Université Paris-Sud
Licence d'Informatique

Informatique Théorique : Théorie des Langages, Analyse Lexicale, Analyse Syntaxique

Jean-Pierre Jouannaud
Professeur

Adresse de l’auteur :

LIX
École Polytechnique
F-91128 Palaiseau CEDEX

URL : http://www.lix.polytechnique.fr/

Table des matières

Table des matières

Chapitre 1

Introduction

Ce cours se propose d'étudier en détail la notion de langage formel, initialement introduite par Noam Chomsky et son équipe dans le but de formaliser les langues naturelles. Si l'application aux langues naturelles a révolutionné leur étude, le but ultime consistant à automatiser le traitement de ces langues a largement échoué, au jour d'aujourd'hui en tout cas. Par contre, les travaux qui en ont résulté ont trouvé leur application naturelle dans le traitement automatisé des langages informatiques : la compilation, généralement découpée en analyse lexicale, analyse syntaxique, analyse sémantique et génération de code est une suite de traitements langagiers de complexité croissante : l'analyse lexicale met en oeuvre les traitements les plus simples, qui relèvent des langages dits réguliers ; l'analyse syntaxique a pour but d'analyser la structure syntaxique des phrases, qui relève des langages dits "hors-contxte" ; enfin, les traitements dits sémantiques, comme le typage, mettent en jeu des structures langagières complexes, dites contextuelles. On retrouve également la notion de langage en théorie de la calculabilité, et en théorie de la complexité. On la retrouve enfin au coeur de l'un des grands succès de la discipline, la vérification de programmes, qui a réussi ces dernières années une percée industrielle remarquée.

Le but de ce cours est de faire le tour de ces différentes notions dans le contexte informatique : les automates finis et les expressions rationnelles, qui se sont révélés un outil de modélisation exceptionnel et ont, à ce titre, envahi tout le paysage scientifique, informatique, mathématiques, physique, biologie, etc ; leur application à l'analyse lexicale ; les grammaires hors-contexte et les automates à pile, autres outils de modélisation très commode dont les utilisations ont moins débordé la sphère informatique ; leur application à l'analyse syntaxique.

Comme tout cours, celui-ci fait de nombreux emprunts, en particulier à [2], ouvrage remarquable dont une lecture approfondie est recommandée.

Chapitre 2

Langages formels

On appellera *vocabulaire* un ensemble fini V_t dont les éléments sont appellés *lettres*, et *mot* toute suite finie de lettres, notée $u = u_1 \dots u_n$ pour un mot u de *longueur* n. On notera par VT^* l'ensemble des mots, et par ε le mot de longueur nulle, appellé *mot vide*.

L'ensemble des mots est muni d'une opération interne, le *produit de concaténation*, telle que si $u = u_1 \dots u_n$ et $v = v_1 \dots v_p$, alors le produit uv est le mot w tel que $w_i = u_i$ pour $i \in [1..n]$ et $w_{n+j} = v_j$ pour $j \in [1..p]$. Il est aisé de vérifier que la concaténation des mots est associative et possède ε pour élément neure. On notera parfois le produit sous la forme $u \cdot v$ afin d'éviter certaines ambiguités.

Tout produit, par répétition, donne naissance à une notion de puissance. La puissance nème d'un mot est définie par récurrence sur n comme suit : $u^0 = \varepsilon$ et $u^{n+1} = u \cdot u^n$.

Muni du produit de concaténation et de son élément neutre, V_t^* est appellé le *monoïde libre* sur V_t. Cette dénomination fait référence au fait que tout mot u est soit le mot vide ε, soit commence par une certaine lettre a auquel cas il s'écrit de manière unique sous la forme av pour un certain mot v de taille diminuée de une unité. Il sera donc possible de prouver une propriété P d'un ensemble de mots en montrant $P(\varepsilon)$, puis en montrant $P(av)$ en supposant $P(v)$. Il s'agit là d'un raisonnement par récurrence classique lié à cette décomposition des mots. Mais on peut aussi décomposer un mot u en exhibant sa dernière lettre : tout mot u est soit le mot vide ε soit s'écrit de manière unique sous la forme va où $a \in V_t$ et $v \in V_t^*$, ce qui fournit un autre principe de récurrence. Nous utiliserons l'un ou l'autre suivant les cas.

Toute partie de V_t^* est appellée *langage*. On disnguera deux langages très particuliers, le *langage vide* noté $\emptyset$ qui ne possède aucun mot, et le *langage unité* $\{\varepsilon\}$ réduit au mot vide.

On définit à nouveau des opérations sur les langages, opérations ensemblistes classiques ou opérations qui font référence aux opérations correspondantes sur les mots :

$L_1 \cup L_2$	désigne l'union des langages L_1 et L_2
$L_1 \cap L_2$	désigne l'intersection des langages L_1 et L_2
$\overline{L}$	désigne le complémentaire dans V_t^* du langage L
$L_1 \times L_2 = \{uv \mid u \in L_1 \text{ et } v \in L_2\}$	désigne le produit des langages L_1 et L_2
L^n tel que $L^0 = \{\varepsilon\}$ et $L^{n+1} = L \times L^n$	désigne la puissance nème du langage L
$L^* = \bigcup_{i \in \mathbb{N}} L^i$	désigne l'itéré du langage L
$L^+ = \bigcup_{i > 0} L^i$	désigne l'itéré strict du langage L

Le lecteur est invité à montrer les propriétés suivantes :

1. Pour tout mot u et tous entiers n, m, $u^{n+m} = u^{m+n} = u^n \cdot u^m = u^m \cdot u^n$.
2. Pour tout langage L et tous entiers n, m, $L^{n+m} = L^{m+n} = L^n \times L^m = L^m \times L^n$.
3. Si $\varepsilon \in L$, alors $L^* = L^+$.

Chapitre 3

Automates de mots finis

Les automates sont un outil de modélisation fondamental, qui servent à représenter des dispositifs automatiques, par exemples des systèmes réactifs, tout autant que des objets mathématiques ou physiques. Dans ce capitre, nous nous intéressons aux automates de mots finis, le cas important des mots infinis étant abordé dans un chapitre ultérieur.

3.1 Automates déterministes

Définition 3.1 *Un* automate fini déterministe $\mathcal{A}$ *est un triplet* (V_t, Q, T) *où*

1. V_t *est le* vocabulaire *de l'automate ;*
2. Q *est l'ensemble fini des* états *de l'automate ;*
3. $T : Q \times V_t \to Q$*, est une application partielle appellée* fonction de transition *de l'automate.*

On notera $q \xrightarrow{a} q'$ pour $T(q, a) = q'$. Lorsque T est totale, autrement dit s'il y a dans chaque état exactement une transition pour toute lettre de l'alphabet, l'automate est dit *complet*. On omettra souvent le qualificatif fini.

Définition 3.2 *Étant donné un automate déterministe* $\mathcal{A} = (V_t, Q, T)$*, on appelle* calcul *toute suite (éventuellement vide) de transitions* $q_0 \xrightarrow{\alpha_1} q_1 \cdots q_{n-1} \xrightarrow{\alpha_n} q_n$*, aussi notée* $q_0 \xrightarrow{w}_{\mathcal{A}} q_n$*, ou encore simplement* $q_0 \xrightarrow{w} q_n$ *en l'absence d'ambiguités, issue d'un état* q_0 *et atteignant un état* q_n *en ayant lu le mot* $w = \alpha_1 \cdots \alpha_n$*.*

Notons que le calcul produit par la lecture d'un mot w par un automate fini déterministe est automatique et se trouve donc exempte d'ambiguïté : la lecture des lettres composant le mot provoque des transitions bien définies jusqu'à être bloqué en cas de transitions manquantes, ou bien jusqu'à atteindre un certain état après la lecture complète du mot. On en déduit la propriété fondamentale des automates déterministes complets :

Lemme 3.3 *Soit* $\mathcal{A} = (V_t, Q, T)$ *un automate fini déterministe complet. Alors, pour tout mot* $u \in V_t^*$ *et tout état* $q \in Q$*, il existe un unique état* $q' \in Q$ *tel que* $q \xrightarrow{u}_{\mathcal{A}} q'$*.*

En pratique, il peut être utile de rendre un automate complet en ajoutant un nouvel état appelé *poubelle*, étiqueté par $\perp$, vers lequel vont toutes les transitions manquantes. Les calculs bloquants aboutissent alors dans la poubelle où ils restent capturés.

Définition 3.4 *Un automate déterministe vient avec la donnée*

- *d'un* état initial $i \in Q$ *;*
- *d'un ensemble d'*états acceptants $F \subseteq Q$ *;*

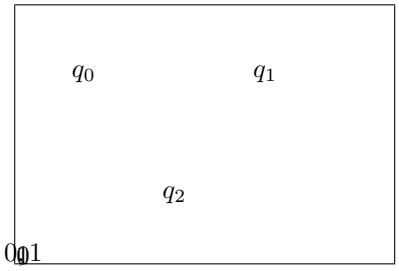


FIG. 3.1 – Automate déterministe reconnaissant les entiers naturels en numération binaire.

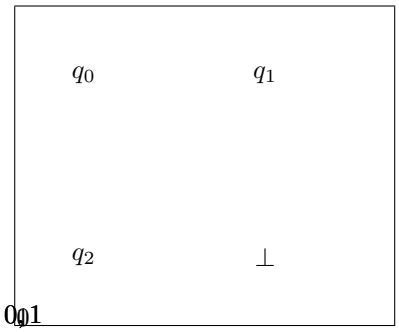


FIG. 3.2 – Automate déterministe complet reconnaissant les entiers naturels en numération binaire.

On notera par $\mathcal{A}_i^F$ le quintuplet (V_t, Q, i, F, T), ou plus simplement $\mathcal{A}$ si i et F sont donnés sans ambiguité dans le contexte.

Un mot w est reconnu *par l'automate $\mathcal{A}_i^F$ s'il existe un calcul dit* réussi *issu de l'état initial i et terminant en état acceptant après avoir lu le mot w. On note par $\mathcal{L}ang(\mathcal{A})$ le langage des mots reconnus par l'automate $\mathcal{A}$.*

Un langage reconnu par un automate est dit reconnaissable*. On note par $\mathcal{R}ec$ l'ensemble des langages reconnaissables.*

Il est clair que l'ajout d'une poubelle ne change pas les mots reconnus.

On a l'habitude de dessiner les automates, en figurant les états par des cercles, en indiquant l'état initial par une flèche entrante, les états acceptants par un double cercle ou une flèche sortante, et la transition de l'état q à l'état q' en lisant la lettre α par une flèche allant de q vers q' et étiquetée par α. La figure 3.1 représente un automate (incomplet) reconnaissant le langage des entiers naturels en représentation binaire.

L'automate obtenu reconnaît exactement le même langage si l'on convient que le nouvel état poubelle n'est pas acceptant. Ainsi, la figure 3.2 présente un automate complet reconnaissant le langage des entiers naturels en représentation binaire.

Si l'automate $\mathcal{A}$ est complet, on pourra donc étendre l'application T aux mots, en posant $T(q, u) = q' \in Q$ tel que $q \xrightarrow{u} q'$. On peut donc dans ce cas reformuler la condition d'acceptation des mots comme $u \in \mathcal{L}ang(\mathcal{A})$ ssi $T(i, u) \in F$.

3.2 Automates non déterministes

Définition 3.5 *Un* automate non-déterministe *$\mathcal{A}$ est un triplet (V_t, Q, T) où*

1. *V_t est le* vocabulaire *de l'automate ;*

FIG. 3.3 – Arbre des calculs d'un automate non déterministe.

2. *Q est l'ensemble des* états *de l'automate ;*
3. *$T : Q \times V_t \to \mathcal{P}(Q)$, est la* fonction de transition *de l'automate.*

On notera comme précédemment $q \xrightarrow{\alpha} q'$ pour $q' \in T(q, \alpha)$ avec $\alpha \in V_t$, et par $T(q, u)$ l'ensemble (peut-être vide) des états atteignables depuis q en lisant le mot u.

Notons qu'un automate déterministe est un cas particulier d'automate non-déterministe qui associe à tout élément de $Q \times V_t$ une partie de Q possédant zéro ou un élément (exactement un si c'est un automate complet). On pourra dire d'un automate déterministe qu'il est incomplet s'il peut se bloquer, c'est-à-dire s'il existe un état q et une lettre a tels que $T(q, a) = \emptyset$. Si l'automate est complet, nous aurons une forme affaiblie du Lemme 3.3 :

Lemme 3.6 *Soit $\mathcal{A} = (V_t, Q, i, F, T)$ un automate fini non-déterministe complet. Alors, pour tout mot $u \in V_t^*$ et tout état $q \in Q$, il existe un (non nécessairement unique) état $q' \in Q$ tel que $q \xrightarrow{u}_{\mathcal{A}} q'$.*

La notion de calcul est la même pour cette nouvelle variété d'automates que pour les automates déterministes, ainsi que la notion de reconnaissance, nous ne les répètons pas. Le non-déterministe a toutefois une conséquence essentielle : il peut y avoir plusieurs calculs issus de l'état initial i qui lisent un mot w donné, dont certains peuvent se bloquer et d'autres pas, et dans ce dernier cas certains peuvent terminer en état acceptant et d'autres pas. L'acceptation a lieu s'il existe au moins un calcul réussi. Si tous les calculs échouent, il n'y aura pas d'autre moyen de le savoir que de les explorer tous avant de savoir que le mot w n'est pas reconnu. La bonne notion n'est donc pas celle de calcul, mais d'arbre de calcul associé à un mot lu par l'automate :

Définition 3.7 *Étant donné un automate non déterministe $\mathcal{A}$, l'arbre de calcul de racine $q \in Q$ engendré par la lecture du mot u est un arbre doublement étiquetté défini par récurrence sur u :*

Si $u = \varepsilon$, alors l'arbre est réduit à sa racine q ;

Si $u = av$, la racine q possède des transitions sortantes étiquettées par $a \in V_t$ vers différents arbres de calcul engendrés par la lecture de v et dont les racines sont étiquettées par les états de $T(q, a)$.

