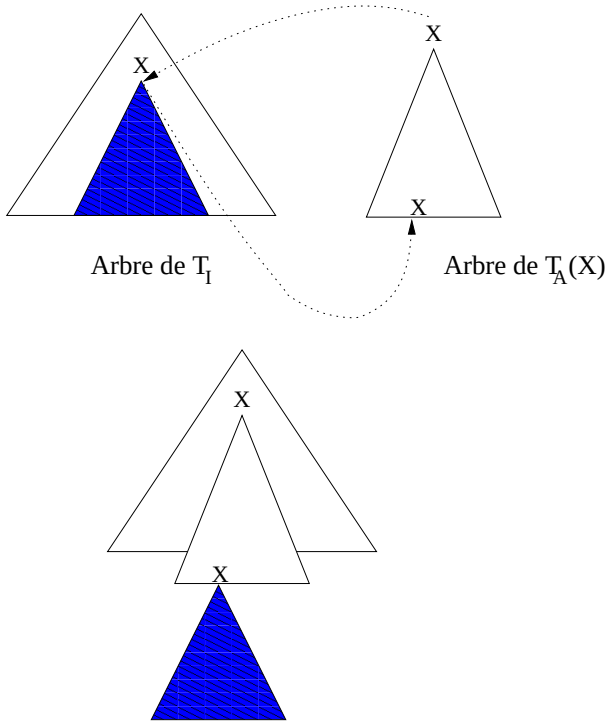


FIG. 10.1 – Un arbre de $T_A(X)$ s'insérant dans un arbre de T_I

Pour continuer, appelons S_I et S_A les ensembles de n-uplets correspondants aux nombres d'occurrences de symboles terminaux à la frontière des arbres de T_I et T_A[2]. Ces deux ensembles de n-uplets, comme les ensembles d'arbres correspondants, sont des ensembles finis. Notons alors $S_I = \{t_1 \dots t_l\}$ et $S_A = \{s_1 \dots s_k\}$ ces deux ensembles et K l'ensemble semi-linéaire de n-uplets défini par :

$$K = \bigcup_{i=1}^{i=l} \{t_i + \sum_{j=1}^{k} x_j.s_j, \forall j, x_j \in N\}$$

Montrons alors que $K = \psi(L_U)$, en commençant par montrer inductivement que $K \subset \psi(L_U)$. Considérons en effet t un n-uplet de S_I : par construction t est l'image par ψ d'un mot de L_U. Soit

[2]On fait abstraction de la variable qui apparaît à la frontière d'un arbre de T_A.

maintenant un n-uplet se décomposant en $t = t_1 + s$, avec t_1 dans $\psi(L_U)$ et s dans S_A. Il existe donc d'une part un mot w_1 dans L_U tel que $\psi(w_1) = t_1$; d'autre part une variable X telle que s est l'image par ψ de la frontière d'un arbre τ de $T_A(X)$. Or, par définition de L_U, l'arbre de dérivation de w_1 contient un nœud interne étiqueté par X. En "excisant" le sous-arbre de racine X et en insérant à sa place le sous-arbre τ, conformément au schéma de la Figure 10.1, on obtient un nouveau mot v de L_U qui est bien tel que $\psi(v) = t$. Ceci permet de conclure que $K \subset \psi(L_U)$.

La réciproque se fonde sur des arguments similaires et nous nous contentons de l'esquisser ici. Soit w un mot de L_U : soit son arbre de dérivation est dans S_I, ce qui permet de conclure directement de $\psi(w)$ est dans K ; soit son arbre de dérivation n'est pas dans S_I, entraînant qu'au moins une variable X apparaît strictement plus que $|\,U\,|$ fois dans la dérivation. L'idée est alors d'opérer l'opération inverse de l'insertion, en extrayant le dernier sous-arbre τ de racine X qui soit dans $T_A(X)$. Cette opération conduit à un arbre de dérivation strictement plus petit que l'arbre original, correspondant au mot v et $\psi(w) - \psi(v) = s_i$ pour un certain i. En procédant inductivement, on se ramène de proche en proche à un mot dont l'arbre de dérivation est dans T_I, ce qui permet finalement de conclure.

10.2.2 Rationnels et les hors-contextes sont lettre-équivalents

Le résultat que nous avons annoncé en début de cette section peut maintenant être établi assez simplement. Rappelons en, pour commencer, l'énoncé.

Théorème 10.4. *Pour tout langage hors-contexte L, il existe un langage rationnel L' qui est lettre-équivalent à L.*

Preuve. Si L est un langage hors-contexte sur un alphabet de n symboles, il existe un nombre fini d'ensembles linéaires de n-uplets $S_1 \dots S_p$ tel que $\psi(L)$ est l'union des S_i. Pour chaque ensemble S_i, il existe un ensemble de n-uplets $\{t^i_0 \dots t^i_{k_i}\}$ tel que $S_i = \{t^i_0 + \sum_{j>0} x_j.t^i_j\}$. Pour trouver un langage lettre-équivalent à L, il suffit d'associer à chaque $t^i_k = (x_1, \dots, x_n)$ le mot w^i_k contenant x_1 fois le symbole a_1, x_2 fois a_2 : $w^i_k = a_1^{x_1} a_2^{x_2} \dots a_n^{x_n}$.

On obtient alors un langage rationnel lettre-équivalent à L en prenant l'union des langages rationnels L_i, où chaque L_i est défini par :

$$L_i = w^i_0 (w^i_1)^* \dots (w^i_{k_i})^*$$

Un corollaire immédiat est que tout langage hors-contexte sur un alphabet d'un seul symbole est rationnel : pour que la modélisation par un langage ou une grammaire hors-contexte fournisse un surcroît d'expressivité par rapport à la modélisation rationnelle, il faut au moins deux symboles dont la modélisation hors-contexte permettra, comme on va le voir, de contrôler les appariements.

10.3 Langages parenthésés, langages hors-contexte

10.3.1 Langages de Dyck

Nous donnons maintenant une nouvelle caractérisation des langages CF qui permet de toucher du doigt à la fois le gain de complexité que ces langages apportent par rapport aux langages rationnels et leur limitation intrinsèque. Pour débuter, nous introduisons un nouveau type de grammaire et de langage, les grammaires et langages de Dyck.

Définition 10.4 (Grammaires et langages de Dyck). *Soit* $\Sigma = \{a_1, a'_1, \ldots a_n, a'_n\}$ *un alphabet composé de* $2n$ *symboles appariés (*a_1 *avec* a'_1 *...), on appelle* langage de Dyck *d'ordre* n*, noté* D_n *le langage engendré par la grammaire hors-contexte suivante :* $(\{S\}, \Sigma, S, \{S \to \varepsilon, S \to SS, S \to a_1 S a'_1, \ldots, S \to a_n S a'_n\}$.

Les couples de symboles terminaux (a_i, a'_i) sont dits *conjugués*. Une autre interprétation des langages de Dyck consiste à définir un processus de réduction des mots de $\Sigma^\star$ qui supprime toute paire de symboles conjugués voisins. Si $w = ua_i a'_i v$, alors w peut se réduire à uv. Il est facile de montrer que la relation de réduction définit une relation d'équivalence sur les mots de $\Sigma^\star$. Le langage de Dyck sur $\Sigma^\star$ est alors simplement l'ensemble des mots de $\Sigma^\star$ qui peuvent se réduire (qui sont congrus par la relation de réduction) au mot vide.

Les grammaires de Dyck sont des grammaires hors-contexte particulières dans lesquelles les terminaux sont toujours insérés par paires d'éléments conjugués. Considérons, par exemple, l'alphabet composé des 3 paires de symboles $\Sigma = \{(,), [,], <>\}$. On observe alors par exemple la dérivation gauche suivante dans D_3 :

$$S \Rightarrow\ (S) \Rightarrow\ (SS) \Rightarrow\ (< S > [S]) \Rightarrow\ (<> [S]) \Rightarrow\ (<> [])$$

Les langages de Dyck possèdent quelques propriétés simples, dont la démonstration est laissée en exercice :

Proposition 10.1. *Si* D_n *est le langage de Dyck sur l'alphabet à* n *symboles, alors les propriétés suivantes sont vraies :*
(i) si u *et* v *sont dans* D_n*, alors* uv *est dans* D_n
(ii) si uv *est dans* D_n *et si* u *est dans* D_n*, alors* v *est dans* D_n
(iii) si u *est dans* D_n*, alors* $a_i u a'_i$ *est également dans* D_n*, pour toute valeur de* i*.*
(iv) tout mot u *non vide de* D_n *se factorise sous la forme* $u = a_i v a'_i w$*, avec* v *et* w *des mots de* D_n*.*
(v) si $a_i a'_i v$ *est un élément de* D_n*, alors* v *est également dans* D_n

10.3.2 Deux nouvelles caractérisations des langages CF

Dans cette section, nous présentons deux théorèmes qui suggèrent que les langages de Dyck représentent, d'une certaine manière, l'essence des langages hors-contexte. Nous commençons par théorème suivant, dû à Chomsky et Schützenberger, qui fournit une caractérisation des langages hors-contexte à partir de la notion de langage de Dyck :

Théorème 10.5 (Chomsky-Schützenberger). *Tout langage hors-contexte est l'image par un morphisme de l'intersection d'un langage rationnel et d'un langage de Dyck.*

Ce premier résultat permet de toucher précisément le surcroît d'expressivité qu'ont les grammaires hors-contexte par rapport aux grammaires rationnelles, à savoir leur capacité à contrôler des paires de symboles appariés ; ce qu'une grammaire régulière est incapable de faire. Ce résultat exprime également de manière très parlante la limitation de la capacité d'expression des grammaires CF : (i) les symboles appariés vont par deux ; (ii) il n'y a qu'un nombre fini de paires.

Preuve : soit L un langage CF, dont nous supposerons sans perte de généralité qu'il ne contient pas ε ; soit $G = (\Sigma, V, S, P)$ une grammaire sous forme normale de Chomsky pour L. Notons $\{a_1 \ldots a_n\}$ les éléments de Σ et $A_i \to B_i C_i, i = 1 \ldots m$ les productions binaires de G. Considérons alors le langage de Dyck D_{n+m} défini sur l'ensemble de terminaux Σ' suivant : $\Sigma' = \Sigma \cup \{a'_1 \ldots a'_n\} \cup \{a_{n+1}, a'_{n+1} \ldots a_{n+m}, a'_{n+m}\}$. Chaque symbole a_i de Σ a son conjugué a'_i dans Σ' ; l'ensemble des

terminaux de D_{n+m} étant complété par une paire de nouveaux symboles terminaux conjugués pour chaque production de G.

Définissons maintenant un morphisme h de $\Sigma'^\star$ dans $\Sigma^\star$ qui efface tous les symboles de Σ'/Σ. Formellement, h est défini inductivement sur $\Sigma'^\star$ par :

$$\begin{cases} h(\varepsilon) = \varepsilon \\ \forall a \in \Sigma, h(a) = a \\ \forall a \in \Sigma'/\Sigma, h(a) = \varepsilon \\ \forall u \in \Sigma'^\star \text{ tel que } |u| > 1, u = av \text{ et } h(u) = h(a)h(v) \end{cases}$$

Pour compléter cette construction, introduisons finalement la grammaire $G' = (\Sigma', V, S, P')$, où P' contient les productions suivantes :
– (i) $\forall A \to a \in P, A \to aa' \in P'$
– (ii) $\forall A \to a \in P, \forall i = 1 \dots m, A \to aa'a'_{n+i}C_i \in P'$
– (iii) $\forall i = 1 \dots m, A_i \to a_{n+i}B_i \in P'$
Cette grammaire étant linéaire à droite, elle engendre un langage rationnel (cf. la section 4.2.4).

Montrons alors que $L = h(L(G') \cap D_{n+m})$ en prouvant la double inclusion. L'inclusion $L \subset (h(L(G') \cap D_{n+m})$ se déduit du résultat suivant :

Proposition 10.2. *$\forall A \in V$, si $A \Rightarrow^\star_G u$, alors $\exists v$ tel que (i) $v \in D_{n+m}$, (ii) $A \Rightarrow^\star_{G'} v$ et (iii) $u = h(v)$*

Si $A \Rightarrow^\star_G u$ en une étape de dérivation, on doit avoir $u = a$, $A \to a$; or aa' est dans $D_{n+m} \cap L(G')$, puisque la grammaire de Dyck contient $S \to aSa', S \to \varepsilon$, et G' contient par construction la production $A \to aa'$. Comme de plus on a $h(aa') = a$, on conclut que tout mot de longueur 1 de G vérifie la propriété précédente. Supposons que ce résultat soit vrai jusqu'à la longueur k et considérons u dérivant de A_i dans G par une série de $k+1$ étapes. La première de ces étapes est alors nécessairement de la forme $A_i \to B_iC_i$, et il existe un entier j tel que $B_i \stackrel{\star}{\Rightarrow} u_1 \dots u_j$ et $C_i \stackrel{\star}{\Rightarrow} u_{j+1} \dots u_k$. L'hypothèse de récurrence nous assure alors de l'existence de deux mots v et w tels que :
– $v \in D_{n+m}, B_i \Rightarrow^\star_{G'} v, u_1 \dots u_j = h(v)$
– $w \in D_{n+m}, C_i \Rightarrow^\star_{G'} w, u_{j+1} \dots u_k = h(w)$
La dernière étape de la dérivation (gauche) de B_i dans G' est de la forme $A \to aa'$; or, la production $A \to aa'a'_{n+i}C_i$ est par construction également dans G' pour toutes les valeurs de i, ce qui implique $B_i \Rightarrow^\star_{G'} va'_{n+i}C_i$.

Puisque $A_i \to a_{n+i}B_i$ est également une production de G', on déduit alors que $A_i \Rightarrow_{G'} a_{n+i}B_i \Rightarrow^\star_{G'} a_{n+i}va'_{n+i}C_i \Rightarrow^\star_{G'} a_{n+i}va'_{n+i}w$, dont l'image par h est précisément u. On note enfin que, comme v et w sont dans D_{n+m}, par application de la propriété 10.1(iii) des langages de Dyck, $a_{n+i}va'_{n+i}w$ est également dans D_{n+m}, ce qui permet d'aboutir au résultat souhaité.

La conclusion s'impose alors : si $u \in L(G)$, alors $S \Rightarrow^\star_G u$ et donc il existe v dans $h(L(G') \cup D_{m+n})$ tel que $u = h(v)$.

La réciproque se fonde sur le résultat suivant, que nous allons également démontrer par induction.

Proposition 10.3. *$\forall A \in V$, si $A \Rightarrow^\star_{G'} v$, avec $v \in D_{m+n}$, alors $A \Rightarrow^\star_G h(v)$.*

Lorsque la dérivation a lieu en une étape, ce résultat est immédiat : on doit nécessairement utiliser dans G' une production de la forme $A \to aa'$, qui a une contrepartie $A \to a$ dans G. Supposons donc

que la propriété soit vraie pour toutes les dérivations de longueur au moins k et considérons une dérivation de longueur $k+1 : A \Rightarrow^{\star}_{G'} \quad v$. La première étape de cette dérivation ne peut être de la forme $A \rightarrow aa'$ (car il y a plus d'une étape de dérivation), ni de la forme $A \rightarrow aa'a'_{n+j}C_j$ (car on aurait alors une proto-phrase dans laquelle a'_{n+j} apparaîtrait devant a_{n+j}, ce qui est impossible). La seule possibilité restante est donc que la dérivation de v débute par : $A \rightarrow a_{n+j}B_j$ pour j entre 1 et m. v se factorise donc sous la forme $v = a_{n+j}v'$, et v' contient nécessairement a'_{n+j}, soit encore $v = a_{n+j}v_1a'_{n+j}v_2$. v étant dans D_{n+m}, il est de surcroît assuré que v_1 et v_2 sont également dans D_{n+m}. La définition de G' fait que l'introduction de a'_{n+j} n'est possible que parallèlement à celle de C_j et que la dérivation de v est de la forme $A \stackrel{\star}{\Rightarrow} \quad a_{n+j}v_1a'_{n+j}C_j \stackrel{\star}{\Rightarrow} \quad a_{n+j}v_1a'_{n+j}v_2$ (rappelons en effet que G' est linéaire à droite : le seul non-terminal d'un proto-mot est toujours le plus à droite). Remarquons maintenant que, pour chaque production de la forme $A \rightarrow aa'a'_jC_j$, la production $A \rightarrow aa'$ est également dans G. Cela signifie qu'au lieu d'appliquer la production introduisant C_j, on aurait tout aussi bien pu achever la dérivation de v_1 depuis B_j . On a donc finalement montré :

– $B_j \Rightarrow^{\star}_{G'} \quad v_1$, avec v_1 dans D_{n+m}

– $C_j \Rightarrow^{\star}_{G'} \quad v_2$, avec v_2 dans D_{n+m}

Par application de l'hypothèse inductive, il vient alors que $B_j \Rightarrow^{\star}_{G} \quad h(v_1)$, $C_j \Rightarrow^{\star}_{G} \quad h(v_2)$, et donc que $A \rightarrow B_jC_j \Rightarrow^{\star}_{G} \quad h(v_1)h(v_2) = h(a_{n+j}v_1a'_{n+j}v_2) = h(v)$, soit précisément ce qu'il fallait démontrer.