Un arbre de calcul est représenté à la figure 3.3.

On a la propriété évidente :

Lemme 3.8 *Un mot $u \in V_t^*$ est reconnu par un automate non déterministe $\mathcal{A}$ ssi l'une des feuilles de son arbre de calcul est étiquettée par un état acceptant.*

Déterminisation d'un automate non-déterministe Nous allons maintenant déterminiser un automate non-déterministe (sans transitions vides) en ajoutant de nouveaux états : si Q est l'ensemble des états d'un automate non-déterministe, l'ensemble des états de l'automate déterministe associé sera $\mathcal{P}(Q)$, l'ensemble des parties de Q.

Définition 3.9 *Soit $\mathcal{A} = (V_t, Q, T)$ un automate non-déterministe. On définit $\mathcal{D}et(\mathcal{A})$ comme l'automate $(V_t, \mathcal{P}(Q), T_{det})$ où $T_{det}(K, a) = \bigcup_{q \in K} T(q, a)$.*

Théorème 3.10 *Soit $\mathcal{A}$ un automate non-déterministe. Alors $\mathcal{D}et(\mathcal{A})_{\{i\}}^{\{K \in \mathcal{P}(Q) \mid K \cap F \neq \emptyset\}}$ est déterministe et reconnaît le même langage que $\mathcal{A}_i^F$.*

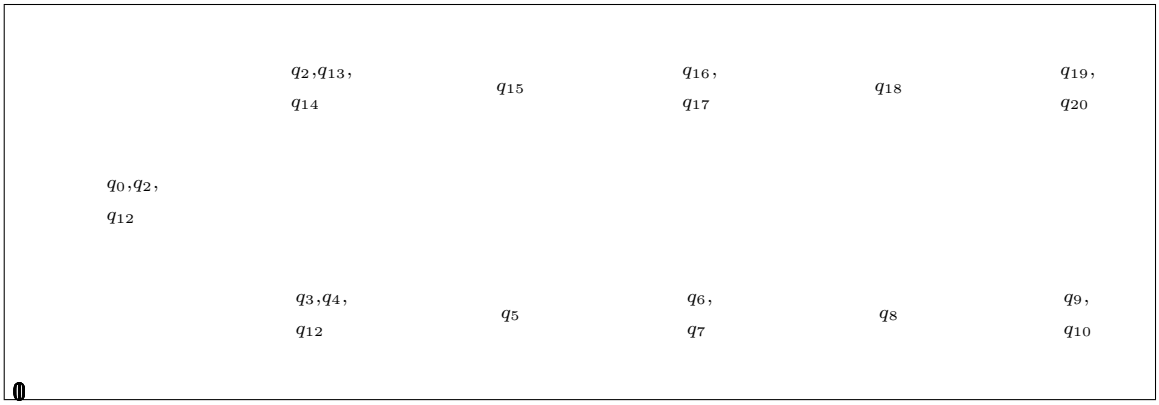


FIG. 3.4 – Automate déterminisé.

Si la construction peut sembler complexe au néophyte, la preuve est par contre très simple, et basée sur l'idée que s'il est possible dans l'automate non-déterministe d'atteindre les états $q_1, \ldots, q_n$ depuis l'état q en lisant la lettre a, alors il sera possible dans l'automate déterminisé d'atteindre un état contenant la partie $\{q_1, \ldots, q_n\}$ depuis tout état contenant l'état $\{q\}$ en lisant cette même lettre a. En fait, l'idée de la preuve est parfaitement décrite par la figure 3.3.

L'automate déterminisé correspondant à l'automate de la figure 3.6 est représenté sur la figure 3.4, où seulement les états accessibles sont montrés. Les états sont étiquetés par l'ensemble des noms des états de l'automate non-déterministe qui le constituent.

On conclue de ces développements que les automates déterministes, les automates non déterministes et les automates non déterministes avec transitions vides reconnaissent exactement les même langages. La différence est que les automates non déterministes sont exponentiellement plus économes, la borne étant atteinte comme le montre l'un des exercices. Ils constituent pour cette raison un outil précieux pour l'utilisateur. Ils sont par contre d'une manipulation informatique plus complexe causée par leur non-déterminisme.

3.3 Automates non déterministes avec transitions vides

Un automate avec transitions vides est un automate non-déterministe où certaines transitions peuvent être étiquetées par ε, qui dénotera l'absence de lettre lue lors de la transition.

Définition 3.11 *Un* automate non-déterministe $\mathcal{A}$ avec transitions vides *est un triplet* (V_t, Q, T) *où*

1. V_t *est le* vocabulaire *de l'automate ;*
2. Q *est l'ensemble des* états *de l'automate ;*
3. $T : Q \times (V_t \cup \varepsilon) \to \mathcal{P}(Q)$ *est la* fonction de transition *de l'automate.*

On notera $q \xrightarrow{\alpha} q'$ *pour* $q' \in T(q, \alpha)$, *avec* $\alpha \in V_t$ *ou* $\alpha = \varepsilon$.

Par exemple, la figure 3.5 représente un automate non-déterministe avec transitions vides reconnaissant le langage sur l'alphabet $\{0, 1\}$ contenant exactement une occurrence de chacun des mots 00 et 11.

Notons que la notation $q \xrightarrow{\alpha} q'$ devient ambigue, puisqu'elle prend maintenant deux significations différentes suivant que α est considérée comme une lettre (ou comme le symbole ε) et il y aura alors une transition unique, ou comme un mot (de longueur 1 ou 0), et il pourra alors y avoir un nombre arbitraire de transitions (dont au plus une sera non-vide).

À nouveau, les notions de calcul et de reconnaisance sont inchangées, et celle d'arbre de calcul ne nécessite qu'une adaptation triviale, du fait qu'il existe maintenant des transitions étiquettées par

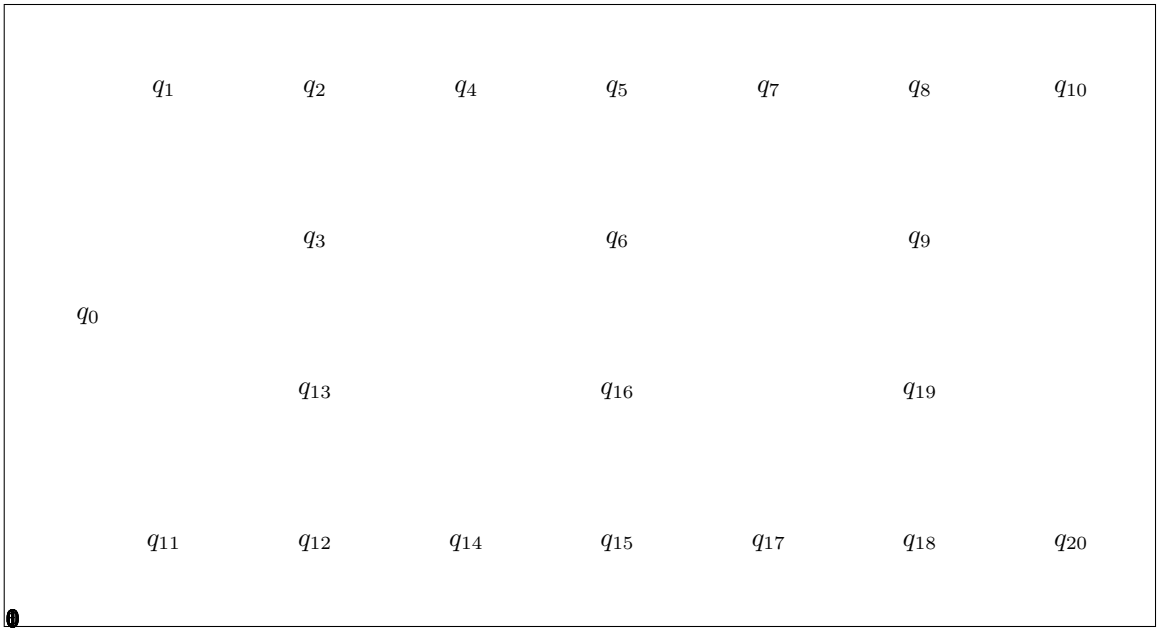


FIG. 3.5 – Un automate non-déterministe avec transitions vides.

ε. Les calculs d'un automate avec transitions vides autorisent le passage par un nombre quelconque de transitions vides au cours de l'exécution. Le nombre d'états parcourus ne sera donc plus égal au nombre de lettres du mot d'entrée mais pourra être beaucoup plus grand. L'arbre de calcul d'un automate avec transitions vides s'avère du coup être un outil moins intéressant que pour les automates non-déterministes sans transitions vides.

Élimination des transitions vides d'un automate Nous voulons maintenant montrer que le langage reconnu par un automate avec transitions vides peut également l'être par un automate non-déterministe sans transitions vides. Cette opération va nécessiter l'addition de nouvelles transitions dans l'automate :

Définition 3.12 *Soit $\mathcal{A} = (V_t, Q, i, F, T)$ un automate non-déterministe avec transitions vides. On définit $\mathcal{NDet}(\mathcal{A})$ comme l'automate $(V_t, Q, \{i\}, F_{\mathcal{NDet}(\mathcal{A})}, T_{\mathcal{NDet}(\mathcal{A})})$ où*
$q \xrightarrow{a}_{\mathcal{NDet}(\mathcal{A})} q'$ *ssi* $q \xrightarrow{\varepsilon}_{\mathcal{A}} q'' \xrightarrow{a}_{\mathcal{A}} q'$
$F_{\mathcal{NDet}(\mathcal{A})} = \{q \mid q \xrightarrow{\varepsilon}_{\mathcal{A}} f \in F\}$

Dans la définition ci-dessus, la lettre a est considérée comme une lettre, alors que le symbole ε est considéré comme un mot de longueur nulle (aussi bien dans $i \xrightarrow{\varepsilon}_{\mathcal{A}} q''$ que dans $q \xrightarrow{\varepsilon}_{\mathcal{A}} f \in F$). Notons que cette construction n'augmente pas la complexité de l'automate définie comme son nombre d'états.

Théorème 3.13 *Soit $\mathcal{A}$ un automate non-déterministe avec transitions vides. Alors $\mathcal{NDet}(\mathcal{A})_i^F$ est un automate non-déterministe qui reconnaît le même langage que $\mathcal{A}_i^F$.*

La preuve est simple, et procède par découpage d'un calcul allant de $i = q_0$ à $q_n = f \in F$ dans l'automate $\mathcal{A}$ en tronçons de la forme $q_i \xrightarrow{\varepsilon \in V_t^*}_{\mathcal{A}} \xrightarrow{a \in V_t}_{\mathcal{A}} q_{i+1}$ suivis d'un dernier tronçon de la forme $q_{n-1} \xrightarrow{\varepsilon \in V_t^*}_{\mathcal{A}} f \in F$.

Appliquée à l'automate de la figure 3.5, la construction précédente nous donne l'automate non-déterministe de la figure 3.6. Les états q_1 et q_{11} ont été supprimés car non accessibles.

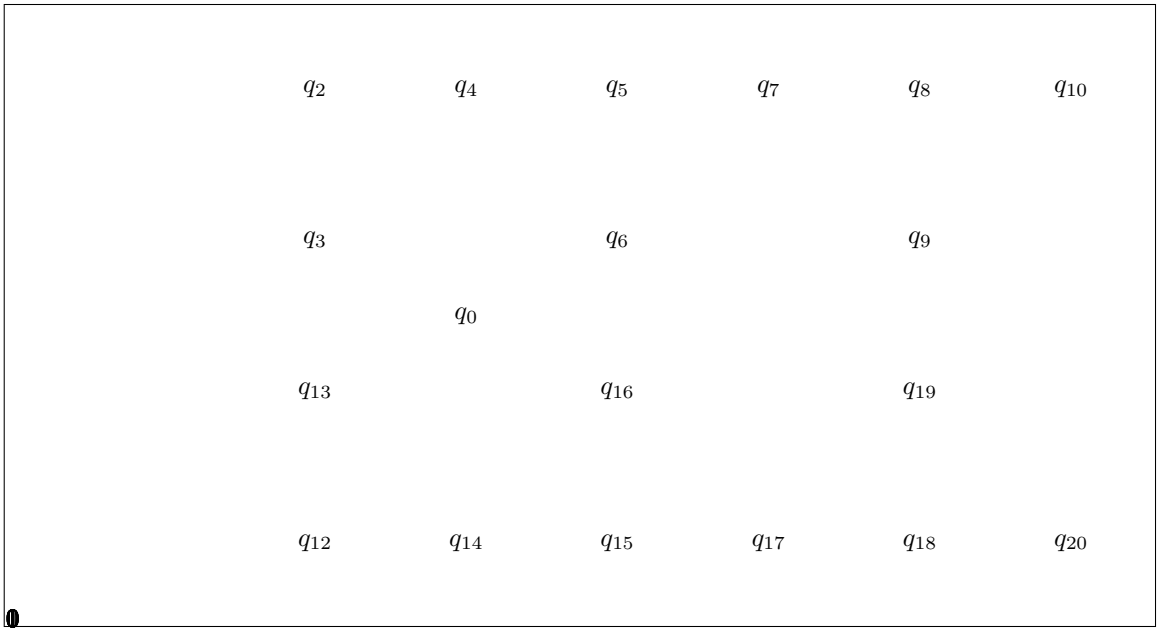


FIG. 3.6 – Automate non-déterministe sans transitions vides.

3.4 Minimisation d'un automate déterministe

Dans tout ce paragraphe, il est fondamental de supposer les automates *déterministes complets*, même si leur représentation en graphe correspond parfois à un automate incomplet par souci de clarté des dessins. On supposera également que tous les états des automates manipulés sont accessibles.

La déterminisation d'un automate non-déterministe fait exploser le nombre d'états, mais tous les états de $\mathcal{P}(Q)$ ne sont pas utiles à la construction du déterminisé d'un automate dont Q est l'ensemble des états. De plus, il se trouve que l'automate construit est en fait souvent plus compliqué que nécessaire. En particulier, il contient généralement des états inaccessibles. L'élimination des états inaccessibles n'est cependant pas toujours suffisante pour obtenir l'automate le plus simple possible : si deux états d'un automate reconnaissent le même langage, on peut *réduire* l'automate en les confondant. Le problème, bien sûr, va être de déterminer si deux états q et q' reconnaissent le même langage, de manière à partitionner l'ensemble des états afin de confondre les états d'une même classe.

3.4.1 Automate minimal

La première question, bien sûr est celle de l'existence et de l'unicité d'un automate minimal pour tout langage reconnaissable. C'est l'objet du développement qui suit.

Définition 3.14 *On dit qu'une relation d'équivalence $\simeq$ sur les mots d'un vocabulaire V est*

- d'index fini *si elle partitionne l'ensemble des mots en un nombre fini de classes,*
- une congruence droite *si* $\forall u, v \in V^* \ \forall a \in V \ u \simeq v \Longrightarrow ua \simeq va$.

Lemme 3.15 *Soient $\sim$ une congruence droite d'index fini sur V_t^*, Q l'ensemble de ses classes d'équivalence, et $F \subseteq Q$. Alors l'*automate quotient $\mathcal{A} = (V_t, Q, [\varepsilon], F, T)$ *avec $T([u], a) = [ua]$ est un automate fini déterministe complet qui reconnait le langage des mots appartenant à une classe de F.*

Preuve: Comme $\sim$ est d'index fini, Q est fini. Comme $\sim$ est une congruence droite, la définition de T ne dépend pas du choix de u dans $[u]$, donc T est bien une fonction totale. Donc $\mathcal{A}$ est un automate fini déterministe complet.

Reste à montrer qu'il reconnait le langages des mots appartenant à une classe de F. Pour cela, nous montrons par récurrence sur le mot v la propriété :

$$\forall v \in V_t^* \; \forall [u] \in Q \; [u] \xrightarrow{v}_{\mathcal{A}} [uv]$$

Si $v = \varepsilon$, la propriété est trivialement vraie.
Si $v = aw$, alors $[u] \xrightarrow{a}_{\mathcal{A}} [ua]$ par définiton de T, et $[ua] \xrightarrow{w}_{\mathcal{A}} [uaw]$ par hypothèse de récurrence, d'où la propriété.

Appliquant maintenant cette propriété pour $u = \varepsilon$, on obtient $\forall v \in V_t^* \; [\varepsilon] \xrightarrow{v}_{\mathcal{A}} [v]$, et donc v est reconnu ssi il appartient à une classe de F. □

Définition 3.16 *Soit L reconnaissable par un automate déterministe complet $\mathcal{A} = (V_t, Q, i, F, T)$, et $\sim_{\mathcal{A}}$ l'*équivalence de transition *engendrée sur les mots de V définie par $u \sim_{\mathcal{A}} v$ ssi $T(i,u) = T(i,v)$.*

Lemme 3.17 *L'équivalence de transition $\sim_{\mathcal{A}}$ d'un automate fini $\mathcal{A}$ est une congruence droite d'index fini dont les classes sont en correspondance biunivoque avec les états de $\mathcal{A}$. L'automate quotient associé aux classes de F satisfait $\mathcal{A}_{\sim_{\mathcal{A}}} = \mathcal{A}$.*

Preuve: Soit $u \sim_{\mathcal{A}} v$. Alors $T(i,u) = T(i,v)$ et donc pour tout $w \in V^*$, $T(i,uw) = T(T(i,u),w)) = T(T(i,v),w)) = T(i,vw)$, et par définition, $uw \sim_{\mathcal{A}} vw$. Comme les classes sont en correspondance biunivoque avec les états, la congruence est d'index fini et $\mathcal{A}_{\sim_{\mathcal{A}}} = \mathcal{A}$. □

Définition 3.18 *Étant donné un langage L sur V_t et un mot $u \in V_t^*$, on appelle :*
contexte à droite *de u dans v l'ensemble $D_L(u) = \{v \mid uv \in L\}$;*
congruence syntaxique *de L l'équivalence $u \sim_L v$ ssi $D_L(u) = D_L(v)$.*

Lemme 3.19 *La congruence syntaxique d'un langage quelconque est une congruence droite.*

Preuve: Soit $u \sim_L v$, donc $D_L(u) = D_L(v)$. Soit $a \in V_t$. $D_L(ua) = \{w \in V_t^* \mid aw \in D_L(u)\} = \{w \in V_t^* \mid aw \in D_L(v)\} = D_L(va)$. Donc $ua \sim_L va$, prouvant que $\sim_L$ est une congruence droite. □

Théorème 3.20 (Myhill-Nerode) *Tout langage reconnaissable L sur un alphabet V est reconnu par l'automate $\mathcal{A}_{\sim_L}$ qui est l'automate fini déterministe minimal (en nombre d'état) reconnaissant L.*

Preuve: On se donne un automate $\mathcal{A}$ quelconque reconnaissant L. Un tel automate existe, puisque L est reconnaissable. On montre que$\sim_{\mathcal{A}}$ est un raffinement de $\sim_L$, c'est-à-dire que $u \sim_{\mathcal{A}} v \Longrightarrow u \simeq_L v$. Cela aura pour corrollaire que l'automate associé à la congruence syntaxique par les Lemmes 3.19 et 3.15 reconnait L, que son nombre d´´etats est au plus égal au nombre d'états de $\mathcal{A}$ et donc qu'il est minimal puisque $\mathcal{A}$ est quelconque.

Soit $u \sim_{\mathcal{A}} v$, et comme $\sim_{\mathcal{A}}$ est une congruence droite, pour tout $w \in V^*$, $uw \sim_{\mathcal{A}} vw$, et donc $uw \in L$ ssi $vw \in L$ puisque L est une union de classes d'équivalence de $\sim_{\mathcal{A}}$. Par définition, $u \sim_L v$. □

Il nous reste maintenant à calculer cet automate minimal unique (à renommage près des états).

3.4.2 Calcul de l'automate minimal

L'idée est de calculer l'automate minimal d'un langage reconnaissable à partir d'un automate arbitraire le reconnaissant. Pour cela, plutôt que de raisonner en terme de congruence sur les mots du langage, nous allons considérer la partition induite sur les états de l'automate, qui satisfait la propriété clé de *compatibilité* :

Définition 3.21 *Étant donné un automate déterministe complet $\mathcal{A} = (V_t, Q, i, F, T)$, une partition $\mathcal{P}$ de Q est* compatible *ssi*

1. *elle est un rafinement de la partition $\{F, Q \setminus F\}$:*

$$\forall C \in \mathcal{P}\ C \subseteq F \text{ or } C \subseteq (Q \setminus F)$$

2. *toute classe de la partition est* compatible *:*

$$\forall C \in \mathcal{P}\ \forall q, q' \in C\ \forall a \in V_t\ \exists C'\ T(q,a) \in C' \text{ ssi } T(q',a) \in C'$$

Les partitions compatibles ont bien sûr un lien étroit avec les congruences droites.

Définition 3.22 *Étant donnée une partition $\mathcal{P}$ de Q, l'*équivalence de partition *sur les mots associée à $\mathcal{P}$ est définie par $u \sim_{\mathcal{P}} v$ ssi u et v appartiennent à une même classe de $\mathcal{P}$.*

On vérifie aisément que l'équivalence d'une partition compatible est une congruence droite.

Lemme 3.23 *Soit $\mathcal{P}$ une partition compatible des états d'un automate $\mathcal{A} = (V_t, Q, i, F, T)$. Alors $\sim_{\mathcal{P}}$ est une congruence droite d'index fini et l'automate*

$$\mathcal{A}_{\mathcal{P}} = (V_t, \mathcal{P}, C_0, \{C \in \mathcal{P} \mid C \subseteq F\}, T_{\mathcal{P}})$$

C_0 est la classe qui contient l'état i

$$T_{\mathcal{P}}(C, a) = C' \text{ ssi } \exists q \in C\ \exists q' \in C' \text{ tel que } T(q,a) = q'$$

est un automate fini déterministe qui reconnait le langage $\mathcal{L}(\mathcal{A})$.

Preuve: La propriété de congruence droite découle directemnt de la compatibilité de la partition. La congruence est d'index fini puisque son nombre de classes est majoré par la taille de Q. Enfin, l'automate $\mathcal{A}_{\mathcal{P}}$ est tout simplement l'automate $\mathcal{A}_{\sim_{\mathcal{P}}}$ dont les états acceptants sont les classes associées à la partition de F : il reconnait donc le langage $L(\mathcal{A})$ d'après le Lemme 3.15. □

L'automate minimal est donc associé à la partition compatible la plus grossière, dont le nombre de classes est minimal, ce qui peut se faire classiquement par une construction itérative jusqu'à l'obtention en temps fini du point fixe. On peut pour cela procéder de deux manières différentes :

Partant d'une partition compatible, celle dont les classes contiennent exactement un état de l'automate de départ, on construit jusqu'à stabilisation une suite de partitions compatibles plus fines que la partition minimale, ce qui nous fournit cette dernière comme résultat. Pour cela, on confond à chaque étape plusieurs classes lorsque leur fusion conserve la compatibilité. La difficulté est qu'on ne peut procéder par fusion de deux classes à la fois : il faut en fait *deviner* les fusions nécessaires à chaque étape. Le processus est donc hautement non-déterministe et conduit à un algorithme exponentiel.

Partant de la partition potentiellement compatible la plus grossière, soit $(F, Q \setminus F)$, on construit jusqu'à stabilisation une suite de partitions plus grossières que la partition minimales jusqu'à obtention de cette dernière, puisque le résultat est nécessairement une partition compatible. Cet algorithme n'assure donc pas la compatibilité des partitions intermédiaires obtenues, mais il vérifie la propriété de compatibilité de chaque classe, et si elle s'avère fausse pour une classe, il la *force* en découpant cette classe en sous-classes compatibles. Cela peut bien sûr affecter la compatibilité des classes existantes, et il faut donc recommencer le processus s'il y a eu un découpage. Au pire, on s'arrêtera lorsque chaque classe contiendra un unique état de l'automate de départ, dans le cas où ce dernier était déjà minimal. L'algorithme obtenu est polynomial en le nombre d'états de l'automate.