Nous introduisons maintenant un second résultat fondamental, qui donne une autre caractérisation des langages hors-contexte à partir des langages de Dyck. L'intuition principale de ce résultat est que dans la dérivation d'un mot dans une grammaire hors contexte, toute variable introduite (en partie droite) doit être "éliminée" par une variable (appariée) qui figure en partie gauche d'une règle.

Théorème 10.6 (Shamir). *Soit L un langage hors-contexte sur $\Sigma^{\star}$, alors il existe un alphabet de symboles appariés Σ', un symbole X de Σ' et un morphisme h de Σ dans l'ensemble des langages sur $\Sigma'^{\star}$ tel que $u \in L$ si et seulement si $X\phi(u) \cap D \neq \emptyset$, avec D le langage de Dyck sur Σ'.*

Preuve : Soit L un langage CF, et $G = (\Sigma, V, S, P)$ une grammaire sous forme normale de Greibach engendrant L. Sans perte de généralité, on suppose que chaque production de G est de la forme $A \rightarrow a\alpha$, où α ne contient que des symboles non-terminaux. À chaque variable X de V, on associe un conjugué X', et l'on note Σ' l'ensemble des variables et de leurs conjugués. Rappelons que α^R dénote le miroir du mot α.

Nous introduisons alors le morphisme h, dit morphisme de Shamir, défini inductivement sur $\Sigma^{\star}$ par :

$$\begin{cases} h(\varepsilon) = \varepsilon \\ \forall a \in \Sigma, h(a) = \{X'\alpha^R \mid X \rightarrow a\alpha \in P\} \\ \forall u \in \Sigma^{\star} \text{ tel que } |u| > 1, u = av \text{ et } h(u) = h(a)h(v) \end{cases}$$

La démonstration du théorème précédent repose sur la proposition suivante :

Proposition 10.4. *$A \Rightarrow^{\star}_{G} \quad u\alpha$, avec α dans $V^{\star}$ si et seulement si $X\phi(u)$ contient un mot réductible (au sens de l'élimination des conjugués) en α^R.*

Preuve : Comme précédemment, nous allons montrer la double implication par induction sur la longueur des dérivations. Pour une dérivation de longueur 1, $A \rightarrow a\alpha$ dans G implique que $X'\alpha^R$ est dans $\phi(a)$, et donc que $X\phi(a)$ contient α^R, qui est effectivement réductible à lui-même.

Supposons la propriété vraie pour des dérivations de longueur au plus k, et considérons la déri-

vation suivante :

$$\begin{array}{lll} X & \stackrel{\star}{\Rightarrow}_G & u_1 \dots u_k A \beta \\ & & u_1 \dots u_k u_{k+1} \alpha \beta \end{array}$$

L'hypothèse de récurrence garantit qu'il existe un élément z de $\Sigma'^{\star}$ qui est dans $X\phi(u_1 \dots u_l)$ et réductible à $\beta^R A$. Puisque $A'\alpha^R$ est dans $\phi(u_{k+1})$, $zA'\alpha^R$ est dans $X\phi(u_1 \dots u_k)\phi(u_{k+1}) = X\phi(u_1 \dots u_{k+1})$, et réductible en $\beta^R\alpha^R$, donc en $(\alpha\beta)^R$, ce qu'il fallait démontrer.

La réciproque se montre de manière similaire et est laissée en exercice.

On déduit finalement que $S \stackrel{\star}{\Rightarrow}_G u$ si et seulement si il existe un mot de $S\phi(u)$ qui soit réductible à ε, donc qui appartienne au langage de Dyck sur $\Sigma'^{\star}$.

Chapitre 11

Problèmes décidables et indécidables

Nous revenons, dans ce chapitre, sur les questions de décidabilité pour les langages hors-contextes, question qui ont déjà été évoquées à la section 5.3.3. Nous commençons pas introduire deux problèmes indécidables, le problème des mots et le problème de la correspondance de Post. Dans un second temps, nous utilisons ce dernier problème pour dériver une batterie de résultats concernant les grammaires et les langages hors-contexte.

11.1 L'expressivité calculatoire des systèmes de récriture

Dans cette section, nous montrons que les mécanismes de récriture définis par des grammaires formelles ont une expressivité comparable aux machines de Turing, à travers une étude du problème des mots. Nous introduisons ensuite un second problème indécidable, celui de la correspondance de Post.

11.1.1 Le problème des mots

Le problème des mots pour les grammaires formelles est défini de la manière suivante.

Définition 11.1 (Problème des mots). *Le* problème des mots *consiste à décider, pour une grammaire G donnée, et u, v dans $\Sigma^\star$, si $u \Rightarrow_G^\star v$.*

En fait, si une telle dérivation existe, alors il est possible de la trouver, en parcourant de manière systématique par énumération le graphe des dérivations finie de u par G. Le point difficile est de détecter qu'une telle dérivation n'existe en fait pas et qu'il est vain de la chercher. On retrouve là une situation similaire à celle rencontrée lors de l'examen du problème de l'arrêt pour les machines de Turing : la difficulté est d'identifier les configurations dans lesquelles la machine de s'arrête pas.

La preuve de l'indécidabilité du problème des mots repose sur la réduction suivante du problème à un problème de mots, correspondant à une ré-expression sous la forme de récritures du comportement d'une machine de Turing. Soit MT une machine de Turing sur l'alphabet $\Sigma \cup \{\#\}$, ce dernier symbole dénotant le caractère "blanc" et Q l'ensemble des états de la machine. Deux états de Q sont distingués : q_I l'unique état initial et q_F un unique état final. Les actions de MT sont décrites par des déplacements L (une case vers la gauche) et R (une case vers la droite) et I (rester sur la

case courante) et des actions de lecture/écriture $y : z$, exprimées par la fonction de transition δ de $(Q \times \Sigma)$ dans $(Q \times \Sigma \times \{L, R, I\})$.

On note (α, q, x, β) une configuration instantanée de MT, décrivant une situation où la tête de lecture est positionnée sur un x, encadrée à sa gauche par la séquence α et à sa droite par la séquence β ; le contrôle est dans l'état q.

Cette situation est représentée linéairement par la séquence $\$\alpha q x \beta\$$ sur l'alphabet $Q \cup \Sigma \cup \{\$\}$, avec $\$$ un nouveau symbole servant de frontière. Ces actions sont converties en règles de récriture d'une grammaire G selon :
- $qxy \to xpz$ pour toute action $\delta(q, x) = (p, R, y : z)$: cette action transforme le ruban $\alpha xy\beta$ en $\alpha xz\beta$, met la machine dans l''état p et déplace la tête de lecture vers la droite, sur le z qui vient d'être récrit. La nouvelle configuration est donc $\$\alpha xpz\beta\$$.
- $yq \to pz$ pour toute action $\delta(q, x) = (p, L, y : z)$
- $qx\$ \to qx\#\$$ pour tout q et tout x dans Σ : cette production rajoute simplement un symbole # (blanc) à droite de la case courante.
- $q_F x \to q_F$ pour tout x : efface tout symbole du côté droit du ruban
- $xq_F \to q_F$ pour tout x : efface tout symbole du côté gauche du ruban

Cette construction assure que toute séquence de mouvements de M se traduit de manière équivalente par des récritures dans G. En particulier, si M s'arrête avec un ruban entièrement blanc sur une entrée u, alors il existe une dérivation dans G récrivant $\$uq_I\#\$$ en $\$q_F\$$. Savoir résoudre le problème des mots dans G donnerait alors la solution du problème de l'arrêt d'une machine de Turing, qui est indécidable. Le problème des mots est indécidable.

11.1.2 Le problème de Post

Le problème de la *correspondance de Post* est un problème de combinatoire sur les mots dont on admettra ici l'indécidabilité. Ce problème est réputé comme étant "la mère" de tous les problèmes de décidabilité pour les langages formels et est utilisé dans de nombreuses démonstrations.

Formellement, un système de Post est défini de la façon suivante :

Définition 11.2. *Étant donné un alphabet Σ, un système de Post est tout simplement un ensemble de n couples de mots (u_i, v_i), pour i variant de 1 à n.*

Une solution au problème de la correspondance est une séquence d'indices $i_1 \dots i_k$ (le même indice pouvant apparaître plusieurs fois) telle que :

$$u_{i_1} \dots u_{i_k} = v_{i_1} \dots v_{i_k}$$

L'analogie la plus simple pour saisir le problème est celle des dominos : chaque couple (u_i, v_i) représente une tuile de domino de la forme :

u_i
v_i

Étant donné un ensemble de dominos, la question est alors de savoir s'il est possible de les concaténer de telle manière que le mot "du haut" soit égal au mot "du bas" ?

Attention : la question n'est pas simplement de savoir s'il existe un certain arrangement des u_i et un arrangement des v_i qui conduisent à deux mots identiques[1] ; mais bien de trouver deux arrangements qui utilisent les parties hautes et basses des "tuiles" disponibles.

[1] Sauriez-vous prouver que ce problème-là est facile, c'est-à-dire ici, décidable ?

Considérons par exemple le système défini par : $\{(a, ab), (ba, aba), (b, aba), (bba, b)\}$. Ce système admet une solution, représentée dans la Table 11.1.

a	bba	ba
ab	b	aba

TAB. 11.1 – Une solution au problème de la correspondance de Post

Comment trouver une telle solution ? Un raisonnement simple nous permet de dériver des conditions nécessaires. Ainsi, la première tuile doit nécessairement être telle que soit u_i est un préfixe de v_i, soit v_i est un préfixe de u_i. Une condition similaire, portant sur les suffixes, se déduit pour la dernière tuile. Cette condition n'étant respectée que pour deux des quatre tuiles (a, ab) et (bba, b), toute solution doit débuter par l'une de ces deux tuiles, et s'achever avec (ba, aba), qui est la seule tuile respectant la condition sur les suffixes.

Ce problème, en apparence simple, est en réalité insoluble par un algorithme général :

Théorème 11.1. *Le problème de la correspondance de Post est indécidable.*

Ce résultat classique se démontre à partir de la non-décidabilité du problème dit "Problème du mot" dans un système de Thue.

11.2 Quelques problèmes pour les CFG

Nous présentons dans cette section les résultats classiques d'indécidabilité pour les langages hors-contexte, en suivant pour l'essentiel la présentation donnée dans (Denning et al., 1978).

11.2.1 L' intersection

La clé des démonstrations qui suivent repose sur une transformation d'un système de Post en un couple de grammaires. Comme pour les transformations utilisées en théorie de la complexité[2], il s'agit de transformer tout problème de Post en un problème portant sur les CFG, de telle manière que le problème original ait une solution si et seulement si le problème dérivé en a une.

La transformation qui nous intéresse est la suivante : soit $S = \{(u_i, v_i), i = 1 \dots n\}$ un système de Post défini sur un alphabet Σ, considérons alors les grammaires $G_t = (\Sigma \cup \{1 \dots n\}, \{S_t\}, S_t, P_t)$ (resp. $G_b = (\Sigma \cup \{1 \dots n\}, \{S_b\}, S_b, P_b)$), où les productions de G_t et G_b sont définies comme suit :
– $S_t \rightarrow \alpha$ si et seulement si : soit $\alpha = iS_tu_i$, soit $\alpha = iu_i$
– $S_b \rightarrow \alpha$ si et seulement si : soit $\alpha = iS_bv_i$, soit $\alpha = iv_i$
G_t produit alors tous les mots susceptibles d'apparaître sur la ligne "du haut" lorsque l'on concatène des tuiles dans un système de Post ; G_b se chargeant de la ligne "du bas". Mais ce n'est pas tout : lors d'une dérivation, on "empile" à gauche de S les indices des tuiles qui sont utilisées, pour produire des mots formés d'un côté par des indices, de l'autre par des symboles de Σ. À titre d'illustration, reprenons le système défini à la section précédente, et construisons la grammaire G_t correspondante, dont les productions sont listées dans la Table 11.2.

[2] À la différence près que pour des démonstrations de décidabilité, il n'est pas besoin de prendre en compte la complexité algorithmique de la transformation.

$$
\begin{array}{ll}
S_t \rightarrow & 1S_t a \mid 1a \\
S_t \rightarrow & 2S_t ba \mid 2ba \\
S_t \rightarrow & 3S_t b \mid 3b \\
S_t \rightarrow & 4S_t bba \mid 4bba
\end{array}
$$

TAB. 11.2 – La grammaire "top" d'un système de Post

Une dérivation gauche dans cette cette grammaire est par exemple $S \Rightarrow 1S_t a \Rightarrow 14S_t bbaa \Rightarrow 142babbaa$, engendrant non seulement le mot *babbaa*, mais également la liste (inversée) des tuiles utilisées : la tuile numéro 1, puis la 4, puis la 2.

Un second point important pour la suite est que $L(G_t)$ et $L(G_b)$ sont deux langages *déterministes* : pour s'en convaincre, il suffit de réaliser qu'un automate à pile les reconnaissant aurait un fonctionnement très simple, consistant à empiler un symbole Z_i à chaque lecture d'un indice i, puis, rencontrant un premier symbole de Σ à dépiler Z_i conditionnellement à la lecture de u_i.

L'intérêt de cette construction repose dans l'équivalence suivante : le problème de Post a une solution si et seulement si $L(G_t)$ et $L(G_b)$ ont une intersection non vide. La preuve est la suivante : si le système de Post à une solution $i_1 \dots i_k$, conduisant au mot w, alors l'application dans G_t (resp. G_b) des $k-1$ règles $S_t \rightarrow i_j S_t u_{i_j}$ (resp. $S_b \rightarrow i_j S_b v_{i_j}$) pour j variant de k à 2, suivie de $S_t \rightarrow i_1 u_{i_1}$ (resp. $S_b \rightarrow i_1 v_{i_1}$) dérive le mot $i_k i_{k-1} \dots i_1 w$ dans $L(G_t)$ comme dans $L(G_b)$. Inversement, si l'intersection est non-vide et contient le mot $i_k \dots i_1 w$, alors il est immédiat de vérifier que $i_1 \dots i_k$ est une solution du système de Post. S'en déduit le théorème suivant :

Théorème 11.2. *Il n'existe pas d'algorithme décidant si l'intersection des langages de deux grammaires hors-contexte est vide.*

Si un tel algorithme existait, il est en effet clair qu'il pourrait directement être mis à profit pour résoudre le problème de la correspondance de Post, qui est un problème indécidable. Un tel algorithme ne peut donc exister, le problème de l'intersection de deux CFG est donc indécidable.

Proposition 11.1. *Il n'existe pas d'algorithme décidant si l'intersection de deux grammaires hors-contexte contient un nombre infini d'éléments.*

Il suffit de remarquer qu'il existe une solution au problème de Post exactement dans les cas où l'intersection précédente est infinie. Si, en effet, il existe une solution w, il en existe nécessairement une infinité, puisque toute itération w^+ de la solution définit une nouvelle solution. La réciproque étant triviale, on obtient le résultat énoncé ci-dessus.

11.2.2 Une rafale de problèmes indécidables

Complément et Universel Les démonstrations qui suivent reposent principalement sur le déterminisme des langages $L(G_t)$ et $L(G_b)$ et sur la stabilité des langages déterministes par complémentation (cf. la section 9.3). Ceci permet en particulier d'affirmer que $\overline{L(G_t)}$ et $\overline{L(G_b)}$ sont deux langages hors-contexte déterministes, dont il est possible de construire la grammaire à partir de celles de G_t et G_b. Leur union, $\overline{L(G_t)} \cup \overline{L(G_b)}$, est également un langage hors-contexte (pas nécessairement déterministe). On déduit :

Théorème 11.3. *Soit G une grammaire hors-contexte, le problème $\overline{L(G)} = \emptyset$ est indécidable.*

C'est une conséquence directe de la loi de De Morgan : $\overline{A \cup B} = \overline{A} \cap \overline{B}$. Si l'on savait décider le problème précédent, il suffirait de considérer la question $\overline{L(G)} = \emptyset$ pour G une grammaire CF pour $\overline{L(G_t)} \cup \overline{L(G_b)}$ pour résoudre le problème de l'intersection, puis le problème de Post. Comme corollaire, on note au passage que le problème : $L(G) = \Sigma^\star$ est également indécidable.