Étant donnée une partition P des états d'un automate, la fonction suivante recherche une classe de P qui n'est pas compatible. Dans le cas où toutes les classes sont compatibles la fonction retourne $\emptyset$. Puisque tous les classes dans les partitions construites par l'algorithme sont non-vides, le résultat $\emptyset$ signale le cas d'échec de la recherche.

type classe sur Q = ensemble sur Q
type partition sur Q = ensemble sur classe sur Q
fonction *non_compatible*($\mathcal{A}$: automate sur (V_t, Q),
P : partition sur Q) : classe sur Q
% *Recherche d'une classe non compatible de* P
si il existe $C \in P$, $p \in C$, $q \in C$ et $a \in V_t$ tels que
transition$(\mathcal{A}, p, a) \in C'$ et *transition*$(\mathcal{A}, q, a) \in C''$ avec $C' \neq C''$
retourner C
sinon
retourner $\emptyset$
fin si
fin fonction

La fonction suivante calcule alors, à partir d'une partition P donnée, le *raffinement compatible le plus grossier* de P, c'est-à-dire la sous-partition compatible de P ayant le plus petit nombre possible de classes (on peut montrer qu'elle est unique) :

fonction *partition*($\mathcal{A}$: automate sur (V_t,Q),
P : partition sur Q) : partition sur Q
% *Calcul d'un raffinement compatible de la partition* P
soit $C =$ *non_compatible*$(\mathcal{A}, P)$
si $C = \emptyset$
retourner P
sinon
soit P' la partition la plus grossière de C telle que
pour tout $C' \in P'$, pour tout $C'' \in P$, pour tous $p, q \in C'$
pour tout $a \in V_t$:
transition$(\mathcal{A}, p, a) \in C''$ ssi *transition*$(\mathcal{A}, q, a) \in C''$
retourner *partition*$(\mathcal{A}, (P - \{C\}) \cup P')$
fin si
fin fonction

La fonction suivante calcule l'automate minimisé.

fonction *minimisation*($\mathcal{A}$: automate sur (V_t, Q)) : automate sur $(V_t,$ classe sur $Q)$
% *Calcul du minimisé de A*
% *Calcul de la plus grossière partition compatible*
soit $F = \{q \in Q \mid$ *est_acceptant*$(\mathcal{A}, q)\}$
$P =$ *partition*$(\mathcal{A}, \{Q - F, F\})$
% *Calcul de l'état initial de l'automate minimal*
soit I_{min} la classe de P telle que *etat_initial*$(\mathcal{A}) \in I_{min}$
% *Calcul des états acceptants de l'automate minimal*
pour chaque $C \in P$:
soit $F_{min}(C) =$ true **ssi** $C \cap F \neq \emptyset$
fin pour
% *Calcul de la fonction de transition de l'automate minimal*
pour chaque $(C, a) \in P \times V_t$ **faire**
soit $q \in C$ (un représentant quelconque)
soit C' la classe de P telle que *transition*$(\mathcal{A}, q, a) \in C'$
$T_{min}(C, a) = C'$

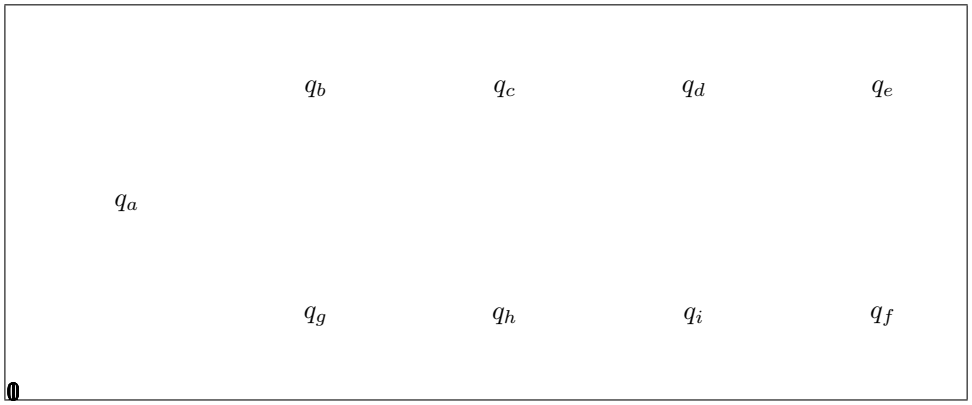


FIG. 3.7 – Automate déterminisé et minimisé.

fin pour
retourner $(I_{min}, F_{min}, T_{min})$
fin fonction

Le minimisé de l'automate de la figure 3.4 est représenté sur la figure 3.7.

Théorème 3.24 *Soit $\mathcal{A} = (V_T, Q, i, F, T)$ un automate déterministe et complet. L'automate résultat de la fonction minimisation est l'automate déterministe complet avec un nombre minimal d'états acceptant le même langage que $\mathcal{A}$.*

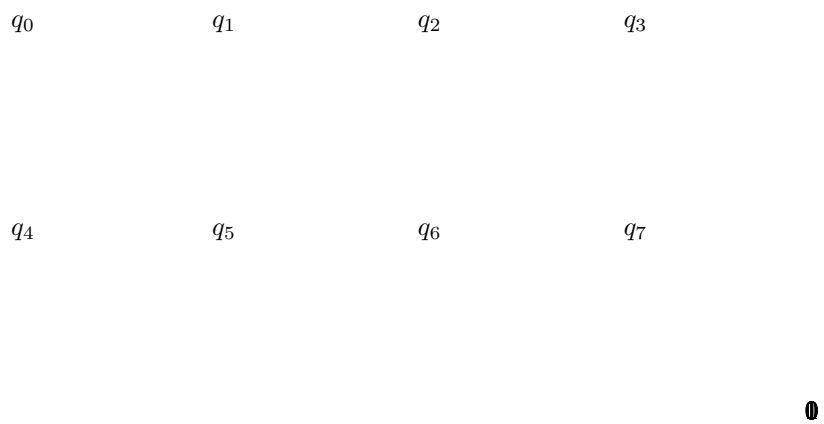
En pratique, le calcul de la partition compatible la plus grossière possible se fait à l'aide d'un tableau ayant autant de lignes et de colonnes que d'états. Il est facile, à l'aide d'un tel tableau, de marquer les états incompatibles entre eux. À l'initialisation, les états non-acceptants seront marqués comme incompatibles avec les états acceptants. Puis on marque récursivement tout couple d'états (p, q) tels que le couple $(\mathit{transition}(p, a), \mathit{transition}(q, a))$ soit marqué pour une certaine lettre $a \in V_t$. Le marquage une fois terminé, la ligne p du tableau de marquage indique les états qui sont dans la même classe de la partition que p : ce sont les états q tels que les couples (p, q) ne sont pas marqués. L'algorithme est donc :

pour chaque q ∈ Q **faire**
 pour chaque q ∈ Q **faire**
  si acceptant(p) **et non** acceptant(q)
   marquer(p,q)
  fin si
 fin pour
fin pour

répéter
 stable = true
 pour chaque q ∈ Q **faire**
  pour chaque p ∈ Q **faire**
   pour chaque a ∈ V_t **faire**
    si marqué(p,q) **et non** marqué(transition(q,a), transition(p,a))
     marquer(p,q)
     stable = false
    fin si
   fin pour
  fin pour
 fin pour
jusqu'à stable

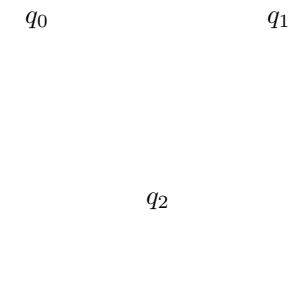
3.5 Exercices

Exercice 3.1 *Donner l'automate minimal équivalent à l'automate suivant :*



Exercice 3.2 *Donner un automate qui reconnaisse l'ensemble des entiers naturels en représentation octale.*

Exercice 3.3 *Considérons l'automate suivant sur l'alphabet* $\{a, b, c\}$*, d'état initial* q_0 *et d'états acceptants* $\{q_2\}$.



1. *Quel sont les états atteints lors de l'exécution de cet automate respectivement sur les mots* a*,* b*,* c*,* $cabc$*,* $baca$*,* $baac$*,* aa*,* bb*, et* ε *?*
2. *Parmi ces mots, quels sont ceux qui appartiennent au langage reconnu par l'automate ?*

Exercice 3.4 *1. Donner un automate déterministe qui reconnaît une date de la forme "jour/mois" où le jour est noté par un entier entre* 1 *et* 31 *et le mois est noté par un entier entre* 1 *et* 12*. Attention aux mois avec moins de 31 jours ! Par exemple, votre automate doit accepter "7/4" (pour le 7 avril) et "31/12" (pour le 31 décembre), mais pas "30/2".*

2. *Étendre l'exercice aux dates de la forme "jour/mois/année", où l'année est notée par un entier entre* 1000 *et* 1999*. On suppose que les années bissextiles sont les années qui sont divisibles par* 4*.*

Exercice 3.5 *Déterminisez l'automate* (V_t, Q, q_0, F, δ)*, défini comme suit :*

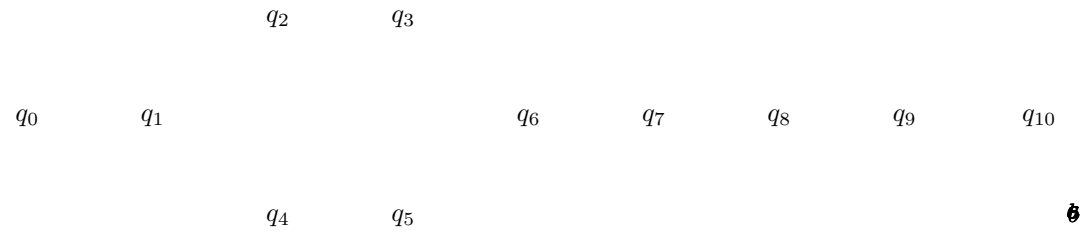
- $V_t = \{a, b\}$
- $Q = \{1, 2, 3, 4, 5\}$
- $q_0 = 1$
- $F = \{5\}$
- $\delta(1, a) = \{1\}$
- $\delta(1, b) = \{1, 2\}$
- $\delta(2, a) = \{3\}$
- $\delta(3, a) = \{4\}$
- $\delta(4, b) = \{5\}$
- $\delta(5, a) = \{5\}$
- $\delta(5, b) = \{5\}$

Exercice 3.6 *Soit l'automate :*

Quel est langage accepté par cet automate ? Le déterminiser et minimiser.

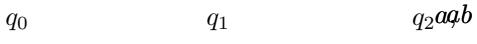
Exercice 3.7 *Construire l'automate déterministe équivalent à l'automate suivant :*

Exercice 3.8 *Soit l'automate :*



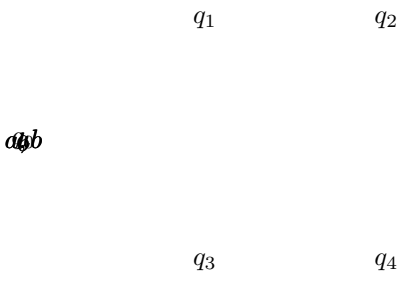
1. *supprimer les transitions vides*
2. *Le déterminiser*
3. *Est-il minimal ?*

Exercice 3.9 *Donnez la version deterministe puis minimale de l'automate suivant.*



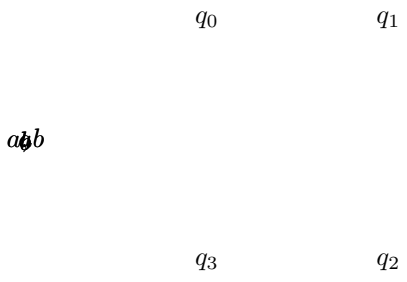
Exercice 3.10
1. *Donner un automate déterministe qui accepte le langage sur l'alphabet $\{a, b\}$ dont tous les mots admettent un a comme troisième lettre.*
2. *Donner un automate non-déterministe qui accepte le langage sur l'alphabet $\{a, b\}$ dont tous les mots admettent un a comme troisième lettre* à partir de la droite.
3. *Donner un automate déterministe pour le même langage.*

Exercice 3.11 *Considérons l'automate non-déterministe suivant :*



1. *Construire un automate déterministe équivalent.*
2. *Minimiser l'automate obtenu.*

Exercice 3.12 *Donnez la version deterministe des automates suivants.*


q_0
q_1
a,b
q_3
q_2

Chapitre 4

Nettoyage des automates

Notons tout d'abord que les automates non déterministes avec transitions vides se réduisent en des automates non déterministes sans transitions vides en temps linéaire en leur taille. Les algorithmes de complexité au moins linéaire concernant les automates non déterministes seront donc encore vrais pour les automates non déterministes avec transitions vides. Néanmoins, la pluspart des résultats exposés dans ce chapitre s'appliquent directement aux automates avec transitions vides.

Qu'il soit ou non déterministe, un automate peut posséder des états superflus, en ce sens qu'ils peuvent être tout simplement retirés sans changer le langage reconnu.

Définition 4.1 *Étant donné un automate (déterministe ou pas, avec ou sans transitions vides)* $A = (V_t, Q, i, F, T)$*, on dira que l'état* $q \in Q$ *est*

- accessible, *s'il existe un mot* w *tel que* $i \xrightarrow{w}{}^* q$ *;*
- productif, *s'il existe un mot* w *tel que* $q \xrightarrow{w}{}^* f \in F$ *;*
- utile, *s'il est à la fois accessible et productif.*

Un automate est dit réduit *si tous ses états sont utiles.*

Afin d'éliminer les états inutiles d'un automate, il faut en éliminer successivement les états inaccessibles et les états improductifs, ce que l'on appellera le *nettoyage*. L'automate obtenu sera un automate réduit équivalent à l'automate de départ, il aura pour transitions celles qui relient des états utiles de l'automate de départ. Reste à calculer les états productifs, puis les états accessibles. Pour cela, on observe qu'un état q est productif ssi $q \in F$ ou bien il existe un état productif q' et une lettre $a \in (V_t \cup \varepsilon)$ tels que $q' \in T(q, a)$. On en déduit un algorithme simple qui procède, en temps linéaire, jusqu'à stabilisation par ajouts successifs d'états productifs à un ensemble réduit initialement à F. De même, un état q est accessible ssi $q = i$ ou bien il existe un état accessible q' et une lettre $a \in (V_t \cup \varepsilon)$ tels que $q \in T(q', a)$. On en déduit un algorithme similaire qui procède, toujours en temps linéaire, jusqu'à stabilisation par ajouts successifs d'états accessibles à un ensemble réduit initialement à $\{i\}$.

Théorème 4.2 *Pour tout automate fini* $\mathcal{A}$*, déterministe ou pas, il existe un automate fini de même nature sans état inutile qui peut être obtenu à partir du premier en temps linéaire.*

4.1 Décision du vide

Déterminer si un automate reconnait le langage vide est un problème aux très nombreuses applications pratiques, nous en verrons une à la section 5.1. Le nettoyage nous permet de décider cette question en temps linéaire, puisque dans ce cas l'automate obtenu doit être vide. En fait, il n'est même pas nécessaire de nettoyer l'automate pour cela.