Rationnel Montrons maintenant que le problème de la rationalité de $L(G)$ également indécidable. Considérons, de nouveau, le langage intersection $L(G_t) \cap L(G_b)$: nous savons déjà que ce langage est vide si et seulement s'il existe pas de solution au problème de la correspondance de Post associé. Remarquons alors que, dans ce cas, ce langage est trivialement rationnel. Lorsqu'au contraire le problème de Post a une solution, alors nous avons vu qu'il en avait une infinité, consistant à des itérations w^i d'une solution de base w. Ces familles de solution donnent lieu à des mots de la forme $x^i w^i$ dans $L(G_t) \cap L(G_b)$, entraînant que cet ensemble ne peut être rationnel (par une application directe du lemme d'itération pour les langages rationnels). Il s'ensuit que :

Lemme 11.1. *$L(G_t) \cap L(G_b)$ est rationnel si et seulement si le problème de Post associé n'a pas de solution.*

Il s'ensuit immédiatement que si on savait décider la rationalité de $L(G)$ pour G hors-contexte, on saurait en particulier le faire pour la grammaire engendrant $\overline{L(G_t)} \cup \overline{L(G_b)}$, puis, du fait de la stabilité par complément des langages rationnels, pour $L(G_t) \cap L(G_b)$. Ceci impliquerait alors la décidabilité du problème de la correspondance de Post, d'où :

Théorème 11.4. *Soit G une grammaire hors-contexte, le problème "$L(G)$ est-il rationnel ?" est indécidable.*

Déterministe La question du déterminisme demande d'introduire un nouveau résultat, non démontré ici :

Théorème 11.5. *Soit G une grammaire d'un langage CF déterministe, alors le problème "$L(G)$ est rationnel" est décidable.*

On déduit immédiatement que la question du déterminisme du langage engendré par G est, elle, indécidable : sinon, il suffirait de décider dans un premier temps du déterminisme de $L(G)$ puis de :
– conclure à la non rationalité si $L(G)$ n'est pas déterministe
– utiliser le résultat précédent si $L(G)$ est déterministe
En conséquence :

Théorème 11.6. *Soit G une grammaire d'un langage CF, alors le problème "$L(G)$ est déterministe ?" est indécidable.*

Ambiguïté L'ambiguïté d'une grammaire est également une question indécidable. Ce résultat dérive de la construction canonique d'une grammaire G pour $L(G_t) \cup L(G_b)$ consistant à ajouter aux grammaires G_t et G_b un nouvel axiome S, ainsi que deux nouvelles productions $S \to S_t$, $S \to S_b$. G_t et G_b étant déterministes, elles ne peuvent donner lieu à aucune ambiguïté. G est donc ambiguë si et seulement si un même mot se dérive à la fois dans G_b et dans G_t, donc si et seulement si le problème de Post a une solution.

Théorème 11.7. *Soit G une grammaire CF, alors le problème "G est ambiguë" est indécidable.*

Des résultats complémentaires sont donnés, par exemple, dans (Hopcroft and Ullman, 1979, p. 217 et suivantes).

Chapitre 12

Parsage tabulaire

Nous nous étudions dans ce chapitre des stratégies d'analyse dédiées à l'analyse de langages ambigus, tels que ceux qui sont couramment utilisés pour décrire des énoncés en langage naturel. Rappelons qu'un langage CF ambigu est un langage tel que toute grammaire CF qui le représente est ambiguë, c'est-à-dire assigne plus plus d'un arbre de dérivation à au moins un mot de L. Dans ce contexte, les stratégies déterministes, garantissant une complexité linéaire, que nous avons présentées dans les chapitres précédents (notamment aux chapitres 7 et 9) sont nécessairement inadaptées. Le problème est double : (i) comment parvenir à conserver une complexité raisonnable pour tester l'appartenance d'une chaîne à un langage (ii) comment représenter efficacement l'ensemble (potentiellement exponentiellement grand) des différentes analyses pour une phrase donnée. Nous expliquons tout d'abord comment représenter dans une structure efficace, *la table des sous-chaînes bien formées*, un ensemble exponentiel d'arbres d'analyse. Nous présentons ensuite des algorithmes ayant une complexité polynomiale (précisément en $O(n^3)$, où n est la longueur de la chaîne à analyser) pour résoudre ce problème, tels que les algorithmes CYK (Younger, 1967) et Earley (Earley, 1970), qui sont présentés aux sections 12.1 et 12.2, ainsi que diverses variantes et extensions.

12.1 Analyser des langages ambigus avec CYK

12.1.1 Les grammaires pour le langage naturel

Nous avons vu (notamment au chapitre 6) que le parsage naïf de grammaires CF impliquait de calculer et de recalculer en permanence les mêmes constituants. Il apparaît donc naturel, pour accélérer la procédure, d'utiliser une structure temporaire pour garder les résultats déjà accumulés. Il existe, du point de vue du traitement automatique des langues , deux raisons importantes supplémentaires qui justifient le besoin de manipuler une telle structure. La première, et probablement la plus importante, est que le langage naturel est abominablement ambigu, et qu'en conséquence, les grammaires naturelles seront naturellement ambiguës, c'est-à-dire susceptibles d'assigner plusieurs structures syntaxiques différentes à un même énoncé. En sus de l'ambiguïté lexicale, massive, l'exemple le plus typique et potentiellement générateur d'une explosion combinatoire des analyses est celui du rattachement des groupes prépositionnels.

Considérons de nouveau la grammaire "des dimanches", reproduite à la Table 12.1, et intéressons-nous par exemple à la phrase : *Louis parle à la fille de la fille de sa tante*. Selon la manière dont on analyse le "rattachement" des groupes prépositionnels (correspondant au non-terminal *GNP*), on

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

TAB. 12.1 – La grammaire G_D des repas dominicaux

dispose pour cette phrase d'au moins trois analyses :
– un seul complément rattaché au verbe *parle* : *à la fille de la cousine de sa tante*
– deux compléments rattachés au verbe *parle* : *à la fille* ; *de la cousine de sa tante*
– deux compléments rattachés au verbe *parle* : *à la fille de la cousine* ; *de sa tante*
Notez qu'il y en aurait bien d'autres si l'on augmentait la grammaire, correspondant à la possibilité d'utiliser des groupes prépositionnels comme compléments circonstanciels (temps, lieu, moyen...), ainsi : *Ma sœur parle à la table d'à côté de la fille de la cousine de sa tante*.

Linguistiquement, la raison de la prolifération de ce type d'ambiguïté en français provient de la structure des groupes nominaux et noms composés qui sont majoritairement de la forme *N PP* (*DET*) *N*, éventuellement entrelardés ici où là de quelques adjectifs. Ceci est particulièrement manifeste dans les domaines techniques, où l'on parle de *termes* : *une machine de Turing*, *une turbine à vapeur*, *un réseau de neurones*, *un projet de développement d'un algorithme de résolution du problème de l'approximation d'équations aux dérivées partielles*....

Nous n'avons mentionné ici que des cas d'ambiguïtés globales, c'est-à-dire qui persistent au niveau de la phrase toute entière. Un analyseur perd également beaucoup de temps à explorer inutilement des ambiguïtés *locales*. Considérez par exemple la construction d'un nœud *S* dans une analyse ascendante de *l'élève de Jacques a réussi le contrôle*. Dans cet exemple, un analyseur naïf risque de construire toute la structure de racine *S* dominant le préfixe : *Jacques a réussi le contrôle*, qui, bien que correspondant à une phrase complète parfaitement correcte, laisse non analysé le préfixe *l'élève de*, et n'est donc d'aucune utilité.

Il est alors nécessaire, pour des applications en traitement des langues, de représenter de manière compacte un ensemble d'arbres de dérivations : on appelle un tel ensemble d'arbres une *forêt de dérivations*.

Second desiderata important : les systèmes de parsage envisagés jusqu'alors fournissent une réponse binaire (oui/non) à la question de l'appartenance d'une chaîne à un langage. Dans le cas où la réponse est négative, cela donne bien peu d'information aux traitements ultérieurs, quand bien même un grand nombre de constituants (incomplets) ont été formés pendant le parsage. Dans la mesure où les systèmes de TLN sont souvent confrontés à des énoncés non-grammaticaux au

sens strict (style de rédaction (eg. les titres) ; systèmes de dialogues : hésitations, reprises... ; sorties de systèmes de reconnaissance des formes (de la parole ou de l'écriture manuscrite) , cette manière de procéder est insatisfaisante : faute d'une analyse complète, on aimerait pouvoir produire au moins une analyse partielle de l'énoncé, qui pourra servir aux niveaux de traitement ultérieurs (par exemple l'analyseur sémantique).

La solution à tous ces problèmes consiste à utiliser une table des *sous-chaînes bien formées*, qui va d'une part mémoriser les structures partielles dès qu'elles ont été créées, afin d'éviter leur recalcul, et qui va, d'autre part, fournir des éléments d'informations aux niveaux de traitement supérieurs, même en cas d'échec de l'analyseur.

12.1.2 Table des sous-chaînes bien formées

La structure la plus généralement utilisée pour représenter des analyses partielles ou multiples est une table à double entrée : le premier indice, correspondant à la position de début du constituant, varie entre 0 et n ; le second indice correspond à la position du premier mot non couvert et varie entre 0 et $n+1$. Chaque cellule $T[i, j]$ de cette table contient la liste des constituants reconnus entre i et j, i inclus, j exclus.

En supposant que la phrase à analyser soit : *ma sœur mange*, on aurait la table des sous-chaînes bien formée suivante (voir la Table 12.2).

4	S		V, GV
3	GN	N	
2	DET		
	ma	sœur	mange
	1	2	3

TAB. 12.2 – Table des sous-chaînes bien formées

La cellule [1, 3] contient un GN, indiquant que l'on a reconnu un groupe nominal entre les positions 1 et 3 ; à l'inverse, la cellule [2, 4] est vide : aucun constituant de longueur 2 ne commence à cette position.

Une manière alternative d'implanter cette table consiste à la représenter sous la forme d'un graphe orienté sans cycle (ou DAG pour *Directed Acyclic Graph*). Chaque nœud de ce graphe représente un mot (ou un indice de mot), et chaque transition entre nœuds est étiquetée par le constituant trouvé entre les deux nœuds. Plusieurs transitions entre deux nœuds représenteront alors des fragments ambigus, c'est-à-dire qui peuvent être analysés de plusieurs manières différentes.

Ce type de représentation de l'ambiguïté est illustrée par l'exemple de l'énoncé : *La fille parle à sa mère de sa tante*, dont la table des sous-chaînes bien formées est partiellement[1] représentée à la Figure 12.1. Nous avons volontairement introduit sur ce schéma une information supplémentaire, qui n'est pas représentée dans la table, en étiquetant chaque arc avec la production qui l'induit, alors la table n'enregistre en réalité que l'identité du non-terminal correspondant.

Comme figuré sur ce graphe, il existe deux manières d'analyser *à sa mère de sa tante* : soit comme un *GNP*, soit comme une succession de deux *GNP*s. Ces deux interprétations sont effectivement représentées dans la table, qui, pourtant, ne contient qu'un seul arc étiqueté GV entre les noeuds 3 et 10.

[1] Il manque donc des arcs : sauriez-vous dire lesquels ?

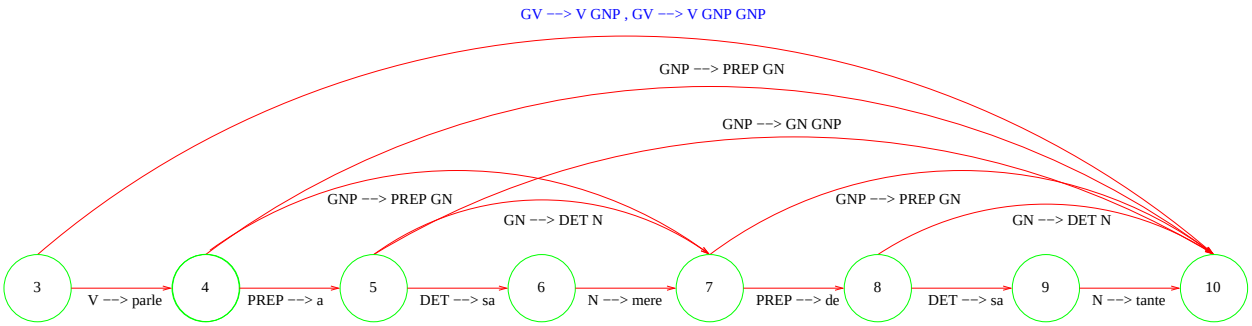


FIG. 12.1 – Représentation graphique d'une ambiguïté syntaxique

Dans cette représentation, le succès du parsage correspond à l'écriture dans la case "en haut à gauche" ($T[1,n]$) du tableau d'un constituant de type S (version tabulaire), ou encore d'un arc étiqueté S "couvrant" toute la phrase (version graphique).

Ces tables peuvent s'utiliser indépendamment de la stratégie de recherche ou de parsage utilisée, ascendante, descendante, mixte, dans la mesure où elles ne servent qu'à stocker de l'information. La seule contrainte d'utilisation est relative à l'ordre dans lequel le remplissage aura lieu. Ainsi, pour effectuer un parcours complet de l'espace de recherche, il faudra, dans une analyse descendante, vérifier que l'on introduit une nouvelle entrée dans la table qu'au moment où l'on connaît tous les constituants qui peuvent entrer dans cette case. Dans le cas contraire, on risque d'utiliser par anticipation, lors de la consultation de la table, un résultat partiel, et oublier des analyses. Le même type de vérification est nécessaire lorsque l'on adopte une stratégie de parsage ascendant.

12.1.3 L'algorithme CYK

L'algorithme CYK (Younger, 1967), du à Cocke, Younger et Kasami est destiné au parsage de grammaires sous forme normale de Chomsky (voir la section 8.2.1). Il fournit un premier exemple de d'utilisation de ces structures de table. Cet algorithme[2]implémente une stratégie d'analyse strictement ascendante. La mise sous CNF garantit en particulier, qu'on ne rencontrera ni production vide, ni chaîne de règles, deux configurations potentiellement problématiques pour des stratégies purement ascendantes. Commençons par donner une première version de cet analyseur, formalisé par l'algorithme 10.

La première étape de cet algorithme consiste à initialiser le tableau $T[]$ en insérant toutes les productions de type $A \rightarrow a$ potentiellement utilisées dans l'entrée u. Cette étape correspond au remplissage de la "diagonale" du tableau" ; elle est simplifiée par le fait que dans une grammaire sous forme normale de Chomsky, les seules règles introduisant des terminaux sont de la forme $A \rightarrow a$. La boucle principale consiste à construire, pour des tailles croissantes de l, les constituants de longueur l débutant en position i : il suffit, pour cela, de considérer toutes les factorisations en deux parties de $u_i \dots u_{i+l}$ et de chercher celles qui sont couvertes par deux terminaux B et C, avec $A \rightarrow BC$ une production de la grammaire.

Notez que, même s'il existe deux règles $A \rightarrow BC$ et $A \rightarrow DE$ permettant d'insérer A dans la cellule $T[i,j]$, la cellule $T[i,j]$ ne contiendra qu'une seule occurrence de A. C'est cette "factorisation" qui permet de représenter un nombre potentiellement exponentiel d'analyses dans une table

[2]C'est plutôt d'un méta-algorithme qu'il faudrait en fait parler, car la dénomination CYK regroupe en fait des stratégies d'analyse variées : gauche droit, en ligne, non directionnel...

Algorithm 10 – Analyseur CYK pour une grammaire CNF $G = (\Sigma, V, S, P)$

```
// la phrase à analyser est u = u_1 ... u_n
// la table d'analyse T est initialement vide : T[i, j] = ∅
for j := 1 ... n do
    foreach A → u_i ∈ P do
        T[i, i + 1] := T[i, i + 1] ∪ {A}
    od
od
// Boucle principale : Cherche les constituants de longueur croissante
for l := 2 ... n do
    for i := 1 ... n − l + 1 do
        for k := i + 1 ... i + l − 1 do
            if B ∈ T[i, k] ∧ C ∈ T[k, i + l] ∧ A → BC ∈ P
                then T[i, k + l] := T[i, k + l] ∪ {A}
            fi
        od
    od
od
if S ∈ T[1, n] then return(true) else return(false) fi
```

polynomiale.