Lemme 4.3 *Le langage d'un automate fini $\mathcal{A}$ est non vide si et seulement si l'un au moins des états de F est accessible.*

Preuve: Il suffit d'écrire la condition d'existence d'un mot reconnu par l'automate. □

Ce lemme montre que la décidabilité du vide se ramène à l'existence de chemins joignant l'état initial à un état acceptant dans le graphe de l'automate. Comme l'existence d'un chemin se détermine en temps linéaire par des calculs d'arbres couvrants du graphe, on en déduit :

Théorème 4.4 *Le vide du langage reconnu par un automate quelconque est décidables en temps linéaire.*

4.2 Décision de la finitude

On obtient de même :

Lemme 4.5 *Le langage d'un automate $\mathcal{A}$ est infini si et seulement s'il existe un état q ayant les trois propriétés suivantes :*
(i) q est accessible ;
(ii) q est productif ;
(iii) q est situé sur un cycle non trivial (dont au moins une transition est non vide).

Une façon simple de procéder consiste donc à tester l'existence d'un cyle non trivial dans l'automate obtenu par nettoyage. La finitude s'obtient bien sûr par négation de la propriété précédente, c'est-à-dire dans le cas où il n'existe aucun état q ayant ces trois propriétés à la fois.

Ce lemme montre que la décidabilité de la finitude se ramène elle-aussi à l'existence de chemins d'une certaine forme dans le graphe de l'automate. Comme l'existence d'un chemin ou d'un cycle se détermine en temps linéaire par des calculs d'arbres couvrants du graphe, on en déduit :

Théorème 4.6 *La finitude et l'infinitude du langage reconnu par un automate quelconque sont décidables en temps linéaire.*

4.3 Décision du plein

Reste le problème du plein du langage reconnu par un automate fini. Si l'automate est déterministe, ce problème a la même complexité que le problème du vide, puisqu'ils s'échangent par échange des états acceptants et non-acceptants. Cela n'est pas vrai des automates non-déterministes.

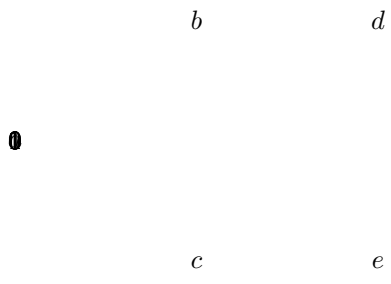
Théorème 4.7 *Le plein du langage reconnu par un automate déterministe (respectivement, non déterministe) est décidable en temps linéaire (respectivement, exponentiel).*

Le cas des automates déterministes se traite par un passage au complémentaire en temps linéaire sans changer la taille de l'automate (à un facteur près). Pour les automates non-déterministes, il est nécessaire de déterminiser au préalable, d'où le saut de complexité.

4.4 Exercices

Exercice 4.1 *Donner un algorithme pour calculer l'ensemble des états accessibles d'un automate.*

Exercice 4.2 *On considère l'automate suivant reconnaissant le langage L_0 :*



On demande de nettoyer cet automate, puis de donner le langage qu'il reconnait.

Chapitre 5

Propriétés de clôture des langages reconnaissables

Soit f une opération d'arité n sur les mots. Étant donnés les alphabets $V_1, \ldots, V_n$ et les langages L_1 sur $V_1, \ldots, L_n$ sur V_n, on définit le langage $f(L_1, \ldots, L_n) = \bigcup_{u_i \in L_i} f(u_1, \ldots, u_n)$ sur l'alphabet $V_1 \cup \ldots \cup V_n$

On dit que les langages reconnaissables sont clos par une opération f si $f(L_1, \ldots, L_n)$ est reconnaissable lorsque les langages $L_1, \ldots, L_n$ le sont.

Les langages reconnaissables possèdent de très nombreuses propriétés de clôture, en particulier par les opérations Booléennes, par substitution (des lettres dans les mots d'un langage par des mots pris arbitrairement dans un langage régulier associé à chaque lettre, le cas particulier où chaque lettre est remplacée par un mot étant appellé homomorphisme), par homomorphisme inverse, par quotient à droite, par shuffle, par pompage, etc.

Ces propriétés jouent deux rôles essentiels : elles servent à montrer que certains langages sont bien reconnaissables, lorsqu'ils sont obtenus à partir de langages reconnaissables par une opération de côture ; elles servent à contrario à montrer que certains langages ne sont pas reconnaissables, lorsqu'ils permettent d'obtenir un langage non reconnaissable par une opération de côture. La clôture par pompage joue un rôle clé pour cette dernière application.

5.1 Clôture Booléenne

Nous nous intéressons ici aux opérations Booléennes sur des langages définis par des automates les reconnaissant, de sorte que les opérations peuvent être vues comme agissant directement sur les automates.

On commence par la clôture par complémentaire :

Théorème 5.1 *Soit $A = (V_t, Q, q_0, F, T)$ un automate déterministe complet. Alors $A = (V_t, Q, q_0, Q \setminus F, T)$ reconnait le langage complémentaire de $\mathcal{L}ang(A)$.*

Ce résultat est une conséquence directe du Lemme 3.3 et de la reformulation de la notion d'acceptation qui s'en est ensuivie. Il est donc fondamental dans cet énoncé que l'automate de départ soit complet. Passons maintenant à l'union et à l'intersection :

Définition 5.2 *Étant donnés deux automates déterministes $A_1 = (V_t, Q_1, T_1)$ et $A_2 = (V_t, Q_2, T_2)$ sur le même alphabet V_t, l'automate $A = (V_t, Q_1 \times Q_2, T)$ où $T((q_1, q_2), a) = \{(T(q_1, a), T(q_2, a)\}$ est appellé* produit synchronisé *de A_1 par A_2.*

Notons que l'automate produit a pour effet de synchroniser les transitions, donc les calculs des automates de départ.

Théorème 5.3 *Soient $A = (V_t, Q, T)$ et $A' = (V_t, Q', T')$ deux automates déterministes sur le même alphabet V_t, et B leur produit synchronisé. Alors, l'automate $B^{F\times F'}_{(q_0,q'_0)}$ reconnait le langage $\mathcal{L}ang(A^F_{q_0}) \cap \mathcal{L}ang(A'^{F'}_{q'_0})$, et si les automates de départ sont complets, l'automate $B^{(F\times Q')\cup(F'\times Q)}_{(q_0,q'_0)}$ reconnait le langage $\mathcal{L}ang(A^F_{q_0}) \cup \mathcal{L}ang(A'^{F'}_{q'_0})$.*

La construction ci-dessus se généralise sans difficulté pour reconnaitre l'intersection des langages reconnus par deux automates non-déterministes sans transitions vides A et A' ne possèdant pas le même alphabet : il suffit pour cela de définir T sous la forme plus générale (mais équivalente pour les automates déterministes complets) $T((q_1, q_2), a) = \{(q'_1, q'_2) \mid q'_1 \in T(q_1, a)$ et $q'_2 \in T(q_2, a)\}$ qui fournit un automate incomplet sur l'alphabet union (mais complet sur l'alphabet intersection) qui répond à la question.

Par contre, il faut faire attention en ce qui concerne l'union des langages reconnus, car il faut disposer d'automates qui soient complets sur l'alphabet union. Il suffit pour cela de rajouter une poubelle de manière à ce qu'un automate ne se bloque pas pendant que l'autre est en train de reconnaitre. Cette construction fonctionne également pour des automates non déterministes.

Les propriétés de clôture Booléenne ont de très nombreuses applications. Par exemple, pour vérifier si un automate $\mathcal{A}$ modélisant un certain automatisme possède une propriété P, qui peut être vue comme une propriété des mots sur l'alphabet de l'automate, la technique classique consiste à construire un automate $\mathcal{A}_P$ qui accepte tous les mots (et rien que ceux là) qui satisfont la propriété P, puis à vérifier que $\mathcal{L}ang(\mathcal{A}) \subseteq \mathcal{L}ang(\mathcal{A}_P)$. Cette inclusion peut se récrire en $\mathcal{L}ang(\mathcal{A}) \cap \overline{\mathcal{L}ang(\mathcal{A}_P)} = \emptyset$, ce qui, grâce aux propriétés de clôture, revient à tester le vide de l'automate $\mathcal{A} \cap \overline{\mathcal{A}_P}$.

5.2 Clôture par produits

Théorème 5.4 *Les langages reconnaissables sont clos par produit de langages.*

Preuve: Soient L et L' deux langages reconnus respectivement par les automates $\mathcal{A} = (V_t, Q, i, F, T)$ et $\mathcal{A}' = (V_t, Q', i', F', T')$ supposés sans transitions vides. On construit l'automate $\mathcal{A}_\times = (V_t, Q \cup Q', i, F', T_\times)$, tel que

$$\forall q \in Q,\ \forall q' \in Q',\ \forall a \in V_t,\ T_\times(q, a) = T(q, a),\ T_\times(q', a) = T'(q', a),\ T_\times(f \in F, \varepsilon) = i'$$

et l'on vérifie aisément qu'il reconnait bien le langage $L \times L'$. □

Les langages reconnaissables étant clos par produit, ils le sont bien sûr par une puissance quelconque. Ils le sont même par produit itéré.

Théorème 5.5 *Les langages reconnaissables sont clos par produit itéré (on dit aussi par étoile de Kleene) et par produit itéré strict.*

Preuve: Soient L un langage reconnus par l'automate $\mathcal{A} = (V_t, Q, i, F, T)$ supposé sans transition vide. On construit tout d'abord l'automate $\mathcal{A}^+ = (V_t, Q, i, F, T^+)$, tel que

$$\forall q \in Q,\ \forall a \in V_t,\ T^+(q, a) = T(q, a),\ T^+(f \in F, \varepsilon) = i$$

et l'on vérifie aisément qu'il reconnait bien le langage L^+ par récurrence sur le nombre d'utilisations lors d'un calcul acceptant des transitions vides allant d'un état de F à l'état i.

La construction pour L^* est la même si $\varepsilon \in L$, puisque dans ce cas $L^* = L^+$, mais elle est nettement plus complexe dans le, cas général, pour lequel on définit $\mathcal{A}^+ = (V_t, Q \cup \{i', f'\}, i', \{f'\}, T^*)$, tel que

$$\forall q \in Q,\ \forall a \in V_t,\ T^*(q, a) = T(q, a),\ T^+(i' \in F, \varepsilon) = \{i, f'\},\ T^+(f \in F, \varepsilon) = \{i, f'\}$$

et l'on vérifie aisément qu'il reconnait bien le langage L^+ par récurrence sur le nombre d'utilisations lors d'un calcul acceptant des transition vide allant d'un état de F à l'état i. □

En fait la construction peut se simplifier y compris lorsque le langage L ne reconnait pas le mot vide : on peut confondre les états i et i' à condition que l'automate $\mathcal{A}$ n'ait pas de boucle sur l'état i, et on peut également "confondre" les états f avec l'état f' (ce qui aboutira à avoir une transition vide depuis chaque état acceptant vers l'état i) à condition qu'il n'ait pas de boucle sur l'état i'. Dans le cas contraire, c'est-à-dire lorsqu'il y a des boucles à la fois sur i et f, on peut encore confondre i' et f'. En pratique il est important de construire des automates aussi compacts que possible, afin de contrôler au maximum les phénomènes d'explosion combinatoire rencontrés lors des déterminisations.

5.3 Clôture par morphisme

Définition 5.6 *Étant donnée une application h de V_t dans V_t^*, on définit l'homomorphisme engendré par h, encore noté h, comme l'unique application de V_t^* dans V_t^* qui vérifie*

$$h(\varepsilon) = \varepsilon \text{ et } h(av) = h(a) \cdot h(v)$$

*L'*image par un homomorphisme h *d'un langage L est le langage*

$$h(L) = \{u \in V_t^* \mid \exists v \in V_t^* \text{ tel que } u = h(v)\}$$

Théorème 5.7 *Les langages reconnaissables sont clos par homomorphisme.*

Preuve: Le résultat se prouvera facilement dès que nous aurons défini les expressions rationnelles, par récurrence sur leur structure. □

5.4 Pompage des langages reconnaissables

Soit $m = uvw$ un mot sur le langage V_t. On dit que les mots de la forme uv^kw pour $k \in \mathbb{N}$ sont obtenus par pompage de v dans m. Les langages reconnaissables possèdent une propriété de clôture remarquable vis à vis de l'opération de pompage qui sert à contrario à montrer que certains langages qui ne la possèdent pas ne sont pas reconnaissables. Nous allons d'abord illustrer la démarche avec le langage $L = \{a^nb^n \mid n \in \mathbb{N}\}$, qui est connu pour être le langage le plus simple qui ne soit pas reconnaissable.

Supposons que L soit reconnaissable par un certain automate $\mathcal{A} = (\{a, b\}, Q, i, F, T)$. Notons $N = |Q|$, et considérons le mot a^Nb^N de L. Il est reconnu par un calcul de la forme :

$$i = q_0 \xrightarrow{a} q_1 \dots q_{N-1} \xrightarrow{a} q_N \xrightarrow{b} q_{N+1} \dots q_{2N-1} \xrightarrow{b} q_{2N} \in F$$

Comme l'automate n'a que N états, il existe deux entiers i et j tels que $0 \leq i < j \leq N$ et $q_i = q_j$. On peut donc reformuler le calcul précédent comme suit

$$i = q_0 \xrightarrow{a^i} q_i \xrightarrow{a^{j_i}} q_j \xrightarrow{a^{N_j}} q_N \xrightarrow{b^N} q_{2N} \in F$$

Comme $q_i = q_j$, le calcul

$$i = q_0 \xrightarrow{a^i} q_i = q_j \xrightarrow{a^{N_j}} q_N \xrightarrow{b^N} q_{2N} \in F$$

reconnait le mot $a^{N-(j-i)}b^N$ qui n'est pas un mot de L puisque $j - i$ est strictement positif. Notre hypothèse que le langage L est reconnaissable est donc erronée.