Une fois la table construite, il suffit de tester si la table contient S dans la cellule $T[1, n]$ pour décider si la séquence d'entrée appartient ou non au langage engendré par la grammaire.

La correction de l'algorithme CYK repose sur *l'invariant* suivant, qui est maintenu de manière ascendante :

Proposition 12.1. *$A \in T[i, j]$) si et seulement si $A \stackrel{\star}{\Rightarrow}_G u_i \dots u_{j-1}$.*

La preuve de cette proposition repose sur une récurrence simple sur la longueur des constituants $(j - i)$. L'étape d'initialisation garantit que l'invariant est vrai pour les constituants de longueur 1. Supposons qu'il soit vrai pour tout longueur $\leq l$. L'insertion dans la cellule $T[i, j]$ est déclenchée uniquement par les situations où l'on a à la fois :

– $B \in T[i, k]$, ce qui implique que $B \stackrel{\star}{\Rightarrow} u_i \dots u_{k-1}$
– $C \in T[k, j]$, ce qui implique que $C \stackrel{\star}{\Rightarrow} u_k \dots u_{j-1}$
– $A \to BC \in P$, ce qui entraîne bien que $A \stackrel{\star}{\Rightarrow} u_i \dots u_{j-1}$

Quelle est la complexité de l'algorithme CYK ? La boucle principale du programme comprend trois boucles **for** se répétant au pire n fois, d'où une complexité cubique en n. Chaque cellule de T contient au plus $|V|$ valeurs ; dans le pire des cas, il y aura autant de productions CNF que de paires de non-terminaux, d'où une complexité $|V|^2$ pour la recherche de la partie droite. Au total, l'algorithme CYK a donc une complexité totale : $|V|^2 n^3$. C'est un premier résultat important : dans le pire des cas, le test d'appartenance de u à une grammaire CF G a une complexité polynomiale en la longueur de u, bornée par un polynôme de degré 3. Notez que la complexité dépend, à travers la constante, de la taille de grammaire (après Chomsky normalisation).

Différentes variantes de CYK sont possibles, selon que l'on adopte un ordre de parcours qui est

soit « les plus courtes chaînes d'abord » (ce qui correspond à l'algorithme présenté ci-dessus), soit « par sous-diagonales ». Ces deux parcours sont représentés respectivement en trait pleins et pointillés dans la Figure 12.2. Cette table illustre le fait que, dans le premier parcours, la cellule $T[1,4]$ est traitée après la cellule $T[5,7]$), c'est-à-dire après l'examen de toutes les sous-chaînes de longueur de 2 ; alors que dans le second parcours, elle est examinée dès que deux sous-constituants sont disponibles. Cette seconde stratégie correspond à une analyse *en ligne*, pendant laquelle les constituants maximaux sont construits au fur et à mesure que les mots sont présentés à l'analyseur.

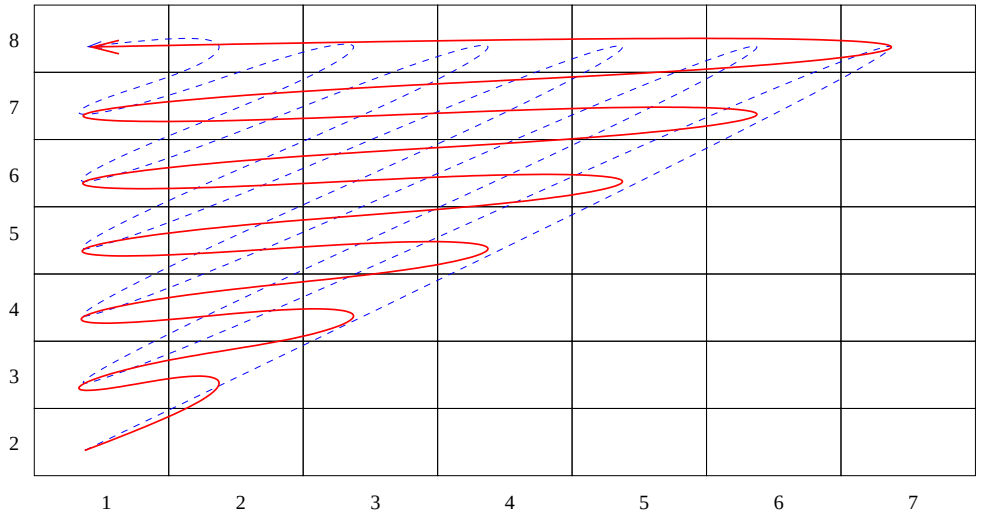


FIG. 12.2 – Deux implantations différentes de CYK

Pour compléter, notez que CYK est adaptable aux cas non-CNF et qu'il est possible, en analysant la version Chomsky-normalisée, de reconstruire (au prix d'une structure auxiliaire gérant la correspondance entre règles CNF et les autres) les analyses fournies par la grammaire originale.

12.1.4 Du test d'appartenance à l'analyse

Comment reconstruire à partir de la table T les différents arbres de dérivations ? Cette étape de reconstruction, implantée naïvement, demande deux extensions à l'algorithme précédent 10. Il s'agit tout d'abord de sauvegarder la manière dont un non-terminal est inséré dans une cellule, en associant à chaque non-terminal A de $T[i,j]$ un ensemble BP, contenant par exemple des quintuplets[3] (r,i,j,i',j'), où r est l'indice de la A-production appliquée, et (i,j) et (i',j') les indices dans la table des "descendants" de A.

Avec cette extension, la construction de l'arbre d'analyse s'opère en traversant depuis la racine le graphe de descendance induit par les BP. La procédure *MakeTrees* de l'algorithme 11 implémente un tel parcours.

[3] Si l'on y réfléchit un peu, des quadruplets suffisent. Pourquoi ?

Algorithm 11 – Construction des arbres de dérivation à partir d'une table CYK

```
Tree = ∅ // l'arbre est une pile contenant les dérivations gauches
MakeTrees(S, 1, n)
funct MakeTrees(A, p, l))
  if (l = 1)
    then do // une feuille
            foreach (A → a) ∈ BP(A, p, l) do
              push(Tree, A → a)
              if s = n then PrintTree(Tree) fi
            od
         od
    else do // un noeud interne
            foreach (A → BC, i, j, i', j') ∈ BP(A, l, s)
              MakeTrees(B, i, j)
              MakeTrees(C, i', j')
            od
         od
  fi
```

12.2 Algorithme d'Earley et ses variantes

12.2.1 Table active des sous-chaînes bien formées

Pour l'instant, nous avons considéré la table comme un simple auxiliaire pour mémoriser les constituants trouvés lors de la recherche et pour représenter des analyses partielles et/ou multiples. Nous allons maintenant montrer comment étendre cette représentation pour *diriger* l'analyse, en utilisant la table pour représenter des hypothèses concernant les constituants en cours d'agrégation (dans une stratégie ascendante) ou concernant les buts à atteindre (dans le cas d'une stratégie descendante). Cela nous permettra (i) de nous affranchir de la précondition de mise sous forme normale de Chomsky et (ii) de construire des analyseurs plus efficaces.

Un premier pas, par rapport à CYK, consiste à stocker dans la table d'analyse un peu plus que les simples terminaux reconnus entre deux positions, en utilisant de nouveau (cf. la section 7.2) le concept de *règle pointée* pour représenter les hypothèses et buts courants. Rappelons :

Définition 12.1 (Règle pointée). *Une* règle pointée *est une production augmentée d'un point. Le point indique l'état courant d'application de la règle (autrement dit à quel point les hypothèses qui permettent d'appliquer la règle ont été satisfaites).*

Par exemple, des règles pointées suivantes sont tout à fait licites compte-tenu de la Table 12.1 :
– $S \rightarrow \bullet\ GN\ GV$
– $S \rightarrow GN \bullet GV$
– $S \rightarrow GN\ GV \bullet$
Ces arcs s'insèrent naturellement dans la table des sous-chaînes bien formées, comme représenté à la Figure 12.3. Sur cette figure, la règle pointée $S \rightarrow GN \bullet GV$ entre les nœuds 1 et 3 indique que l'opération réussie de réduction du GN a conduit à reconnaître le début d'une phrase (S), à laquelle il manque encore un groupe verbal pour être complète.

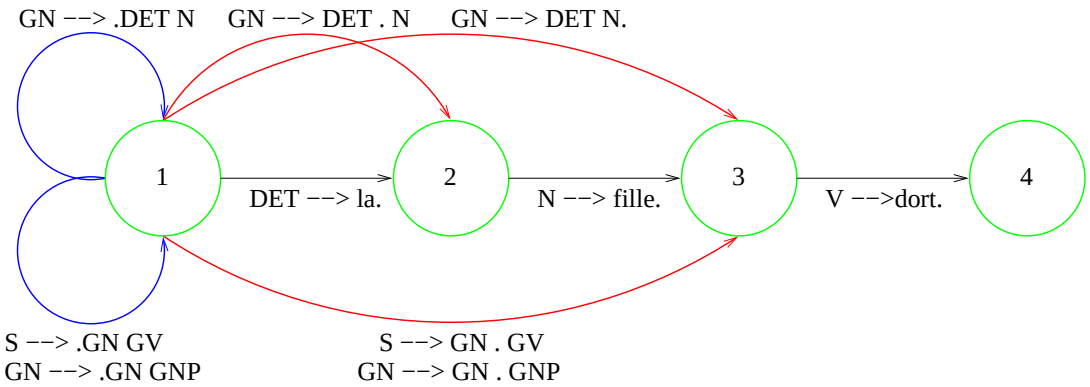


FIG. 12.3 – Un exemple de table active

Le graphe de la Figure 12.3 se transpose directement dans la représentation tabulaire suivante (voir la Table 12.3).

4	$S \rightarrow GN\ GV\bullet$		$V \rightarrow dort\bullet$
3	$S \rightarrow GN \bullet GV, GN \rightarrow GN \bullet GNP$	$N \rightarrow fille\bullet$	
2	$DET \rightarrow la\bullet, GN \rightarrow DET \bullet N$		
	1	2	3
	la	fille	dort

TAB. 12.3 – Une table d'analyse en construction

On appellera *item* tout élément figurant dans une cellule de la table d'analyse. Dans le cas présent, un item est de la forme $[A \rightarrow \alpha \bullet \beta, i, j]$, portant l'information suivante : $\alpha \Rightarrow^{\star}_{G} \quad u_i \ldots u_{j-1}$. Il existe en fait deux sortes d'items dans une table d'analyse :

Définition 12.2 (Items actifs et inactifs). *Un item actif est un item tel que le point n'est pas situé à droite de la partie droite. Un item inactif est un item qui n'est pas actif. Une table active est une table qui contient des items actifs.*

Dans l'exemple de la Table 12.3, $[S \rightarrow GN \bullet GV, 1, 3]$ est un item actif, alors que $[N \rightarrow fille\bullet, 2, 3]$ est inactif. Les items *inactifs* correspondent précisément à ce qui est représenté (et représentable) dans la table d'un analyseur de type CYK. Voyons comment les items actifs interviennent pour définir des stratégies de parsage plus efficaces, en commençant par une nouvelle stratégie purement ascendante.

12.2.2 La règle fondamentale du parsage tabulaire

L'idée générale du parsage à base de table, consiste à essayer d'étendre les items actifs pour les rendre inactifs, puisqu'une analyse complète est par définition représentée par un item inactif $[S \rightarrow \alpha\bullet, 1, n]$. Comment cette opération essentielle se déroule-t-elle ? Par application de la règle suivante :

Définition 12.3 (Règle fondamentale du parsage tabulaire). *Si la table d'analyse T contient l'item actif $[A \rightarrow \alpha_1 \bullet B\alpha_2, i, j]$ et l'item inactif $[B \rightarrow \alpha_3\bullet, j, k]$, alors on peut rajouter dans la table l'item (actif ou inactif) : $[A \rightarrow \alpha_1 B \bullet \alpha_2, i, k]$.*

Cette règle, par la suite dénotée *comp*, peut être vue comme une règle de déduction permettant de construire de nouveaux items en combinant des items déjà présents dans la table.

Un exemple d'application de cette règle fondamentale se lit dans l'exemple la Table 12.3 : ainsi la présence de l'item $[GN \rightarrow DET \bullet N, 1, 2]$ (un *GN* en cours de reconnaissance), et de l'item $[N \rightarrow fille\bullet, 2, 3]$ (un *N* complet) permet d'insérer dans la table le nouvel item $[GN \rightarrow DETN\bullet, 1, 3]$ (un *GN* complet, de longueur 2, débutant en position 1).

Pour achever de définir complètement un algorithme de parsage utilisant ces items, trois points supplémentaires doivent être spécifiés :
– l'initialisation de la table. En effet, la règle fondamentale ne pourra rien dériver tant que la table est vide : il faut donc décider d'une certaine manière initialiser la table.
– la stratégie d'utilisation des règles (ou comment insérer de nouveaux arcs actifs dans le graphe) : il faut également des arcs actifs pour appliquer la règle fondamentale ;
– la stratégie de recherche, qui correspond à la définition d'un ordre pour examiner efficacement les hypothèses actives.

Ces spécifications complémentaires doivent être penser dans l'optique permettre d'optimiser l'analyse, sachant que performances des algorithmes de parsage tabulaire dépendront principalement :
– du nombre d'items créés en cas d'échec de l'analyse : on souhaite qu'il y en ait le moins possible.
– du nombre d'items inutiles en cas de succès de l'analyse : idéalement tous les arcs du graphes doivent être utilisés dans au moins une analyse ; idéalement, il ne devrait plus y avoir d'arcs actifs quand l'analyse se termine.

Nous présentons, dans un premier temps, une instanciation particulière du parsage tabulaire dans le cadre d'une stratégie gauche droite purement ascendante. Nous présentons ensuite diverses variantes, plus efficaces, d'algorithmes utilisant la tabulation.

12.2.3 Parsage tabulaire ascendant

L'implémentation de cette stratégie se fait directement, en complétant la règle fondamentale par les instructions suivantes :
– *init* la règle fondamentale nécessite de disposer d'items actifs. La manière la plus simple pour en construire est d'insérer, pour chaque production $A \rightarrow \alpha$ de P, et pour chaque position dans la phrase, une hypothèse concernant le début de la reconnaissance de A. Comme ces nouveaux items ne "couvrent" aucun terminal de l'entrée courante, leur indice de début sera conventionnellement pris égal à l'indice de fin. Cette étape donne donc lieu à des items $[A \rightarrow \bullet\alpha, i, i]$ selon :
  foreach $(A \rightarrow \alpha) \in P$ **do**
    foreach $i \in (1 \ldots n)$ **do**
      insert$[A \rightarrow \bullet\alpha, i, i]$ in T
    od
  od
– *scan* il est également nécessaire de définir une stratégie pour concernant la reconnaissance des terminaux. La règle suivante exprime le fait que si un item actif "attend" un terminal, alors cet élément trouvé sur l'entrée courante (u) permettra le développement de l'item, soit formellement :
  if $[A \rightarrow \alpha \bullet a\beta, i, j] \in T \ \wedge \ u_j = a$
    then *insert*$[A \rightarrow \alpha a \bullet \beta, i, j+1]$ in T
  fi

La mise en œuvre de cette stratégie sur l'exemple de la phrase : *un père gronde sa fille* est illustrée à la Table 12.4. Cette table a été délibérément expurgée des (très nombreux) items créés à

l'initialisation, qui sont néanmoins utilisés pour dériver de nouveaux items.

Num.	Item	règle	Antécédents
1	$[DET \rightarrow un\bullet, 1, 2]$	*scan*	$[DET \rightarrow \bullet un, 1, 1]$
2	$[GN \rightarrow DET \bullet N, 1, 2]$	*comp*	$[GN \rightarrow \bullet DET\ N, 1, 1]$ et 1
3	$[N \rightarrow père\bullet, 2, 3]$	*scan*	$[N \rightarrow \bullet père, 2, 2]$
4	$[GN \rightarrow DET\ N\bullet, 1, 3]$	*comp*	2 et 3
5	$[S \rightarrow GN \bullet GV, 1, 3]$	*comp*	$[S \rightarrow \bullet GN\ GV, 1, 1]$ et 4
6	$[GN \rightarrow GN \bullet GNP, 1, 3]$	*comp*	$[GN \rightarrow \bullet GN\ GNP, 1, 1]$ et 4
7	$[V \rightarrow gronde\bullet, 3, 4]$	*scan*	$[V \rightarrow \bullet gronde, 3, 3]$
8	$[GV \rightarrow V\bullet, 3, 4]$	*comp*	$[GV \rightarrow \bullet V, 3, 3]$ et 7
9	$[GV \rightarrow V \bullet GN, 3, 4]$	*comp*	$[GV \rightarrow \bullet V\ GN, 3, 3]$ et 7
10	$[GV \rightarrow V \bullet GNP, 3, 4]$	*comp*	$[GV \rightarrow \bullet V\ GNP, 3, 3]$ et 7
11	$[GV \rightarrow V \bullet GN\ GNP, 3, 4]$	*comp*	$[GV \rightarrow \bullet V\ GN\ GNP, 3, 3]$ et 7
12	$[GV \rightarrow V \bullet GNP\ GNP, 3, 4]$	*comp*	$[GV \rightarrow \bullet V\ GNP\ GNP, 3, 3]$ et 7
13	$[S \rightarrow GNGV\bullet, 1, 4]$	*comp*	5 et 8
14	$[DET \rightarrow sa\bullet, 4, 5]$	*scan*	$[DET \rightarrow \bullet sa, 4, 4]$
15	$[GN \rightarrow DET \bullet N, 4, 5]$	*comp*	$[GN \rightarrow \bullet DET\ N, 4, 4]$ et 14
16	$[N \rightarrow fille\bullet, 5, 6]$	*scan*	$[N \rightarrow \bullet fille, 5, 5]$
17	$[GN \rightarrow DET\ N\bullet, 4, 6]$	*comp*	14 et 16
18	$[GV \rightarrow V\ GN\bullet, 3, 6]$	*comp*	9 et 17
19	$[GV \rightarrow V\ GN\ GNP\bullet, 3, 6]$	*comp*	11 et 17
20	$[S \rightarrow GNGV\bullet, 5, 1]$	*comp*	6 et 18
21	$[S \rightarrow GN \bullet GV, 4, 6]$	*comp*	$[S \rightarrow \bullet GN\ GV, 4, 4]$ et 17
22	$[GN \rightarrow GN \bullet GNP, 4, 6]$	*comp*	$[GN \rightarrow \bullet GN\ GNP, 4, 4]$ et 17

TAB. 12.4 – Parsage tabulaire ascendant

De manière implicite, le remplissage de la Table 12.4 obéit à une stratégie d'application des règles suivant :
– insérer tous les items résultant de l'application de *init*
– s'il est possible d'appliquer *comp*, choisir d'appliquer *comp*
– sinon s'il est possible d'appliquer *scan*, appliquer *scan*
Indépendamment d'une choix d'une stratégie de prioritisation des règles, l'implantation d'un algorithme demande de maintenir dans des structures séparées les items avant et après que l'on a évaluer leurs « conséquents ». Dans le jargon du domaine, ces deux structures sont la table (ou *chart*) et *l'agenda*. De manière très schématique, l'algorithme précédent se récrit en
– insérer tous les items résultant de l'application de *init* dans l'agenda.
– prendre un item de l'agenda, l'insérer dans la table, et insérer tous ses conséquents directs dans l'agenda.
Resterait encore pour achever la spécification complète d'un algorithme à définir comment choisir entre plusieurs applications concurrentes de la même règle (ainsi la rafale de *comp* donnant lieu aux items 8 à 13). Ainsi qu'à mieux analyser l'influence des malformations possibles de la grammaire (productions vides, productions non-génératives, récursions gauches...).

Une analyse plus poussée de cet algorithme révèle un certain nombre d'inefficacités :

1. *init* met à disposition de *comp* tous les éléments potentiellement utiles pour faire de l'analyse ascendante. Cette étape est toutefois bien trop prolixe et conduit à l'insertion dans la table de nombreux items qui ne seront jamais considérés ; le cas le plus flagrant étant l'insertion d'items actifs introduisant des terminaux qui n'apparaissent même pas dans la phrase.

2. *comp* est parfois utilisée inutilement : ainsi la création de l'item (inutile) 13, qui ne pourra jamais être développé puisque S n'apparaît dans aucune partie droite.

   De même, 21 et 22 sont inutiles, 21 doublement d'ailleurs parce que (i) le GV manquant ne pourra pas être trouvé au delà de la position 6 et (ii) quand bien même il le serait, reconnaître un S en position 4 ne pourra que conduire l'analyse dans une impasse.

Nous examinons, dans les variantes décrites ci-dessous, divers remèdes à ces inefficacités.

12.2.4 Coin gauche

L'approche "coin gauche" consiste essentiellement à revenir à une stratégie initialisation de la table semblable à celle mise en oeuvre pour CYK. Cela suppose que tous les terminaux soient introduits par des règles de la forme $A \rightarrow a$ et que ces règles soient les seules à introduire des terminaux.[4]Considérons que c'est bien le cas, comme dans la Table 12.1, ce qui nous autorise alors à remplacer l'étape *init* par une nouvelle définition selon :

for $i \in (1 \ldots n)$ **do**
  foreach $A \rightarrow u_i$
    insert$([A \rightarrow u_i \bullet, i, i+1])$ in T
  od
od

Le problème est maintenant de déclencher l'insertion d'items actifs, qui permettront de nourrir la règle fondamentale (*comp*). L'idée de l'analyse du "coin gauche" consiste à émettre une hypothèse sur la reconnaissance d'un constituant que si l'on a déjà complètement achevé de reconnaître son "coin gauche" (c'est-à-dire le premier élément, terminal ou non terminal, de la partie droite). Cette stratégie se formalise par la règle *leftc* :

if $[Y \rightarrow \alpha \bullet, i, j] \in T$
  then foreach $X \rightarrow \beta Y \beta \in P$ **do**
    insert$([X \rightarrow Y \bullet \beta], i, j)$ in T
  od
fi

Cette nouvelle règle effectue en un coup la reconnaissance du coin gauche à partir d'un item inactif, là où l'algorithme précédent décomposait cette reconnaissance en une étape *init* (créant un item actif initial) et une étape de *comp* (reconnaissant le coin gauche).

En présence de productions ε, il faudrait également se donner la possibilité de reconnaître des productions vides par une règle de type :

foreach $A \rightarrow \varepsilon \in P$ **do**
  foreach $i \in 1 \ldots n)$ **do**
    insert$([A \rightarrow \bullet \beta], i, i)$ in T
  od
od

[4]Ces deux conditions sont de pure convenance : on pourrait aussi bien définir un analyseur du coin gauche pour une grammaire ayant une forme quelconque et traiter les terminaux au moyen de deux règles : l'une effectuant un *scan* spécifique pour les terminaux correspondant à des coins gauches, et l'autre effectuant un *scan* pour les terminaux insérés dans les queues de parties droites. Sauriez-vous mettre en oeuvre une telle analyse ?

Il suffit maintenant, pour compléter cette description, de spécifier un ordre d'application de ces différentes règles. Les ordonner selon : *comp* est prioritaire sur *leftc*, elle-même prioritaire sur *init*, conduit à la trace suivante (voir la Table 12.5). Notons qu'ici encore le choix d'appliquer *leftc* avant d'en avoir fini avec les appels à *init* n'est pas une nécessité et un autre ordre aurait en fait été possible.

Num.	Item	règle	Antécédents
1	$[DET \rightarrow un\bullet, 1, 2]$	*init*	
2	$[GN \rightarrow DET \bullet N, 1, 2]$	*leftc*	1
3	$[N \rightarrow père\bullet, 2, 3]$	*init*	
4	$[GN \rightarrow DET\ N\bullet, 1, 3]$	*comp*	2 et 3
5	$[S \rightarrow GN \bullet GV, 1, 3]$	*leftc*	4
6	$[GN \rightarrow GN \bullet GNP, 1, 3]$	*leftc*	4
7	$[V \rightarrow gronde\bullet, 3, 4]$	*init*	
8	$[GV \rightarrow V\bullet, 3, 4]$	*leftc*	7
9	$[S \rightarrow GN\ GV\bullet, 1, 4]$	*compl*	5 et 8
10	$[GV \rightarrow V \bullet GN, 3, 4]$	*leftc*	7
11	$[GV \rightarrow V \bullet GNP, 3, 4]$	*leftc*	7
12	$[GV \rightarrow V \bullet GN\ GNP, 3, 4]$	*leftc*	7
13	$[GV \rightarrow V \bullet GNP\ GNP, 3, 4]$	*leftc*	7
14	$[DET \rightarrow sa\bullet, 4, 5]$	*init*	
15	$[GN \rightarrow DET \bullet N, 4, 5]$	*leftc*	14
16	$[N \rightarrow fille\bullet, 5, 6]$	*init*	
17	$[GN \rightarrow DET\ N\bullet, 4, 6]$	*compl*	15 et 16
18	$[GV \rightarrow V\ GN\bullet, 3, 6]$	*comp*	10 et 17
19	$[GV \rightarrow V\ GN \bullet GNP, 3, 6]$	*comp*	12 et 17
20	$[S \rightarrow GN\ GV\bullet, 1, 6]$	*comp*	6 et 18
21	$[S \rightarrow GN \bullet GV, 4, 6]$	*leftc*	17
22	$[GN \rightarrow GN \bullet GNP, 4, 6]$	*leftc*	17

TAB. 12.5 – Parsage tabulaire : stratégie du coin gauche

La différence avec la trace précédente (cf. la Table 12.4) ne saute pas aux yeux ; elle est pourtant sensible puisque les 22 items de la Table 12.5 sont les *seuls items construits*, alors qu'il fallait, pour avoir la liste complète des items de la table précédente, y ajouter les $n \times |P|$ items initiaux. Remarquons également la présence d'un certain nombre d'items inutiles dans la Table 12.5 : ainsi les items 6, 8, 9, 19, 21 et 22, soit presqu'un item sur trois.

12.2.5 Une stratégie mixte : l'algorithme d'Earley

Répétons-nous : l'efficacité d'un système de parsage dépend fortement du nombre d'items qui ont été inutilement ajoutés dans la table (et corrélativement développés). Nous l'avons mentionné plus haut, dans le cas d'un algorithme purement ascendant, l'ajout d'items inutiles est en fait chose courante. Considérez, en effet, l'analyse d'une phrase comme : *le fils de ma tante pleure*. La reconnaissance du groupe nominal (*GN*) *ma tante*, entraîne l'insertion ascendante des hypothèses correspondant à la règle $S \rightarrow GN\ GV$. Ces hypothèses vont, de plus, être prolongées, puisqu'on va bien trouver par la suite le *GN* (*ma tante*) comme le GV (*pleure*). Pourtant, il est impossible de trouver ici une phrase, puisqu'on laisserait inanalysés les trois premiers mots de la phrase (*le fils de*).

L'idée de l'algorithme d'Earley (Earley, 1970) est de guider la stratégie purement ascendante décrite à la section 12.2.3 par des informations de contrôle descendantes, qui vont permettre de ne développer que les items qui peuvent intervenir dans une analyse complète. À cet effet, la table est complétée par des items correspondant à des *prédictions* (descendantes) ; un hypothèse ne sera alors développée que si elle est correspond à une telle prédiction.

Cette idée est mise en œuvre en remplaçant l'étape d'initialisation aveugle de l'algorithme purement ascendant par une initialisation plus raisonnable, consistant à faire l'hypothèse minimale que l'on débute la reconnaissance d'une phrase (S) à la position 1. Ceci est formalisé par une nouvelle version de *init* :

foreach $S \rightarrow \alpha$ **do**
  insert($[S \rightarrow \bullet\alpha, 1, 1]$) in T
od

La mise en œuvre de cette approche demande une nouvelle règle exprimant les prédictions descendantes : il s'agit d'exprimer le fait qu'on ne fait l'hypothèse de démarrage d'un nouveau constituant que si ce constituant est attendu, ie. que si sa reconnaissance pourra être utilisée pour développer un item déjà existant. Cette nouvelle règle (*pred*) s'écrit donc[5] :

if $[X \rightarrow \alpha \bullet B\beta, i, j] \in T$
  then *insert*($[B \rightarrow \bullet\alpha, j, j]$) in T
fi

L'idée générale de l'algorithme est alors la suivante :
– initialiser les items actifs correspondants à S avec *init*
– appliquer les trois règles restantes, avec l'ordre de priorité suivant : *comp* est prioritaire sur *pred*, lui-même prioritaire sur *scan*.
La Table 12.6 donne la trace de l'utilisation de cette stratégie sur la phrase *Le fils de ma tante pleure*. Conformément aux idées originales de Earley, nous nous sommes permis une petite licence consistant à effectuer un regard avant de 1 au moment de chercher les terminaux. Ceci économise tout un tas de prédictions inutiles d'items tels que $[DET \rightarrow \bullet ma]$... Un résultat identique serait obtenu en complétant l'initialisation par un remplissage "à la CYK" des items introduisant des terminaux. Ce scan un peu amélioré est noté *scan+*.

L'incorporation du contrôle descendant des hypothèses permet donc effectivement d'éviter de construire un certain nombre d'items, comme en particulier ceux qui correspondraient à une hypothèse de démarrage de phrase au milieu (c'est-à-dire après la reconnaissance du *GN ma tante*) de l'énoncé. Sous cette forme, il apparaît toutefois que de nombreux items inutiles sont crées, soit parce que le mécanisme de prédiction est trop libéral (ainsi les items 10 à 14, 28 à 33 (seul 29 est utile), ou ceux créés après l'item 40 ; soit parce que la complétion agit sur des items sans avenir (ainsi les items 36 à 39).

Extensions Il est donc souhaitable et possible de raffiner encore cet algorithme :
– en incorporant des possibilités de regard avant (*look-ahead*), qui vont permettre, à nouveau, d'éviter de développer certains items. Ainsi par exemple, l'item 25 est inséré par prédiction d'un GNP. Or, il suffirait de regarder l'input pour réaliser que c'est impossible, puisqu'un GNP ne peut démarrer qu'avec une des prépositions, et que l'input contient un verbe. Comme pour les analyseurs LL, ce type de filtrage demanderait de calculer la table des *FIRST*. Il est également

[5] Une manière d'interpréter cette règle est de la voir comme simulant, de concert avec *comp*, la construction dynamique d'une table d'analyse LR (voir la section 7.2).

1	$[S \rightarrow \bullet GN\ GV, 1, 1])$	*init*	
2	$[GN \rightarrow \bullet DET\ N, 1, 1])$	*pred*	1
3	$[GN \rightarrow \bullet GN\ GNP, 1, 1])$	*pred*	1
4	$[DET \rightarrow le\bullet, 1, 2])$	*scan+*	2
5	$[GN \rightarrow DET \bullet N, 1, 2])$	*comp*	1,4
6	$[N \rightarrow fils\bullet, 2, 3])$	*scan+*	5
7	$[GN \rightarrow DETN\bullet, 1, 3])$	*comp*	2, 6
8	$[S \rightarrow GN \bullet GV, 1, 3])$	*comp*	1, 7
9	$[GN \rightarrow GN \bullet GNP, 1, 3])$	*comp*	3,7
10	$[GV \rightarrow \bullet V, 3, 3]$	*pred*	8
11	$[GV \rightarrow \bullet V\ GN, 3, 3]$	*pred*	8
12	$[GV \rightarrow \bullet VGNP, 3, 3]$	*pred*	8
13	$[GV \rightarrow \bullet VGNGNP, 3, 3]$	*pred*	8
14	$[GV \rightarrow \bullet V\ GNP\ GNP, 3, 3]$	*pred*	8
15	$[GNP \rightarrow \bullet PP\ GN, 3, 3]$	*pred*	9
16	$[PP \rightarrow de\bullet, 3, 4])$	*scan+*	15
17	$[GNP \rightarrow PP \bullet GN, 3, 4]$	*comp*	15 et 16
18	$[GN \rightarrow \bullet DET\ N, 4, 4])$	*pred*	17
19	$[GN \rightarrow \bullet GN\ GNP, 4, 4])$	*pred*	17
20	$[DET \rightarrow ma\bullet, 4, 5])$	*scan+*	18
21	$[GN \rightarrow DET \bullet N, 4, 5])$	*comp*	18 et 20
22	$[N \rightarrow tante\bullet, 5, 6])$	*scan+*	21
23	$[GN \rightarrow DET\ N\bullet, 4, 6])$	*comp*	21 et 22
24	$[GNP \rightarrow PP\ GN\bullet, 3, 6]$	*comp*	17 et 23
25	$[GN \rightarrow GN\ GNP\bullet, 1, 6])$	*comp*	24 et 9
26	$[S \rightarrow GN \bullet GV, 1, 6])$	*comp*	1, 25
27	$[GN \rightarrow GN \bullet GNP, 1, 6])$	*comp*	3,7
28	$[GV \rightarrow \bullet V, 6, 6]$	*pred*	26
29	$[GV \rightarrow \bullet V\ GN, 6, 6]$	*pred*	26
30	$[GV \rightarrow \bullet VGNP, 6, 6]$	*pred*	26
31	$[GV \rightarrow \bullet VGNGNP, 6, 6]$	*pred*	26
32	$[GV \rightarrow \bullet V\ GNP\ GNP, 6, 6]$	*pred*	26
33	$[GNP \rightarrow \bullet PP\ GN, 6, 6]$	*pred*	27
34	$[V \rightarrow pleure\bullet, 6, 7])$	*scan+*	33
35	$[GV \rightarrow V\bullet, 6, 7]$	*comp*	34
36	$[GV \rightarrow V \bullet GN, 6, 7]$	*comp*	34
37	$[GV \rightarrow V \bullet GNP, 6, 7]$	*comp*	34
38	$[GV \rightarrow V \bullet GNGNP, 6, 7]$	*comp*	34
39	$[GV \rightarrow V \bullet GNP\ GNP, 6, 7]$	*comp*	34
40	$[S \rightarrow GN\ GV\bullet, 1, 7])$	*comp*	26 et 35
41	$[GN \rightarrow \bullet DET\ 7, 7]$	*pred*	36
42	$[GNP \rightarrow \bullet PP\ GN, 7, 7]$	*pred*	37
	...	*pred*	41

TAB. 12.6 – Développement de la table d'analyse avec l'analyseur d'Earley

possible de filtrer des complétions abusives par utilisation d'un regard avant : de cette manière les items 36 à 39 pourraient-ils être également évités. Le bénéfice de ces stratégies est toutefois contesté, dans la mesure où elles induisent un coût algorithmique non négligeable.
– en retardant les prédictions jusqu'à la reconnaissance d'un coin gauche. Ceci peut se voir comme une extension relativement naturelle de l'algorithme de la section 12.2.4, consistant à :
  – ajouter un item supplémentaire "descendant" à l'initialisation, de la forme $[S \rightarrow \bullet\alpha, 1, 1]$
  – ajouter une condition supplémentaire à la règle du coin gauche, demandant qu'un item $[A \rightarrow B \bullet \alpha, i, j]$ ne soit inséré que (i) si $[B \rightarrow \bullet\beta, i, j]$ existe (coin gauche normal) et si (ii) A est prédit de manière descendante, correspondant à un item $[X \rightarrow \gamma \bullet Y\delta, l, i]$, et A peut débuter une dérivation de Y.