Le langage $L = \{a^nb^n \mid n \in \mathbb{N}\}$ n'est donc pas reconnaissable. Cela a d'importantes conséquences pour les langages de programmation, puisque cette structure langagière abonde : il s'agit ni plus ni moins du langage des parenthèses bien formées. Généraliser cet outil est donc essentiel, puisque cela va nous permettre de comprendre quels sont les langages que l'on ne peut pas traiter avec des techniques d'automates.

Théorème 5.8 *Tout langage L reconnaissable satisfait la propriété*
$Pompe \stackrel{d}{=} \exists N > 0$ *tel que* $\forall m \in L$, *si* $|m| \geq N$ *alors* $\exists u, v, w \in V_t^*$ *tels que :*
(i) $m = uvw$,
(ii) $v \neq \varepsilon$,
(iii) $|uv| < N$,
(iv) $\forall k \in Nat,\ uv^k w \in L$

Preuve: C'est le même argument que celui que l'on vient de développer pour montrer que le langage $\{a^n b^n \mid n \in \mathbb{N}\}$ n'est pas reconnaissable. On se donne un automate complet, sans transition vide $\mathcal{A}$ reconnaissant le langage L, et on note $N = |Q|$ qui est positif strictement puisque $\mathcal{A}$ est complet. Soit maintenant $m \in L$ de taille au moins N, et

$$i = q_0 \xrightarrow{m} q_{|m|} \in F$$

un calcul reconnaissant m. Comme l'automate n'a que $N < |m| + 1$ états, il existe deux entiers i et j tels que $0 \leq i < j \leq N$ et $q_i = q_j$. Il existe donc trois mots u, v, w tels que :

$$i = q_0 \xrightarrow{u} q_i \xrightarrow{v} q_j \xrightarrow{w} q_{|m|} \in F$$

et l'on vérifie immédiatement que les conditions (i), (ii) et (iii) sont satisfaites. De plus, comme $q_i = q_j$, le calcul

$$i = q_0 \xrightarrow{u} q_i \xrightarrow{v^k} q_j \xrightarrow{w} q_{|m|} \in F$$

reconnait le mot $uv^k w$ qui est donc un mot de L. □

Il faut insister sur le fait que le "Lemme de la pompe" n'est pas une condition nécessaire et suffisante : il existe des langages non reconnaissables qui satisfont le "Lemme de la pompe". On ne peut donc pas déduire d'un langage qu'il est reconnaissable en montrant qu'il satisfait le "Lemme de la pompe". Mais l'on peut s'en servir bien sûr pour montrer qu'un langage n'est pas reconnaissable, comme nous l'avons fait précédemment pour le langage des parenthèses $\{a^n b^n \mid n \in \mathbb{N}\}$, puisque, par contraposition, un langage qui ne satisfait pas le "Lemme de la pompe" ne peut pas être reconnaissable.

Exprimons donc la négation de la propriété de la pompe, cela se fait par application des règles usuelles de logique permettant de pousser les négations à l'intérieur des formules en changeant les quantificateurs, et par application également du théorème de déduction (permettant de prouver la propriété $A \Longrightarrow B$ en prouvant B avec un ensemble d'hypothèses augmenté de A). On obtient :
$\neg Pompe \stackrel{d}{=} \forall N > 0,\ \exists m \in L,\ |m| \geq N$ tel que $\forall u, v, w \in V_t^*$ satisfaisant
(i) $m = uvw$,
(ii) $v \neq \varepsilon$,
(iii) $|uv| < N$,
alors $\exists k \in Nat$ tel que $uv^k w \notin L$.

La preuve faite au début de ce paragraphe nous a montré que le langage des parenthèses satisfait la propriété $\neg Pompe$, et qu'il n'est donc pas un langage reconnaissable. Mais comment faire avec un langage qui satisfait le théorème de la pompe ? L'idée cette fois est de combiner la propriété de la pompe avec une autre propriété de clôture.

Considérons le langage $L = \{c^m a^n b^n \mid m > 0, n \geq 0\} \cup a^* b^*$, et montrons que ce langage satisfait la propriété $Pompe$. Prenons $N = 3$, et choisissons $u = \varepsilon$, v la première lettre du mot, et w le reste du mot. Si l'on pompe dans un mot de la forme $c^m a^n b^n$, on obtiendra un mot de la même forme, ou bien le mot $a^n b^n$ qui fait partie du langage $a^* b^*$. Si l'on pompe un mot de $a^* b^*$, on obtiendra encore un mot de $a^* b^*$ car on ne pompe qu'une seule lettre. Donc L vérifie la propriété de la pompe.

Soit maintenant $K = c^+ a^* b^*$ un langage rationnel, et h un homomorphisme tel que $h(a) = a, h(b) = b$ et $h(c) = \varepsilon$. On remarque alors que $h(L \cap K) = \{a^n b^n \mid n \geq 0\}$ qui n'est pas reconnaissable, et donc L non plus, puisque $\{a^n b^n \mid n \geq 0\}$ est obtenu à partir de L en utilisant des propriétés de clôture.

Chapitre 6

Expressions rationnelles

Les automates sont le véhicule idéal pour éffectuer des calculs sur les langages reconnaissables. Mais ils ne sont pas très commodes à la communication, en particulier avec la machine, à moins de les dessiner.

On va donc introduire une notation plus pratique, et en même temps très puissante, les expresions rationnelles.

6.1 Expressions rationnelles

Définition 6.1 *On définit le langage $\mathcal{R}at$ des* expressions rationnelles *sur l'alphabet V_t par récurrence sur leur taille :*

Cas de base Tout mot u sur V_t est une expression rationnelle ;

Cas général Si e et e' désignent des expressions rationnelles, alors

1. *$e + e'$ est une expression rationnelle appellée* somme *de e et e' ;*
2. *ee' est une expression rationnelle appellée* produit *de e et e' ;*
3. *e^* est une expression rationnelle appellée* itérée *de e ;*
4. *(e) est une expression rationnelle.*

Toute expression rationnelle dénote un langage dit *rationnel*, que l'on définit par récurrence sur la taille des expressions rationnelles :

Définition 6.2 *On définit le langage $\mathcal{L}ang(\alpha)$ de l'expression rationnelle α par récurrence sur la taille de α :*

Cas de base $\mathcal{L}ang(u) = \{u\}$;

Cas général Si e et e' désignent des expressions rationnelles, alors

1. *$\mathcal{L}ang(e + e') = \mathcal{L}ang(e) \cup \mathcal{L}ang(e')$;*
2. *$\mathcal{L}ang(ee') = \mathcal{L}ang(e) \times \mathcal{L}ang(e')$;*
3. *$\mathcal{L}ang(e^*) = (\mathcal{L}ang(e))^*$;*
4. *$\mathcal{L}ang((e)) = \mathcal{L}ang(e)$.*

On considère parfois, sans nécessité, des expressions rationnelles dites étendues, en ajoutant la construction $\overline{e}$ qui dénote le langage complémentaire de $\mathcal{L}ang(e)$. Cette construction est en effet redondante, comme on pourra le déduire du théorème 6.3.

6.2 Théorème de Kleene

Un fait tout à fait remarquable est que les langages rationnels ne sont rien d'autres que les langages reconnaissables : c'est le théorème de Kleene.

Théorème 6.3 *Rat = Rec*

Preuve: $Rat \subseteq Rec$. La preuve se fait aisément par récurrence sur la taille des expressions rationnelles, grâce aux propriétés de clôture des langages reconnaissables.

Cas de base $u \in V_t^*$: il est aisé de construire un automate $\mathcal{A}_u$ reconnaissant u ;

Cas général Si e et e' désignent des expressions rationnelles, alors, par hypothèse de récurrence, il existe des automates $\mathcal{A}_e$ et $\mathcal{A}_{e'}$ reconnaissant les langages $\mathcal{L}ang(e)$ et $\mathcal{L}ang(e')$.

1. L'existence d'un automate reconnaissant $\mathcal{L}ang(e+e')$ découle de la clôture des langages reconnaissables par union ;
2. L'existence d'un automate reconnaissant $\mathcal{L}ang(ee')$ découle de la clôture des langages reconnaissables par produit ;
3. L'existence d'un automate reconnaissant $\mathcal{L}ang(e^*)$ découle de la clôture des langages reconnaissables par étoile ;
4. L'automate reconnaissant $\mathcal{L}ang((e))$ est tout simplement l'automate reconnaissant e.

$Rec \subseteq Rat$. Cette direction est rendue difficile par l'absence de définition par récurrence pour les automates. On va restaurer une récurrence en exprimant adroitement les chemins de calcul des mots reconnus par un automate, qu'il soit ou non déterministe. Soit donc $\mathcal{A} = (V_t, Q, i, F, T)$ un automate.

On va noter par $L^I_{q,q'}$ avec $q \in I$ et $q' \in I$, le langage des mots reconnus par l'automate $\mathcal{A}$ depuis l'état q jusqu'à l'état q' sans jamais passer par un état de I le long du chemin.

Le principe de la construction va être de découper les chemins de q à q' en une succession de chemins de q à q ne repassant pas par q, puis en un chemin de q à q' ne repassant pas par q ni par q', pour terminer par une succession de chemins de q' à q' ne repassant ni par q ni par q'. Cela revient à faire apparaître le long du chemin qui va de q à q' tous les passages par q, puis tous les passages par q' le long du chemin restant allant de q à q' sans jamais passer par q. On en déduit donc les équations aux langages suivantes :

$$
\begin{aligned}
\mathcal{L}ang(\mathcal{A}) &= \bigcup_{\substack{f \in F \\ f \neq i}} (L^{\{i\}}_{i,i})^* \times L^{\{i,f\}}_{i,f} \times (L^{\{i,f\}}_{f,f})^* \\
L^I_{q,q} &= \{\varepsilon\} \cup \bigcup_{\substack{a \in V_t \text{ s.t.} \\ q \in T(q,a)}} \{a\} \cup \bigcup_{\substack{a \in V_t \text{ and} \\ q'' \in T(q,a) \setminus I}} \{a\} \times (L^{I \cup \{q''\}}_{q'',q''})^* \times L^{I \cup \{q''\}}_{q'',q} \\
L^I_{q,q' \neq q} &= \bigcup_{\substack{a \in V_t \text{ s.t.} \\ q' \in T(q,a)}} \{a\} \cup \bigcup_{\substack{a \in V_t \text{ and} \\ q'' \in T(q,a) \setminus I}} \{a\} \times (L^{I \cup \{q''\}}_{q'',q''})^* \times L^{I \cup \{q''\}}_{q'',q'}
\end{aligned}
$$

On peut maintenant montrer par récurrence sur la taille de l'ensemble d'états autorisés $Q \setminus I$ que le langage $L^I_{q,q'}$ est rationnel.

Cas de base : si $Q = I$, alors on a nécessairement $L^Q_{q,q} = \{\varepsilon\} \cup \bigcup_{\substack{a \in V_t \text{ s.t.} \\ q \in T(q,a)}} \{a\}$ et $L^Q_{q,q' \neq q} = \bigcup_{\substack{a \in V_t \text{ s.t.} \\ q \in T(q,a)}} \{a\}$ qui sont l'un et l'autre rationnels.

Cas général : on applique l'hypothèse de récurrence aux langages $L^{I \cup \{q''\}}_{q'',q''}$ et $L^{I \cup \{q''\}}_{q'',q}$ dans le premier cas, et aux langages $L^{I \cup \{q''\}}_{q'',q''}$ et $L^{I \cup \{q''\}}_{q'',q'}$ dans le second avant de conclure par utilisation des opérations d'addition, produit et itération des expressions rationnelles dont l'interprétation est justement l'union, le produit et l'itéré de langages.

□

Donnons un exemple illustrant le calcul d'une expresion rationnelle pour le langage reconnu par l'automate de la figure 6.1.

FIG. 6.1 – Langage reconnu par un automate.

Langage		Calcul		Expression rationnelle
L	$=$	$(L_{0,0}^{\{0\}})^*$		
$L_{0,0}^{\{0\}}$	$=$	$\{\varepsilon\} \cup \{a\} \times (L_{1,1}^{\{0,1\}})^* \times L_{1,0}^{\{0,1\}}$	$=$	$\varepsilon + a(b + ce^*d)$
$L_{1,0}^{\{0,1\}}$	$=$	$\{b\} \cup \{c\} \times (L_{2,2}^{\{0,1,2\}})^* \times L_{2,0}^{\{0,1,2\}}$	$=$	$b + c(\varepsilon + e)^*$
$L_{1,1}^{\{0,1\}}$	$=$	$\{\varepsilon\} \cup \{c\} \times ()^* \times L_{2,1}^{\{0,1,2\}}$	$=$	ε
$L_{2,1}^{\{0,1,2\}}$	$=$	$\emptyset$	$=$	
$L_{2,2}^{\{0,1,2\}}$	$=$	$\{\varepsilon\} \cup \{e\}$	$=$	$\varepsilon + e$
$L_{2,0}^{\{0,1,2\}}$	$=$	$\{d\}$	$=$	d

Notons que le langage vide $L_{2,1}^{\{0,1,2\}}$ est dénoté par l'expression rationnelle vide.

6.3 Identités remarquables

Deux expressions rationnelles distinctes peuvent dénoter le même langage. Par exemple, $(a+b)^*$ et $(a^*b^*)^*$ dénotent toutes deux le langage des mots quelconques sur l'alphabet $\{a, b\}$. Nous donnons ci-dessous un ensemble d'identités remarquables satisfaites par les expressions rationnelles. Elles pourront être vérifiées en exercice.

1. $r + s = s + r$
2. $(r + s) + t = r + (s + t)$
3. $(rs)t = r(st)$
4. $r(s + t) = rs + rt$
5. $(r + s)t = rt + st$
6. $\emptyset^* = \varepsilon$
7. $(r^*)^* = r^*$
8. $(\varepsilon + r)^* = r^*$
9. $(r^*s^*)^* = (r + s)^*$

6.4 Analyse lexicale avec l'outil LEX

L'analyse lexicale aura pour but de préparer l'étape suivante, l'analyse syntaxique, en engendrant une séquence de *tokens* qui sera ultérieurement analysée par l'analyseur syntaxique. Un analyseur lexical est donc une sorte de gros automate dont les états acceptants reconnaissent les différents tokens (c'est-à-dire les différentes sortes d'unités lexicales) du langage.

Les différents tokens une fois spécifiés sous forme d'expressions rationnelles, un automate déterministe minimal peut être construit pour chacun d'eux. L'outil LEX est un outil disponible dans différents langages de programmation, qui automatise cette transformation.

6.4.1 Notion de token

Lors de l'analyse syntaxique d'un langage, nous avons vu qu'il était pratique de définir la grammaire du langage non pas directement sur le vocabulaire du langage, mais plutôt en supposant déjà définies des grammaires simples (en l'occurrence des grammaires régulières) pour les composants simples du langage appelés *unités lexicales* : les constantes numériques, les identificateurs, les mots-clés, etc. Chaque non-terminal source de l'une de ces grammaires régulières est appelé un *token*. Un token représente donc le langage régulier associé à une unité lexicale donnée.

6.4.2 Fichier de description LEX

En pratique, il est agréable d'utiliser des expressions rationnelles auxiliaires, et la définition de langages rationnels se présente donc sous la forme suivante dans le langage LEX :

1. Une suite de définitions de la forme $D_i = R_i$, où D_i est un nom, et R_i est une expression régulière sur l'alphabet $V \cup \{D_1, \ldots, D_{i-1}\}$.
2. Une suite de règles de la forme **Si** P_i **Alors** A_i, où P_i est une expression rationnelle sur l'alphabet $V \cup \{D_i\}_i$ appelée *filtre*, et A_i est un programme caractéristique du type d'unité lexicale reconnu par le filtre P_i.

Par exemple, pour le langage des expressions arithmétiques, on aura :

lettre = $a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$
chiffre = $0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Si chiffre(chiffre)* **Alors** %traitement des constantes
Si lettre(lettre)* **Alors** %traitement des identificateurs
Si $+$ **Alors** %traitement de l'addition
Si $\times$ **Alors** %traitement de la multiplication

Les traitements peuvent être en général n'importe quelle expression ou instruction du langage de programmation auquel l'outil LEX est associé. Nous l'utiliserons uniquement pour retourner des *tokens* (voir paragraphe suivant).

Les outils LEX autorisent en général d'autres constructions pour les expressions rationnelles, comme les intervalles de lettres, qui permettent d'abréger l'écriture. L'outil LEX le plus utilisé est l'original et produit du langage C ; mais il existe également pour d'autres langages, avec quelques variantes. Noter qu'en CAML, l'outil CAMLLEX ne permet pas les définitions.

6.4.3 Description des tokens

LEX va nous servir à reconnaitre les différentes unités lexicales sous forme de définitions auxiliaires et de règles. Nous pourrons par exemple avoir un fichier LEX de la forme :

lettre = $a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$
chiffre = $0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Si if **Alors** If
Si $then$ **Alors** Then
Si $else$ **Alors** Else
Si chiffre(chiffre)* **Alors** *Id*
Si lettre(lettre)* **Alors** Cte

Notons que puisque toutes les constantes sont remplacées par le token Cte, on a perdu de l'information. Pour la suite de l'interprétation du langage considéré, il faut garder un lien entre le token et la valeur de la constante correspondante. Le même problème se pose pour les identificateurs.

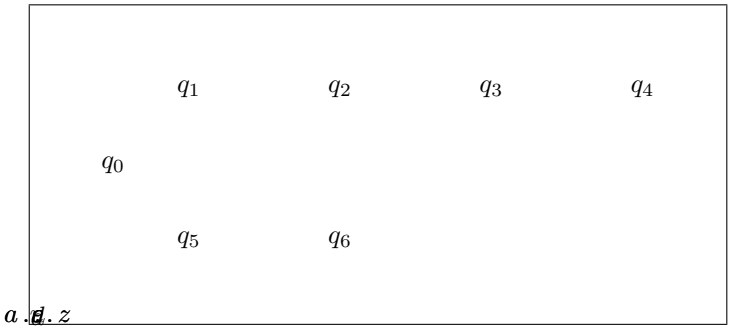


FIG. 6.2 – Automate d'analyse lexicale

Il y a plusieurs façons de conserver ce lien, suivant le type de langage utilisé. En programmation fonctionnelle, on peut considérer le token comme étant une fonction dont l'argument est l'unité lexicale qui l'a produit : c'est ce qui est fait dans CAMLLEX. En programmation impérative, il est d'usage de conserver ce lien dans une table, la *table des symboles*. Le token est alors une paire, formée du token lui-même d'une part, et d'un pointeur sur la table des symboles d'autre part. La table des symboles pourra bien sûr contenir un grand nombre d'informations pour chaque occurrence différente d'un même token.

Décider d'une solution ou d'une autre est une question d'implémentation. En ce qui nous concerne, nous supposerons que l'unité lexicale représentée par le token t est obtenue par un appel à une fonction val, donc est donnée par l'expression val(t).

6.4.4 Règles de priorité

Étant donné un programme à analyser, c'est-à-dire un mot w sur l'alphabet V du langage de programmation considéré, il y aura en général de multiples façons de découper w en unités lexicales. Un principe de sélection est donc ajouté qui garantit l'unicité du découpage :

(i) Le filtre choisi est celui qui reconnaît le plus grand préfixe du mot w.

(ii) Au cas où plusieurs filtres $P_{i_1}, \ldots, P_{i_q}$, avec $i_1 < \ldots < i_q$, reconnaissent le même (plus grand) préfixe u, le filtre de plus petit indice P_{i_1} est choisi.

Soit par exemple un fragment de langage contenant deux types de tokens, le mot clé « end », et les identificateurs. On aura :

lettre = $a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$
Si end **Alors** End
Si lettre(lettre)* **Alors** Id

Les automates correspondants à ces filtres sont représentés sur la figure 6.2 sous la forme d'un automate avec transitions vides dont la partie supérieure traite le mot clé « end » et la partie inférieure traite les identificateurs. À chaque état acceptant peut être associé le token correspondant : End pour q_4 et Id pour q_6.

Soit « endormi » le mot à reconnaître. Les deux filtres reconnaissent le même préfixe, mais le deuxième reconnaît aussi un préfixe plus long, le second filtre est donc choisi et l'état final de l'exécution de l'automate doit donc être l'état q_6. Soit maintenant « end » le mot à reconnaître. Les deux filtres reconnaissent le même préfixe de longueur maximale, « end ». Le premier filtre est donc choisi, et l'état final doit donc être l'état q_4.

Cet automate est déterminisé (et se trouve être minimal) sur la figure 6.3. L'état acceptant q_3 devrait normalement correspondre aux deux états acceptants de la figure 6.2. Le choix du premier

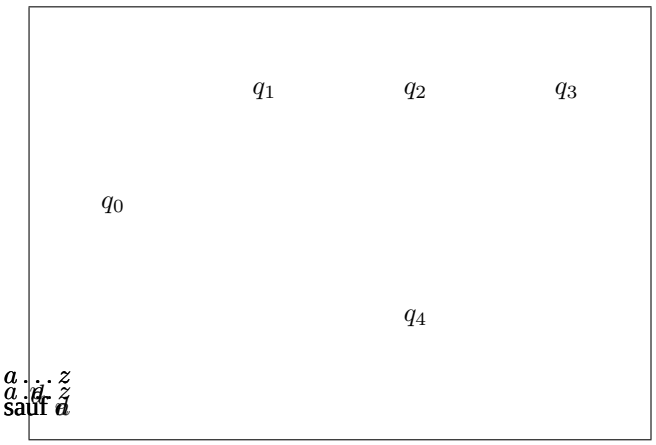


FIG. 6.3 – Automate déterminisé d'analyse lexicale

filtre, c'est-à-dire du mot clé, fait que cet état acceptant est en fait associé au mot clé uniquement. Les tokens correspondants aux états acceptants sont donc End pour q_3 et Id pour q_1, q_2 et q_4.

En pratique, ces règles de priorité imposent que dans LEX, les filtres correspondants aux différents mots clés devront apparaître avant le filtre pour les identificateurs. De plus, on remarque qu'à chaque état acceptant est associé un unique token, décrit par un filtre P_i, et le code à exécuter lorsque cet état acceptant est atteint est donc le code A_i.

6.5 Exercices

Exercice 6.1 *Donner des expressions rationnelles dénotant les langages suivants :*

1. *les entiers en décimal ;*
2. *les entiers relatifs en décimal, c'est-à-dire pouvant optionnellement commencer par un signe $+$ ou un signe $-$;*
3. *les identificateurs de PASCAL ;*
4. *les nombres réels en virgule flottante ;*
5. *les nombres réels en virgule flottante avec exposants.*

Exercice 6.2 *Donner un automate reconnaissant le langage des mots formés d'un nombre pair de a et d'un nombre impair de b. En déduire une expression rationelle pour ce langage.*

Exercice 6.3 *Est-ce que les expressions suivantes sont équivalentes pour n'importe quelles expressions rationnelles r et s :*

1. $(rs + r)^*r$ *et* $r(sr + r)^*$
2. $s(rs + s)^*r$ *et* $rr^*s(rr^*s)^*$
3. $(r + s)^*$ *et* $r^* + s^*$

Exercice 6.4

1. *Donner une expression rationnelle dénotant le langage*

$$L_0 = \{a^n b^p \mid \text{où } n \text{ et } p \text{ sont des entiers naturels pairs(y compris 0)}\}$$

2. *Donner un automate déterministe minimal reconnaissant le langage L_0.*
3. *Donner une expression rationnelle (naïve) dénotant le langage $L = \bigcup_{k\geq 0} L_0^{2k+1}$*

4. *Donner une expression rationnelle d'au plus 13 symboles qui dénote le langage L (il en existe en fait de moins de 10 symboles). Justifiez votre réponse.*
5. *En déduire un automate déterministe minimal reconnaissant L.*
6. *Donner une expression rationnelle dénotant le langage L'_0 des mots obtenus à partir des mots de L_0 en supprimant, pour chacun d'eux, les lettres de rang impair (la première, la troisième, etc.), de telle sorte que le mot $aaaaaabbbbbbbb$ devient $aaabbbb$).*
7. *Donner un automate déterministe minimal pour L'_0.*
8. *Donner une expression rationnelle dénotant le langage L''_0 des mots obtenus à partir des mots de L_0 en supprimant, pour chacun d'eux, leur seconde moitié, de telle sorte que le mot $aaaaaabbbbbbbb$ devient le mot $aaaaaab$.*
9. *Donner un automate déterministe minimal pour L''_0.*
10. *Exprimer à l'aide de L et L''_0 le langage L'' des mots obtenus à partir des mots de L en supprimant, pour chacun d'eux, leur seconde moitié, de telle sorte que le mot $aabbaaaabbaaaabbbbbbbb$ devient $aabbaaaabba$.*
11. *En déduire un automate déterministe minimal pour L''.*

Chapitre 7

Grammaires formelles

Les grammaires sont des outils pour décrire des langages, inspirés des grammaires utilisées pour les phrases du langage naturel, comme par exemple :

> *Une phrase simple est constituée d'un sujet suivi d'un verbe suivi d'une préposition suivie d'un complément. Le sujet est un prénom ou bien un pronom. Les prénoms sont à choisir parmi « Alain », « Béatrice » et « Charles ». Les pronoms sont à choisir parmi « Il » et « Elle ». Les prépositions sont à choisir parmi « à », « de », « au » et « en ». Les verbes sont à choisir parmi « va » et « vient ». Les compléments sont à choisir parmi « l'école » et « marché ».*

Les règles informelles ci-dessus peuvent être décrites par des règles de grammaire.

7.1 Grammaires

Définition 7.1 *Une* grammaire *G est un quadruplet (V_t, V_n, S, R) où*

1. *V_t est un ensemble fini de symboles dits* terminaux*, appelé* vocabulaire terminal *;*
2. *V_n est un ensemble fini (disjoint de V_t) de symboles dits* non-terminaux*, appelé* vocabulaire non terminal *;*
3. *$V = V_n \cup V_t$ est appelé le* vocabulaire *de la grammaire G ;*
4. *S est un non-terminal particulier appelé* source*, ou* axiome *;*
5. *$R \subseteq (V^* \setminus V_t^*) \times V^*$ est un ensemble fini de* règles de grammaire *notées $g \to d$ si $(g, d) \in R$.*

On utilisera des mots en italique commençant par une majuscule pour les non-terminaux, et des lettres minuscules ou des symboles spéciaux ($+$, $\times$, etc.) pour les terminaux.