Complexité L'algorithme d'Earley permet de construire des analyseurs pour des grammaires quelconques , avec une complexité également $O(n^3)$: l'analyseur doit considérer un par un tous les items de l'agenda pour les insérer dans la table. Le nombre de productions pointées étant une constante de la grammaire, il y a au maximum de l'ordre de $O(n^2)$ items ; l'examen d'un item $[A \rightarrow \alpha \bullet B\ \beta, i, j]$, notamment l'étape de complétion, demande de chercher tous les items inactifs potentiellement utilisables $[B \rightarrow \gamma\bullet, j, k]$, introduisant un troisième indice libre k entre 1 et n. On obtient ainsi la complexité annoncée. Dans la pratique, les analyseurs obtenus sont "souvent" quasi-linéaires[6]. On peut finalement montrer que la complexité est dans le pire cas $O(n^2)$ pour des grammaires non-ambiguës.

Préfixe viable Une propriété remarquable de l'algorithme d'Earley est sa capacité à localiser sans délai les erreurs. Cette propriété, dite du *préfixe viable*, garantit qu'une erreur est détectée par l'algorithme lors du scan du premier symbole u_i tel que $u_1 \ldots u_i$ n'est pas un préfixe d'au moins un mot de $L(G)$.

Preuve. C'est vrai pour les items initiaux. Supposons que ce soit vrai après insertion du nième item et considérons l'insertion d'un item supplémentaire. Deux cas sont à considérer :
– soit cet item provient d'une application de *pred* ou de *comp* : dans ce cas, le nouvel item $[B \rightarrow \bullet\alpha, i, j]$ ne couvre aucun symbole terminal supplémentaire ; la propriété du préfixe viable reste donc satisfaite.
– soit cet item provient d'une application de *scan* : ceci signifie que le symbole lu sur l'entrée courante avait été prédit et est donc susceptible de donner lieu à un développement de l'analyse.

12.3 Compléments

12.3.1 Vers le parsage déductif

Nous avons pour l'instant développé de manière relativement informelle les algorithmes de parsage tabulaire. Il nous reste à aborder maintenant une dernière question importante, qui est celle de la correction de ces algorithmes. Comment s'assurer, que formalisés sous la forme d'une technique de remplissage d'une table, ces algorithmes ne "manquent" pas des analyses ?

Informellement, deux conditions doivent être simultanément remplies pour assurer la validité de cette démarche :

[6] En particulier si la grammaire est déterministe !

– chaque analyse possible doit être représentable sous la forme d'un item de la table. C'est évidemment le cas avec les items que nous avons utilisé à la section précédente, qui expriment une analyse complète de $u_1 \dots u_n$ par un item de la forme $[S \rightarrow \alpha\bullet, 1, n]$.
– le mécanisme de déduction des items doit être tel que :
  – seuls sont déduits des items conformes à la stratégie de parsage ;
  – tous les items conformes sont déduits à une étape de donnée de l'algorithme : propriété de *complétude*.

  La notion de conformité introduite ici est relative aux propriétés qu'expriment les items. Ainsi, il est possible de montrer que tout item de l'algorithme d'Earley vérifie la propriété suivante (on parle également *d'invariant*) :

$$[A \rightarrow \alpha \bullet \beta, i, j] \in T \leftrightarrow \begin{cases} S \stackrel{\star}{\Rightarrow} & u_1 \dots u_{i-1} A \gamma \\ \alpha \rightarrow & u_i \dots u_{j-1} \end{cases}$$

Lorsque ces propriétés sont remplies, on dit que l'algorithme est correct. Des résultats généraux concernant la correction des algorithmes tabulaires sont donnés dans (Sikkel and Nijholt, 1997), ou encore dans (Shieber et al., 1994), qui ré-exprime les stratégies tabulaires dans le cadre de systèmes de déductions logiques.

12.3.2 D'autres stratégies de parsage

L'alternative principale aux techniques tabulaires pour le parsage de grammaires ambiguës est le parsage LR généralisé (GLR) (Tomita, 1986). L'idée de base de cette approche conduire une analyse LR classique en gérant les ambiguïtés "en largeur d'abord". Rencontrant une case qui contient une ambiguïté, on duplique simplement la pile d'analyse pour développer en parallèle les multiples branches. S'en tenir là serait catastrophique, pouvant donner lieu à un nombre exponentiel de branches à maintenir en parallèle. Cette idée est "sauvée" par la factorisation des piles dans un graphe, permettant d'éviter de dupliquer les développements des piles qui partagent un même futur. Cette factorisation permet de récupérer la polynomialité de l'algorithme, qui, comme celui d'Earley a une complexité en $O(n^3)$ dans le pire cas, et souvent meilleure pour des grammaires « pas trop ambiguës ».

Les meilleurs résultats théoriques pour le parsage des grammaires CF sont donnés par l'algorithme proposé par dans (Valiant, 1975), qui promet une complexité en $O(2^{2.81})$, en ramenant le problème du parsage à celui du produit de matrices booléennes, pour lequel il existe des algorithmes extrêmement efficaces. Ce résultat n'est toutefois intéressant que d'un point de vue théorique, car la constante exhibée dans la preuve de Valiant est trop grande pour que le gain soit réellement significatif dans les applications les plus courantes.

Pour conclure avec les questions de la complexité, il est intéressant (et troublant) de noter que pour l'immense majorité des grammaires, on sait construire (parfois dans la douleur) des analyseurs linéaires. Par contre, on ne dispose pas aujourd'hui de méthode générale permettant de construire un analyseur linéaire pour n'importe quelle grammaire. En existe-t-il une ? La question est ouverte. Par contre, dans tous les cas, pour lister toutes les analyses, la complexité est exponentielle[7], tout simplement parce que le nombre d'analyses est dans le pire des cas exponentiel.

[7] Si l'on n'y prend garde les choses peuvent même être pire, avec des grammaires qui seraient infiniment ambiguës.

Chapitre 13

Grammaires d'arbre adjoints

Ce chapitre nous entraîne à la découverte d'un nouveau formalisme de description de langages, les *grammaires d'arbres adjoints* (en anglais *Tree adjoining grammars*, ou TAGs). Ce formalisme est intéressant à plus d'un titre : tout d'abord, c'est un représentant de la famille des grammaires d'arbres, qui constituent des modèles formels opérant non plus sur des chaînes, mais sur des objets structurés prenant la forme d'arbres. Ce modèle est intéressant à un second titre, en ce qu'il permet non seulement de représenter tous les langages hors contexte, mais encore un certain nombre de langages strictement contextuels. En troisième lieu, nous verrons que ce formalisme ne permet pas de représenter tous les langages contextuels, mais seulement une sous-classe d'entre eux, connue sous le nom de classe des *langages faiblement contextuels* (en anglais *mildly context sensitive languages*). Cette limite d'expressivité des TAGs a une contrepartie bienvenue : il existe des algorithmes d'analyse polynomiaux pour les TAGs, qui généralisent les algorithmes tabulaires présentés au chapitre 12. Il existe en fait une cinquième raison de s'intéresser aux TAGs, qui est de nature plus linguistique : dans leur version lexicalisée, elles fournissent un formalisme cohérent pour décrire le comportement combinatoire des objets syntaxiques.

Ce chapitre s'organise comme suit : dans un premier temps nous introduisons ce nouveau formalisme et discutons de quelques propriétés formelles importantes. La seconde section présente un algorithme polynomial pour analyser les TAGS, qui transpose aux TAGs l'algorithme CYK étudié à la section 12.1.

13.1 Les grammaires d'arbre adjoints

13.1.1 Introduction

Comme leur nom l'indique, les grammaires d'arbres, contrairement aux grammaires syntagmatiques traditionnelles, manipulent des chaînes de symboles hiérarchiquement structurées, prenant la forme d'arbres étiquetés. Dans le formalisme des grammaires d'arbres adjoints[1], les arbres sont susceptibles de se combiner librement par une unique opération binaire, l'opération *d'adjonction*, qui permet de dériver des arbres à partir d'arbres déjà existants.

[1] L'utilisation de modèles opérant sur les arbres ne se limite pas à ce formalisme : ainsi, par exemple, il est possible de définir des automates d'arbres, qui dans leur version élémentaire ont une expressivité équivalente aux grammaires CF.

L'adjonction Commençons par une analogie : l'adjonction de deux arbres est analogue (dans le domaine des arbres) à ce que ferait une insertion dans le domaine des chaînes : quand u s'insère dans v, v est littéralement coupée en deux : la partie à gauche de u, v_1, et la partie à droite de u, v_2. Le résultat de cette opération est v_1uv_2 et est toujours bien défini. Transposer cette opération sur des arbres non étiquetés revient à détacher "verticalement" un sous-arbre v_2 de v, et à le rabouter sur une feuille de u ; ce nouveau sous-arbre est ensuite rabouté à v_1, la partie "haute" de l'arbre. Lorsque l'on transpose maintenant cette opération à des arbres dont les nœuds portent des étiquettes symboliques, il faut prendre une précaution supplémentaire, pour s'assurer que w peut se "concaténer" à v_1, et v_2 à u. Cette vérification demande de distinguer en fait deux types d'arbres :

– ceux qui reçoivent les insertions, qu'on appelle les *arbres initiaux*
– ceux qui sont s'insèrent (s'adjoignent), qu'on appelle les *arbres auxiliaires*, qui ont une forme bien particulière : ils contiennent nécessairement une feuille distinguée qui porte la même étiquette que la racine. Cette feuille est appelée le nœud *pied* et est distinguée notationnellement par un astérisque. L'adjonction ne peut opérer sur un nœud pied.

La différence de comportement entre ces deux types d'arbres est illustrée à la Figure 13.1, qui représente une adjonction de l'arbre auxiliaire β sur l'arbre initial α. La racine et le pied de β sont étiquetés par le même symbole X, qui étiquette également le nœud de α sur lequel opère l'adjonction.

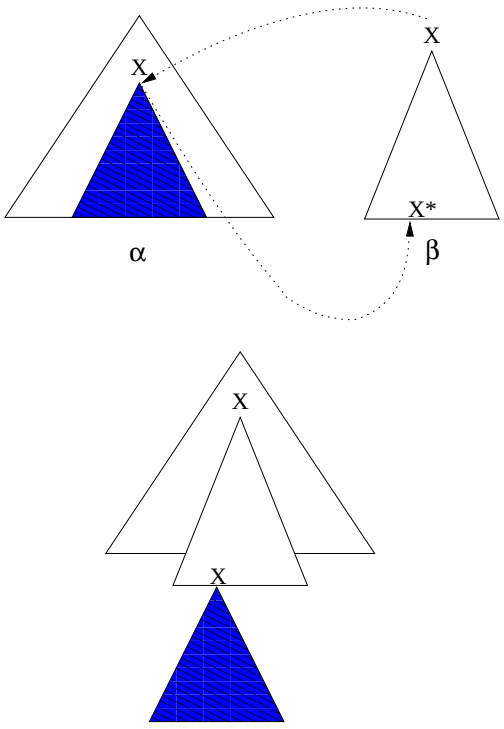


FIG. 13.1 – L'opération d'adjonction

Une autre analogie parlante consiste à voir l'adjonction comme séparant en deux le nœud sur lequel elle opère ; sur la partie "haute" se greffera la racine de l'arbre adjoint ; la partie basse venant se greffer sur le pied de l'arbre adjoint. Dernière remarque importante : l'adjonction est une opération *non contextuelle*, signifiant qu'elle est susceptible de concerner un nœud indépendamment de ses voisins.

Définitions Formellement, une grammaire d'arbres adjoints est définie par un quintuplet $G = (\Sigma, V, S, I, A)$, où Σ et V sont deux ensembles finis et disjoints de symboles, correspondant aux éléments respectivement terminaux et non-terminaux de G, S est un symbole distingué de V (l'axiome) et où I et A sont des ensembles finis d'arbres respectivement initiaux et auxiliaires, étiquetés par des symboles de $\Sigma \cup V$ et respectant les contraintes suivantes :
– les terminaux ne peuvent étiqueter que des feuilles
– les nœuds internes sont étiquetés par des non-terminaux
– il y a exactement un nœud pied par arbre auxiliaire
L'ensemble $E = I \cup A$ regroupe tous les arbres *élémentaires*, que l'on oppose ainsi aux arbres *dérivés*, c'est-à-dire aux arbres construits par une ou plusieurs opérations d'adjonction.

Une dérivation TAG encode une série de combinaisons entre arbres : on note $\gamma_1 \Rightarrow \gamma_2$ si l'arbre dérivé γ_2 est le résultat d'une adjonction sur un nœud de γ_1 ; et $\stackrel{\star}{\Rightarrow}$ la clôture réflexive et transitive de $\Rightarrow$. Le langage d'arbre $T(G)$ engendré par une TAG est défini comme étant l'ensemble de tous les arbres qu'il est possible de dériver des arbres initiaux en utilisant un nombre arbitraire de fois l'opération d'adjonction, soit formellement :

$$T(G) = \{\gamma, \exists \gamma_0 \in I \text{ tq. } \gamma_0 \stackrel{\star}{\Rightarrow} \gamma\}$$

Par définition, $T(G)$ contient tous les arbres initiaux.

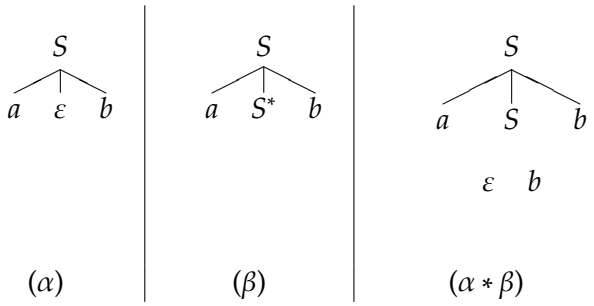
On appelle *frontière* d'un arbre la séquence de symboles recueillis sur les feuilles lors d'un parcours préfixe de l'arbre. Si τ est un arbre, on notera $\overline{\tau}$ sa frontière. Un arbre est dit *complété* si sa frontière ne contient que des symboles terminaux. On définit alors :

Définition 13.1. *Le langage $L(G)$ engendré par une grammaire d'arbres adjoints G est défini par :*

$$L(G) = \{w \in \Sigma^\star, \exists \tau \in T(G), w = \overline{\tau} \text{et } \tau \text{ a pour racine } S\}$$

On appelle langage d'arbre adjoint *(en anglais Tree Adjoining Language) un langage engendré par une TAG.*

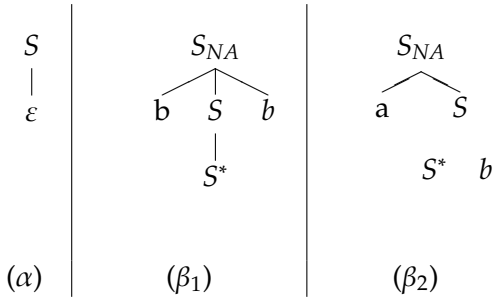
À titre d'illustration, considérons la grammaire suivante, définie par $(\{a, b\}, \{S\}, S, \{\alpha\}, \{\beta\})$, où α et β, ainsi que l'arbre dérivé $\alpha * \beta$, résultant de l'adjonction de β sur le nœud racine de α, sont représentés dans la Table 13.1.



Tab. 13.1 – Une grammaire TAG pour $\{a^n b^n, n > 0\}$

Variantes Nous l'avons mentionné, l'adjonction ne peut opérer sur un nœud pied. Une autre contrainte, souvent proposée dans la littérature, consiste à interdire les adjonctions répétées sur un même nœud. Ce type de contrainte peut être généralisé en distinguant systématiquement dans les arbres initiaux les nœuds sur lesquels l'adjonction est possible, interdite, ou encore

obligatoire. Cette distinction est traditionnellement marquée par les indices respectivement *NA* et *OB* sur ces deux derniers types de nœud. Si l'on interdit les adjonctions multiples sur un même nœud, la grammaire précédente (13.1) n'est plus valide. $\{a^n b^n, n > 0\}$ est engendré par les deux grammaires de la Table 13.2, utilisent respectivement β_1 et β_2 comme unique arbre auxiliaire. L'ajout de contraintes locales sur les nœuds internes augmente strictement, nous le verrons, l'expressivité de ce formalisme.



TAB. 13.2 – Des grammaires TAG avec contraintes locales pour $\{a^n b^n\}$

Une autre variante des TAGs considère une seconde opération binaire, l'opération de *substitution*. Dans cette variante, les feuilles d'un arbre élémentaire peuvent également être étiquetés par des variables. L'opération de substitution consiste alors à greffer un arbre β de racine (non-terminale) X, sur une feuille d'un arbre α étiquetée par le même symbole X. Notationnellement, les feuilles sur lesquelles une opération de substitution peut porter (on parle également de *sites* de substitution) sont identifiées par une flèche verticale descendante, comme sur la Figure 13.2. Les nœuds recevant une susbstitution ne peuvent recevoir d'adjonction. L'ajout de cette nouvelle opération

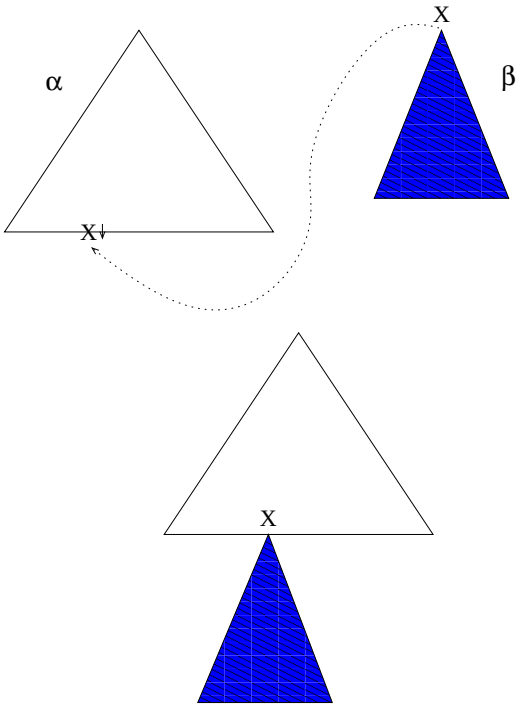
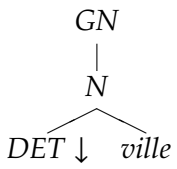
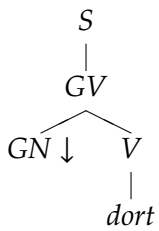
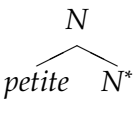
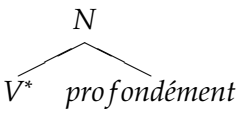
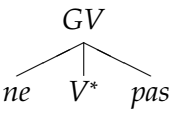


FIG. 13.2 – L'opération de substitution

ne modifie pas l'expressivité des TAGs : au prix de transformations des arbres élémentaires, tout langage engendré par une TAG avec substitution peut également être engendré par une TAG sans substitution.

Dérivations Une dérivation note une séquence d'opérations binaires. Comme pour les CFG, les dérivations se représentent par des arbres, qui permettent de s'abstraire de l'ordre dans lequel les opérations sont effectuées. Que représenter dans ces arbres ? les arbres impliqués, qui sont représentés par leur noms, l'arbre recevant l'opération dominant l'autre opérande ; la position où l'opération a lieu, identifiée par le numéro (par exemple par le numéro de Gorn[2]) du nœud sur lequel porte l'opération ; enfin le type d'opération (adjonction ou substitution), traditionnellement distingué par des branches figurées respectivement en trait continu et discontinu.

Illustons cette notion à l'aide d'un exemple tiré du domaine de la syntaxe naturelle, en considérant les arbres de la Table 13.3, qui permet d'engendrer, par exemple, *la ville dort*, *la ville dort profondément*, *la petite ville ne dort pas*.

		
(α_1)	(α_2)	(α_3)
		
(β_1)	(β_2)	(β_3)

TAB. 13.3 – Une grammairette pour les langues naturelles

L'arbre de dérivation de la phrase *la petite ville ne dort pas* est représenté (avec toutefois un seul type de traits) à la Figure 13.3.

La Figure 13.3 permet d'illustrer deux propriétés importantes des TAGS :
- à la différence des CFGs, il existe une différence entre l'arbre de dérivation, qui encode l'histoire des opérations ayant donné lieu à un arbre et l'arbre dérivé, qui est le résultat de cette suite d'opération.
- comme pour les CFG, l'arbre de dérivation représente un ensemble de séquences d'opérations, l'ordre dans lequel les opérations ayant lieu étant, pour partie, arbitraire. Ainsi, dans l'exemple précédent, l'adjonction de β_3 peut se réaliser avant ou après l'opération de substitution.

13.1.2 Quelques propriétés des TAGs

Nous introduisons ici les principales propriétés des TAGs, en nous inspirant principalement des travaux conduits par A. Joshi et ses collègues et dont les principaux résultats sont présentés dans

[2] L'indice de Gorn d'un nœud encode numériquement le chemin (unique) entre la racine de l'arbre et le nœud selon le schéma récursif suivant :
- la racine possède l'indice 1
- le j-ème fils du nœud d'indice i porte le numéro $i.j$.

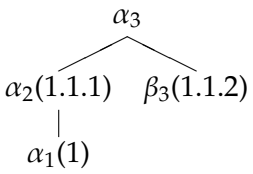


FIG. 13.3 – Un arbre de dérivation

L'arbre représentant *la petite ville ne dort pas* est obtenu par la suite d'opérations suivantes :

- substitution de l'arbre α_2 sur le site *DET*, d'indice de Gorn 1.1, de l'abre α_1. L'arbre résultant a *la ville* comme frontière.
- substitution de l'arbre résultant sur le site *GN* de l'arbre α_3, construisant un arbre de frontière : *la ville dort*.
- adjonction englobante de l'arbre β_3 sur le nœud interne d'indice 1.1.2 (le verbe) de α_3.

(Joshi et al., 1975; Joshi, 1985; Vijay-Shanker and Joshi, 1985; Joshi and Schabes, 1997).

Les Langages d'arbres adjoints (TAL)

Nous présentons ici une discussion de la capacité générative des TAG, tout d'abord dans leur version originale, puis dans le cas où l'on autorise des contraintes locales sur les adjonctions.

CFL ⊂ TAL

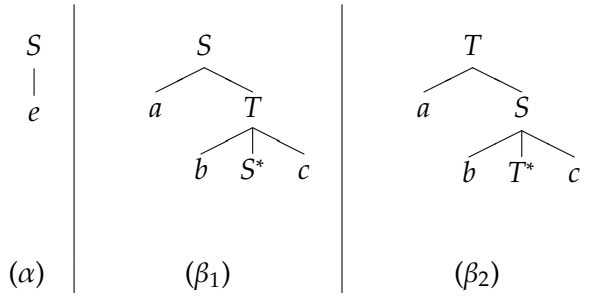
Proposition 13.1. *Pour tout langage hors-contexte L, il existe une TAG G' telle que L(G') = L.*

Preuve. La preuve repose sur une séparation entre les parties récursives et non-récursives de la grammaire CF G représentant L. Appelons L_0 l'ensemble des phrases de L dont la dérivation dans G ne contient aucune récursion de la forme $S \stackrel{\star}{\Rightarrow} \alpha A\beta \stackrel{\star}{\Rightarrow} \gamma A\delta \stackrel{\star}{\Rightarrow} w$. L_0 est nécessairement fini, ainsi que l'ensemble I des arbres de dérivations des mots qu'il contient. I constitue l'ensemble des arbres initiaux de G'. Considérons maintenant l'ensemble des terminaux récursifs de G, c'est-à-dire qui sont tels que $A \stackrel{\star}{\Rightarrow}_\alpha A\beta$. Il est possible, pour chaque non-terminal récursif A, d'isoler (i) un ensemble de dérivations $A \stackrel{\star}{\Rightarrow} \alpha A\beta$ minimal (ie. ne contenant pas d'autre occurence de A) (ii) des ensembles finis de mots dérivables non-récursivement de α et de β. On en déduit, pour chaque non-terminal récursif un ensemble fini d'arbres auxiliaires, puis finalement une grammaire TAG qui engendre le même langage que G. Il est intéressant de noter que cette construction induit une TAG qui engendre les mêmes arbres de dérivation que la grammaire CF initiale et qui lui est donc fortement équivalentes.

Une autre manière, plus directe, de prouver ce résultat consiste à remarquer que, dans la variante autorisant la substitution, les règles CF se transforment trivialement en arbres élémentaires : chaque non-terminal figurant en partie droite devenant un site de substitution dans l'arbre élémentaire correspondant. Le point remarquable de cette transformation est qu'elle ne *crée aucun arbre auxiliaire*. Le fonctionnement des récritures effectuées par une grammaire hors-contexte est donc intégralement pris en charge par l'opération de substitution.

TAL ⊂ CL Les TAL contiennent donc tous les langages hors-contexte et même un (petit) peu plus.

Considérons, en effet, la grammaire TAG dont l'unique arbre initial (α) et les deux arbres initiaux (β_1 et β_2) sont présentés dans la Table 13.4 :



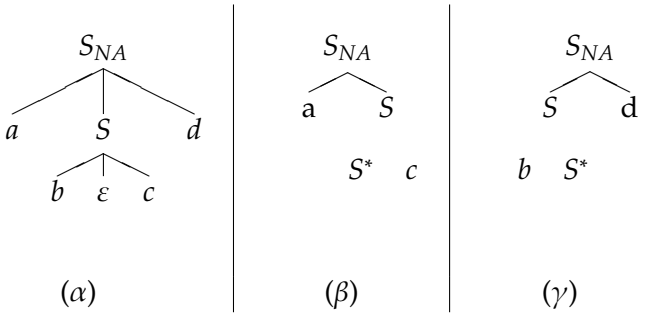
Tab. 13.4 – Une grammaire TAG engendrant un langage contextuel

Quel est le langage engendré par la grammaire de la Table 13.4 ? En étudiant les dérivations possibles avec cette grammaire, il apparaît qu'elle engendre $\{wec^n, n \geq 0\}$, où w satisfait de plus :
– $|w|_a = |w|_b = n$
– si u est un préfixe de w, alors $|u|_a \geq |u|_b$
Ces deux propriétés se démontrent simplement par inférence sur la longueur des dérivations. Ce langage n'est pas un langage CF, son intersection avec le langage rationnel $\{a^\star b^\star e c^\star\}$ donnant lieu au langage strictement contextuel $\{a^n b^n e c^n, n \geq 0\}$. Les TAL constituent donc un sur-ensemble strict des langages CF.

Un point imporant mérite d'être souligné : le surcroît d'expressivité provient uniquement des arbres ayant une profondeur au moins 2, qui sont en fait ceux par lesquels la contextualisation des récritures est opérée. Si l'on se restreint uniquement aux arbres de profondeur 1, définissant la sous-classe des TAGs simples, alors on retrouve une expressivité exactement égale à celle des langages hors-contexte (Joshi et al., 1975). L'intuition est ici que l'on ne peut introduire de dépendance dans les récritures avec seulement des arbres de profondeur 1. Une autre condition nécessaire au saut d'expressivité est l'existence (ou la possibilité de dériver) d'arbres auxiliaires englobants, c'est-à-dire qui sont tels que leur adjonction conduit à une insertion simultanée de symboles de part et d'autre du nœud sur lequel porte l'adjonction. Si l'on restreint, en effet, les adjonctions à opérer de manière latérale, signifiant que le nœud pied est soit le premier, soit le dernier symbole de la frontière, on retrouve de nouveau une expressivité égale à celle des grammaires hors-contexte (Schabes and Waters, 1995).

Le gain d'expressivité est toutefois modeste : dans leur forme primaire, les TAGs ne peuvent en effet représenter le langage contextuel $\{a^n b^n e c^n, n \geq 0\}$. La raison en est claire : un arbre auxiliaire pour ce langage devra avoir un nombre égal de a, de b et de c. Soit β un tel arbre, de racine X. Quelle forme peut prendre la frontière de β ?
– X^*abc : dans ce cas l'adjonction de β sur sa propre tête conduit à une frontière non-conforme : $X^*abcabc$
– aX^*bc : dans ce cas l'adjonction de β sur sa propre tête conduit à une frontière non-conforme : aaX^*bcbc
– ...
L'autorisation de contraintes locales sur l'adjonction permet de gagner (encore un peu) en expressivité. Rappelons que ces contraintes expriment, pour chaque nœud interne d'un arbre élémentaire :
– si une adjonction est possible ou pas
– si elle est possible, quels arbres peuvent s'adjoindre
– si l'adjonction est facultative ou obligatoire. Dans ce dernier cas, un arbre complété devra ne contenir aucun nœud encore en attente d'une adjonction.

Considérons l'exemple de la 13.5, qui inclut de telles contraintes. Cette grammaire engendre le langage $L = \{a^n b^n c^n d^m\}$: toute adjonction de β augmentant simultanément de nombre de a et de c de part et d'autre de b, toute adjonction de γ augmentant simultanément le nombre de b et de d de part et d'autre de c.

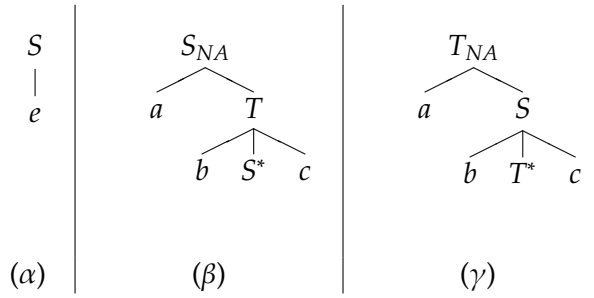


Tab. 13.5 – Une grammaire TAG pour $\{a^n b^m c^n d^m\}$

L'ajout ces contraintes permet également de décrire d'autres langages notoirement contextuels, tels que $\{a^n b^n c^n\}$, ou encore le langage "copie" $\{ww\}$. L'écriture des TAGs correspondantes est laissée en application.

Enchassement et dépendances croisées Les TAGs permettent donc de représenter certains langages contextuels. Ils ne peuvent pas, de loin, les engendrer tous.

Considérons en effet sur les dérivations de la Table 13.6, engendrant le langage contextuel $\{a^n b^n e c^n\}$.



Tab. 13.6 – Une grammaire TAG pour $\{a^n b^n e c^n\}$

Faisons alors l'opération suivante, consistant à identifier par un même indice les terminaux appartenant originellement à un même arbre. L'examen des dérivations montre que l'obtient des séquences de la forme :

$$a_1 a_2 \dots a_n b_n \dots b_1 c_1 \dots c_n$$

Les paires a_i b_i comme les paires b_i c_i se correspondent par "enchassement" ; ce n'est pas le cas des paires a_i c_i qui exhibent des dépendances croisées. Il est clair qu'on ne peut générer trois appariements "enchassés", mais il n'est en fait pas possible non plus d'obtenir avec des TAGs trois dépendances "croisées", consistant à obtenir une séquence de la forme :

$$a_1 a_2 \dots a_n b_1 b_2 \dots b_n c_1 \dots c_n$$

La limitation des TAGs apparaît ici clairement : les TAGs, comme les CFG, peuvent représenter un nombre arbitraire d'appariements entre un nombre fini de symboles appariés ; elles peuvent de

surcroît représenter des dépendances croisées (*cross serial dependencies*), ce que ne permettent pas les CFG. Il existe toutefois une restriction forte sur des dépendances croisées : seules un nombre limité d'entre elles peuvent être représentées, correspondant au maximum à seulement deux (ensembles de) dépendances croisées fonctionnant de manière indépendante.

Croissance bornée Une des avantages du formalisme TAG est qu'il ne manipule que des arbres. Les arbres d'un TAL se "déduisent" les uns des autres par adjonction d'un arbre auxiliaire sur un arbre complété, donnant lieu à une croissance strictement bornée de la longueur des mots. Entre deux mots d'un TAL, la différence de longueur est, en effet, majorée par la longueur de la plus longue frontière d'un arbre adjoint.

Cette observation permet de conclure que des langages comme $\{a^{2^n}, n > 0\}$, qui ne respectent pas cette propriété, ne peuvent être représentés par des TAG.

Propriétés des TALs

Proposition 13.2 (Propriété de clôture). *Les TALs sont clos pour les opérations rationnelles (union, concaténation et étoile de Kleene).*

Preuve. Considérons $G_1 = (\Sigma, V_1, I_1, A_1, S_1)$ et $G_2 = (\Sigma, V_2, I_2, A_2, S_2)$ deux grammaires TAG, avec de plus $V_1 \cap V_2 = \emptyset$. Les propriétés de clôture des TALs se démontrent à l'aide des constructions suivantes :

- $G = (\Sigma, V_1 \cap V_2 \cap \{S\}, I_1 \cap I_2\, A_1 \cap A_2, S)$ réalise l'union de $L(G_1)$ et de $L(G_2)$, au prix d'une petite modification des arbres de racine S_1 (de I_1) et S_2 (de S_2), que l'on étend vers le haut en les "coiffant" d'un nouveau nœud étiqueté par nouvelle racine S.
- la grammaire TAG réalisant la concaténation $L(G_1)L(G_2)$ contient tous les arbres de G_1 et de G_2, auxquels il faut ajouter tous les arbres permettrant de générer la succession et qui sont obtenus en combinant chaque arbre de racine S_1 avec chaque arbre de racine S_2 en un nouvel arbre de racine S, dominant directement S_1 et S_2
- la grammaire TAG réalisant $L(G_1)^\star$ contient tous les arbres de G_1, plus tous les arbres auxiliaires formés en rajoutant une racine S et un frère S^* à un arbre de racine S_1, plus finalement un arbre initial permettant d'éliminer S, consistant en une racine S dominant directement ε(voir la Figure 13.4).

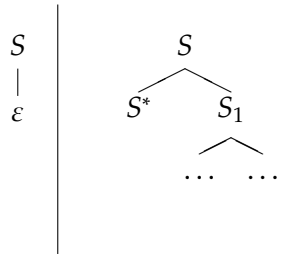


FIG. 13.4 – Deux nouveaux arbres pour l'itération

Les TALs sont également clos pour l'intersection avec les langages rationnels.

Par contre, on dispose des résultats négatifs suivants :

- les TALs ne sont clos ni pour l'intersection ni pour la complémentation.
- l'intersection d'un TAL avec un langage CF ne donne pas nécessairement lieu à un TAL.

13.1.3 Deux formalismes équivalents aux TAGs

Les LIG

Les TAGs permettent de définir une classe de langage nommée *langages faiblement sensibles au contexte*. Il est en fait possible de reconnaitre la même classe de langages en étendant marginalement le formalisme CFG, donnant lieu au formalisme des *grammaires à index linéaires* (en anglais *Linear Index Grammars*, LIGs).

Ce nouveau formalisme comprend une grammaire CF "classique", augmentée par la possibilité d'associer une pile à chaque non-terminal de la grammaire. Les symboles manipulés sur cette pile sont pris dans un ensemble fini d'indices. On note $A[\gamma]$ un non-terminal auquel est associée la pile γ. Chaque production d'une LIG prend alors la forme suivante :

$$A[\gamma] \rightarrow X_1[] \dots X_i[\beta] \dots X_n[]$$

où γ et β se déduisent l'un de l'autre par une opération de pile (*push* ou *pop*). La limitation essentielle des LIG, qui leur confère leur nom de *linéaire*, est qu'un seul et unique non-terminal en partie droite est rendu dépendant de l'histoire de la dérivation de la partie gauche par le mécanisme de transmission de la pile. Tous les autres non-terminaux se dérivent indépendamment de la partie gauche qui leur a donné naissance. Le langage engendré par une LIG contient tous les séquences de terminaux se dérivant de l'axiome S, la pile associée étant originellement vide. Ainsi par exemple la LIG de la Table 13.7 reconnaît le langage $\{wcw, w \in \Sigma^\star\}$, qui est un langage contextuel.

$$\begin{array}{rl} S[\gamma] & \rightarrow aS[\gamma 1] \\ S[\gamma] & \rightarrow bS[\gamma 2] \\ S[\gamma] & \rightarrow T[\gamma] \\ T[\gamma 1] & \rightarrow T[\gamma]a \\ T[\gamma 2] & \rightarrow T[\gamma]b \\ T[] & \rightarrow c \end{array}$$

TAB. 13.7 – Une LIG pour $\{wcw\}$

Les LIGs constituent une restriction des Grammaires à Index (IG) originellement introduites dans (Aho, 1968). Quelques résultats importants :
- les LIG engendrent exactement les mêmes langages que les TAGs (voir eg. (Vijay-Shanker and Weir, 1994)).
- les LIG engendrent un sous-ensemble strict des langages que l'on peut engendrer avec des IG, qui elles-même engendrent un sous-ensemble strict des langages contextuels.

Automates à pile

Comme pour les CFG il est possible de définir un formalisme à base d'automates pour les langages faiblement sensibles au contexte. Ces automates à pile généralisent les PDA présentés au chapitre 9, en remplaçant les piles par des piles de piles. Un mouvement dans l'automate est, comme pour les PDA, déterminé par le symbole sur la tête de lecture et le symbole en haut de pile ; il peut conduire à la création de nouvelles piles (voir par exemple (Joshi and Schabes, 1997)).

13.2 Analyser les TAGs "à la CYK"

Dans cette section, nous décrivons la transposition de l'algorithme CYK, présenté pour les grammaires CF à la section 12.1 au cas des grammaires d'arbres adjoints avec contraintes locales, en suivant, pour l'essentiel, la présentation donnée dans (Shieber et al., 1994). Pour un nœud d'adresse p dans l'arbre τ, noté $\tau@p$, on désigne par $Adj(\tau@p)$ l'ensemble des arbres auxiliaires pouvant s'adjoindre sur ce nœud. Lorsque le nœud n'accepte aucune adjonction, cet ensemble est vide.

Si l'on considère les productions d'une grammaire CF comme des arbres de profondeur 1, on peut dire que CYK implante une stratégie gauche-droite strictement ascendante visant à "découvrir" en remontant un arbre de dérivation formé par concaténation des arbres-productions. La complétion d'un constituant permettant de "remonter" d'un niveau dans l'arbre, par identification de la racine X d'un sous-arbre avec une feuille, également étiquetée X, d'un sous-arbre de niveau supérieur. Dans sa version tabulaire, chaque item de la table correspond à un (ensemble de) sous-arbre(s) couvrant un facteur du mot à analyser.

L'adaptation aux TAGs de CYK procède d'une démarche similaire, à la différence près qu'il faut être capable de gérer des hypothèses sur la sous-chaîne couverte par le pied d'un arbre auxiliaire, qui délimite un segment de longueur arbitraire entre les zones couvertes par les autre fragments de l'arbre. Il faut être capable de gérer des hypothèses sur la réalisation ou non d'une adjonction sur un nœud potentiellement adjoignable.

Les items Comme pour CYK, les items correspondent à des sous-arbres de l'arbre de dérivation. Cette même information sera exprimée dans l'item par τ, le nom de l'arbre, et p, l'indice de Gorn du nœud interne de l'arbre pour l'instant reconnu. Les hypothèses sur la réalisation d'une adjonction sur un nœud ν sont représentées par un point, qui peut être localisé sous le nœud (en position "bottom"), signifiant que l'adjonction est possible ; ou bien au dessus du nœud, signifiant que l'adjonction n'est plus autorisée (soit qu'elle ait déjà eu lieu, soit qu'on fasse l'hypothèse qu'elle n'aura pas lieu).

En plus de ces trois informations, les items contiennent des indices positionnels, en nombre de 4, qui permettent de repérer les séquences, éventuellement discontinues, de l'entrée qui sont couvertes par l'item. Un item a donc la forme suivante : $[\tau, add, pos, i, j, k, l]$, désignant un sous-arbre dont la racine est le nœud d'adresse add de τ, avant ($pos = b$) ou après ($pos = t$) adjonction et tel que :
- soit $\tau@add$ domine un nœud pied et couvre deux fragments potentiellement non-adjacents de l'entrée : entre i et $j - 1$ et entre k et $l - 1$.
- soit $\tau@add$ ne domine pas de nœud pied, impliquant que la frontière de l'arbre est un facteur du mot à analyser et donc que les indices centraux, devenus inutiles, prennent des valeurs arbitraires, conventionnellement notées par : _ ;

Avec ces notations, le succès du parsage d'un mot de longueur n correspond à l'insertion d'un item de la forme $[\tau, 0, t, 1, \_, \_, n]$, où la racine de τ est étiquetée par S.

Initialisation L'initialisation de l'analyse ascendante implique deux opérations : la reconnaissance des symboles terminaux et le démarrage d'hypothèses de "trous", dominés par les pieds d'arbres auxiliaires. Ces deux opérations correspondent respectivement aux étapes d'initialisation suivantes :

// repérage des symboles de l'entrée
foreach $\tau \in I$ **do**

if $label(\tau@p) = u_i$
then $insert([\tau, p, t, i, \_, \_, i+1]$ in T
fi
od
// seulement pour les arbres auxiliaires
foreach $\tau \in A, 1 < i < j < n$ **do**
// p est le numéro du pied de τ
$insert([\tau, p, t, i, i, j, j]$ in T
od

Déductions Nous utilisons ici l'hypothèse (aussi faite dans CYK) que tous les arbres sont au plus binaires. La complétion d'un sous-arbre résultant de la saturation de ses branches filles prend alors seulement deux formes possibles, selon que la racine du sous-arbre possède une ou deux filles :

// complétion unaire : de la fille à la mère
if $[\tau, p.1, t, i, j, k, l] \in T$
then $insert([\tau, p, b, i, j, k, l]$ in T
fi
// complétion binaire : des deux filles à la mère
if $[\tau, p.1, t, i, j, k, l] \in T \ \wedge \ [\tau, p.2, t, l, j', k', m] \in T$
then $insert([\tau, p, b, i, j \vee j', k \vee k', m]$ in T
fi

Ces deux règles de complétion créent des items correspondant à des nœuds potentiellement adjoignables à partir de nœuds saturés par rapport à l'adjonction. La règle de complétion binaire intègre une contrainte supplémentaire : elle ne peut combiner deux nœuds dominant un pied, car il y a au plus un pied par arbre. Trois cas sont possibles :
– l'arbre τ n'a pas de pied : $j = j' = k = k' = j \vee j' = k \vee k' = \_$
– le pied est à gauche $j' = k' = \_, j \vee j' = j, k \vee k' = k$
– le pied est à droite $j = k = \_, j \vee j' = j', k \vee k' = k'$
Reste à considérer maintenant l'opération d'adjonction sur un nœud encore adjoignable. Deux hypothèses sont à considérer :
– soit il n'y a pas d'adjonction, impliquant l'insertion d'une nouvelle hypothèse pour laquelle $pos = t$;
– soit il y a une adjonction, impliquant un arbre auxiliaire complètement reconnu ; ceci n'est naturellement possible que si l'adjonction de σ est possible sur le nœud $\tau@p$.
Ces deux configurations se formalisent par :

// non adjonction sur $\tau@p$
if $[\tau, p, b, i, j, k, l] \in T$
then $insert([\tau, p, t, i, j, k, l])$ in T
fi
// adjonction sur $\tau@p$
if $[\tau, p, b, i, j, k, l] \in T \ \wedge \ [\sigma, 0, t, m, i, l, n] \in T \ \wedge \ \sigma \in Adj(\tau@p)$
then $insert([\tau, p, t, m, j, k, n])$ in T
fi

La réalisation d'une adjonction correspond à l'enchâssement du sous-arbre $\tau@p$ sous le pied de σ, comblant ainsi le "trou" de σ entre les indices i et l. Le nouvel item hérite du trou éventuel dominé par le pied, s'il existe, de τ.

Complexité Quelle est la complexité de cet algorithme ? Il y a un nombre fini d'arbres dans G, impliquant un nombre fini, dépendant uniquement de la grammaire, pour les trois premières positions des items. Au total, pour une phrase de longueur n il y aura au plus $O(n^4)$ items ; pour chacun d'entre la règle d'adjonction demande de considérer de l'ordre de n^2 items pouvant potentiellement s'adjoindre, soit une complexité totale en $O(n^6)$. Malheureusement, cette complexité maximale est également la complexité moyenne, le traitement purement ascendant entraînant l'insertion puis le développement inutile de multiples items correspondant à des hypothèses portant sur des arbres auxiliaires couvrant des séquences comportant des trous de longueur variable. Comme pour le parsage des CFG, il est tout à fait possible d'ajouter des mécanismes de contrôle descendant, qui vont limiter la prolifération de ces hypothèses, donnant lieu à un parsage "à la Earley" des TAGs (Joshi and Schabes, 1997).

Bibliographie

Aho, A. V. (1968). Indexed grammars : an extension of context-free grammars. *Journal of the ACM*, 15(4) :647–671.

Denning, P. J., Dennis, J. B., and Qualitz, J. E. (1978). *Machine, Languages and Computation*. Prentice-Hall, Inc, Englewood Cliffs, NJ.

Earley, J. C. (1970). An efficient context-free parsing algorithm. *Communication of the ACM*, 13(2) :94–102.

Grune, D. and Jacob, C. J. (1990). *Parsing Techniques : a practical Guide*. Ellis Horwood.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to automata theory, languages and computation*. Addison-Wesley.

Joshi, A. K. (1985). Tree adjoining grammars : how much context-sensitivity is required to provide reasonable structural descriptions ? In Dowty, D., Karttunen, L., and Zwicky, A. M., editors, *Natural Language Parsing : psychological, computational and theoritical perspectives*, pages 206–250. Cambridge University Press.

Joshi, A. K., Levy, L. L., and Takahashi, M. (1975). Tree adjunct grammars. *Computer and System Sciences*, 10(1) :136–163.

Joshi, A. K. and Schabes, Y. (1997). Tree adjoining grammars. In Rozenberg, G. and Salomaa, A., editors, *Handbook of formal languages*, volume 3, pages 69–123. Springer-Verlag, Berlin.

Sakarovitch, J. (2003). *Eléments de théorie des automates*. Vuibert, Paris.

Salomaa, A. (1973). *Formal Languages*. Academic Press.

Schabes, Y. and Waters, R. (1995). Tree insertion grammar : a cubic-time parsable formalism that lexicalizes context-free grammar without changing the trees produced. *Computational Linguistics*, 21(4) :479–515.

Shieber, S. M., Schabes, Y., and Pereira, F. C. N. (1994). Principles and implementation of deductive parsing.

Sikkel, K. and Nijholt, A. (1997). Parsing of context-free languages. In Rozenberg, G. and Salomaa, A., editors, *Handbook of formal languages*, volume 2, pages 62–100. Springer-Verlag, Berlin.

Sudkamp, T. A. (1997). *Languages and Machines*. Addison-Wesley.

Tomita, M. (1986). *Efficient parsing for natural language*. Kluwer Academic Press.

Valiant, L. (1975). General context free recognition in less than cubic time. *Journal of Computer System Science*, 10 :310–315.

Vijay-Shanker, K. and Joshi, A. K. (1985). Some computational properties of tree adjoining grammars. In *Proceedings of the 23rd Annual Meeting of the ACL*, pages 82–93, Chicago, IL.

Vijay-Shanker, K. and Weir, D. (1994). Parsing some constraint grammar formalisms. *Computational Linguistics*, 19(4) :591–636.

Younger, D. H. (1967). Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2) :189–208.