Exercice 7.1 *Reformuler la description introductive sous forme de règles de grammaire.*

Un autre exemple est la description de l'ensemble des entiers naturels en représentation binaire, vus comme des suites non vides de 0 et de 1 ayant la propriété que seule la suite 0 peut commencer par un 0. Pour cela, nous allons utiliser une grammaire pour laquelle l'ensemble des symboles non-terminaux est $\{ N , M \}$, la source est le non-terminal N, l'ensemble des symboles terminaux est $\{0, 1\}$ et les règles sont décrites figure 7.1. Ce tableau présente en fait quatre ensembles de règles, dont le premier et le troisième sont conformes à la définition, alors que les règles de même membre gauche ont été regroupées dans le second et le quatrième en séparant les différents membres droits par le symbole « | » signifiant « ou », conformément à une convention usuelle : $N \to u_1 \mid \cdots \mid u_n$ abrège l'ensemble des règles $N \to u_1$, …, $N \to u_n$. Les quatre ensembles de règles de la figure 7.1 décrivent le même langage, celui des nombres entiers naturels écrits en binaire, mais le troisième est

plus compact que le premier grâce à l'utilisation du symbole ε. Le premier et le second ensemble de règles doivent être considérés comme identiques, de même que le troisième et le quatrième, car ils ne diffèrent que par l'économie d'écriture juste mentionnée.

$$\begin{array}{lcl} N & \to & 0 \\ N & \to & 1 \\ N & \to & 1M \\ M & \to & 0 \\ M & \to & 1 \\ M & \to & 0M \\ M & \to & 1M \end{array} \qquad \begin{array}{lcl} N & \to & 0 \mid 1 \mid 1M \\ M & \to & 0 \mid 1 \mid 0M \mid 1M \end{array} \qquad \begin{array}{lcl} N & \to & 0 \\ N & \to & 1M \\ M & \to & \varepsilon \\ M & \to & 0M \\ M & \to & 1M \end{array} \qquad \begin{array}{lcl} N & \to & 0 \mid 1M \\ M & \to & \varepsilon \mid 0M \mid 1M \end{array}$$

FIG. 7.1 – Deux grammaires décrivant les entiers écrits en binaire

7.2 Langage engendré par une grammaire

Une grammaire est un mécanisme permettant *d'engendrer* les phrases du langage, en *récrivant* un mot $u \in V^*$ en un nouveau mot $v \in V^*$ par l'opération consistant à remplacer une occurrence (quelconque) d'un non-terminal N (quelconque) de V_n présent dans u par un membre droit de règle dont N est membre gauche. Ce processus est itéré à partir de la source jusqu'à l'élimination complète des non-terminaux. Par exemple, en utilisant la première grammaire définie dans la figure 7.1 :

$$N \to 1M \to 11M \to 110M \to 1101M \to 11010$$

et en utilisant la deuxième grammaire définie dans la figure 7.1 :

$$N \to 1M \to 11M \to 110M \to 1101M \to 11010M \to 11010$$

Définition 7.2 *Étant donnée une grammaire $G = \{V_t, V_n, S, R\}$, on dit que le mot $u \in V^*$ se* récrit *en le mot $v \in V^*$ dans G avec la règle $g \to d$, et l'on note $u \xrightarrow{g \to d} v$, si*

1. *$u = w_1 g w_2$, où $w_1 \in V^*$ et $w_2 \in V^*$;*
2. *$v = w_1 d w_2$;*

On dit que le mot $u \in V^*$ se récrit *en le mot $v \in V^*$ dans G s'il existe une règle $g \to d \in R$ telle que u se récrit en v dans G. Enfin, on dit que le mot $v \in V^*$* dérive *du mot $u \in V^*$, dans la grammaire G, ce que l'on note par $u \to^* v$, s'il existe une suite finie $w_0, w_1, \ldots, w_n$ de mots de V^* telle que $w_0 = u$, $w_i \to w_{i+1}$ pour tout $i \in [0, n-1]$, et $w_n = v$.*

Nous avons vu ci-dessus que le mot $11010M$ dérive du mot $1M$ dans la troisième grammaire de la figure 7.1. Par convention, nous indiquerons le non-terminal récrit dans un mot en le soulignant, comme dans

$$1\underline{M} \to 11\underline{M} \to 110\underline{M} \to 1101\underline{M} \to 11010M$$

Définition 7.3 *Le* langage engendré *par la grammaire G est l'ensemble des mots de V_t^* qui dérivent de l'axiome de G, que l'on note par $\mathcal{L}ang(G)$.*

Il est aisé de se convaincre que l'ensemble des mots qui appartiennent aux langages engendrés par les grammaires de la figure 7.1 est dans les deux cas l'ensemble des entiers naturels écrits en binaire, c'est-à-dire les suites de 0 et de 1 ne commençant pas par 0, sauf la suite 0 elle-même.

7.3 Classification de Chomsky

Les grammaires telles que nous venons de les définir sont dites *générales*. Les langages engendrés sont dits de *type 0* dans la *hiérachie de Chomsky*. On verra qu'ils sont reconnus par des *machines de Türing*.

7.3.1 Grammaires contextuelles

La grammaire de la figure 7.2 décrit le langage $\{a^n b^n c^n \mid n \geq 0\}$. Ses règles ont la particularité d'avoir un membre droit de taille supérieure ou égale à celle de leur membre gauche.

$$\begin{array}{rcl} S & \rightarrow & aSBC|\varepsilon \\ CB & \rightarrow & BC \\ aB & \rightarrow & ab \\ bB & \rightarrow & ab \\ bC & \rightarrow & bc \\ cC & \rightarrow & cc \end{array}$$

FIG. 7.2 – Grammaire contextuelle

Définition 7.4 *Étant donnée une grammaire $G = \{V_t, V_n, S, R\}$, une règle de R est dite* contextuelle *si elle est de la forme $g \rightarrow d$ avec $|g| \geq |d|$. Une grammaire est* contextuelle*si toutes ses règles le sont, et le langage engendré est alors dit* contextuel *ou de* type 1.

Les langages de type sont reconnus par des machines de Türing particulières utilisant un espace mémoire borné.

7.3.2 Grammaires hors-contexte

La grammaire

$$S \rightarrow \varepsilon|aSb$$

décrit le langage $\{a^n b^n \mid n \geq 0\}$. Cette grammaire a une particularité : le membre gauche de chaque règle est un non-terminal.

Définition 7.5 *Étant donnée une grammaire $G = \{V_t, V_n, S, R\}$, une règle de R est dite* hors-contexte *si elle est de la forme $N \rightarrow u$ où $u \in V^*$. Une grammaire est* hors-contexte *si toutes ses règles le sont, et le langage engendré est alors dit* hors-contexte *ou de* type 2. *On note par $\mathcal{L}_{\mathcal{HC}}$ la classe des langage hors-contexte.*

Les langages hors-contexte sont reconnus par des machines de Türing particulières, les automates à pile, que nous découvrirons dans le prochain chapitre.

7.3.3 Grammaires régulières

Les grammaires de l'exemple de la figure 7.1 sont hors-contexte, mais elles ont une particularité supplémentaire : le membre droit possède au plus une occurrence de symbol non terminal, située à l'extrémité droite.

Définition 7.6 *Étant donnée une grammaire $G = \{V_t, V_n, S, R\}$, une règle de R est dite* régulière *si elle est de la forme $N \rightarrow u$ ou $N \rightarrow uM$, où $M, N \in V_n$ et $u \in V_t^*$. Une grammaire est* régulière *si toutes ses règles le sont, et un langage est alors dit* régulier *ou de* type 3.

Les langages réguliers sont bien sûr reconnus par des machines de Türing particulières :

Théorème 7.7 *Un langage est régulier si et seulement si il est reconnaissable.*

Preuve: Étant donné une grammaire régulière $G = \{V_t, V_n, S, R\}$, on commence par calculer une nouvelle grammaire $G' = \{V_t, V'_n, S, R'\}$ engendrant le même langage dont les règles sont de la forme $N \to a,\ N \to \varepsilon,\ N \to aM$ ou $N \to M$, avec $N, M \in V_n$ et $a \in V_t$. Ceci se fait en remplaçant récursivement toute règle de la forme $N \to a_1 u$ où $|u| > 1$ par les deux règles $N \to aK,\ K \to u$ où K désigne un nouveau non terminal ajouté à la grammaire. On construit ensuite l'automate $\mathcal{A} = (V_t, V'_n \cup \{f\}, S, \{f\}, T)$ tel que $M \in T(N, a)$ ssi $N \to aM \in R'$, $M \in T(N, \varepsilon)$ ssi $N \to M \in R'$, $f \in T(N, a)$ ssi $N \to a \in R'$, et $f \in T(N, \varepsilon)$ ssi $N \to \varepsilon \in R'$. On vérifie alors aisément que $\mathcal{L}ang(A) = \mathcal{L}ang(G')$.

Réciproquement, à partir d'un automate $\mathcal{A} = (V_t, Q, i, F, T)$ sans transition vide, on construit la grammaire régulière $G = \{V_t, Q, i, R\}$ telle que $q \to aq'$ ssi $q' \in T(q, a)$ et $f \to \varepsilon$ ssi $f \in F$, qui engendre le langage $\mathcal{L}ang(A)$. □

Les règles régulières d'une grammaires décrivent donc les parties reconnaissables d'un langage. La syntaxe d'un langage de programmation comporte généralement un nombre fini de grammaires régulières dont les sources sont appellées des *tokens* : on pourra ainsi avoir une grammaire pour les constantes entières, une autre pour les constantes réelles en simple précision, une pour les identificateurs, etc. Ces tokens peuvent être considérés comme les terminaux de la partie non régulière de la grammaire.

7.4 Forme normale de Chomsky

Nous venons de voir le besoin de *simplifier* une grammaire dans certains cas. Nous allons le faire dans un cadre général : celui des grammaires hors-contexte.

Définition 7.8 *Une grammaire hors-contexte $G = \{V_t, V_n, S, R\}$ est dite* propre *si elle vérifie :*

1. $\forall N \to u \in R$, $u \neq \varepsilon$ *ou* $N = S$
2. $\forall N \to u \in R$, $S \notin u$
3. *Les non-terminaux sont tous* utiles, *c'est-à-dire à la fois*
   (a) $\forall N \in V_n$, N *est* atteignable *depuis* S *:* $\exists \alpha, \beta \in V^*$ *tels que* $S \xrightarrow{*} \alpha N \beta$ *;*
   (b) $\forall N \in V_n$, N *est* productif *:* $\exists w \in V_t^*$ *tel que* $N \xrightarrow{*} w$.

Théorème 7.9 *Pour toute grammaire hors-contexte $G = \{V_t, V_n, S, R\}$, il existe une grammaire hors-contexte $G' = \{V_t, V'_n \subseteq V_n \cup \{S'\}, S', R'\}$ propre qui engendre le même langage.*

Preuve: On procède en plusieurs étapes dans cet ordre :

1. On rajoute une règle $S' \to S$, S' devenant la nouvelle source ;
2. Élimination des $M \to \varepsilon$.
   Calculer l'ensemble $E = \{M \in V'_n \mid M \xrightarrow{*} \varepsilon\}$;
   Pour tout $M \in E$ Faire
   Pour toute règle $N \to \alpha M \beta$ de R Faire
   Ajouter la règle $N \to \alpha\beta$ à R
   Fin Faire ;
   Fin Faire
   Enlever les règles $M \to \varepsilon$ telles que $M \in V_n$

3. Élimination des règles $M \to N$. On applique la procédure suivante à G privée de la règle $S' \to \varepsilon$:
   Calculer les paires (M, N) telles que $M \xrightarrow{*} N$;
   Pour chaque paire (M, N) calculée Faire
   Pour chaque règle $N \to u_1 | \ldots | u_n$ de R Faire
   Ajouter la règle $M \to u_1 | \ldots | u_n$ à R
   Fin Faire
   Fin Faire ;
   Enlever toutes les règles $M \to N$
4. Suppression des non terminaux non productifs :
   Calculer les non terminaux productifs ;
   Enlever tous les autres
5. Suppression des non terminaux non atteignables :
   Calculer les non terminaux atteignables ;
   Enlever tous les autres

On remarque que chaque étape ne remet pas en cause la précédente, et donc la grammaire obtenue est propre. Ce ne serait pas le cas si l'on inversait les deux dernières étapes, comme le montre l'exemple

$$G = \{\{a\}, \{S, M, N\}, S, R\} \quad R = \{S \to aMN, M \to a\}$$

□

La mise sous forme propre d'une grammaire hors-contexte est la succession de 4 étapes qui terminent. On en déduit que le vide d'une grammaire est décidable :

Corollaire 7.10 *Le langage engendré par une grammaire G est vide si et seulement si sa mise sous forme propre résulte en une grammaire vide.*

Théorème 7.11 *Pour tout langage hors-contexte L, il existe une grammaire propre G qui l'engendre dont toutes les règles sont de l'une des trois formes $S \to \varepsilon$, $P \to a$, ou $P \to MN$ avec M, N différents de S.*

Preuve: Partant d'une grammaire hors-contexte propre $G = \{V_t, V_n, S, R\}$, on commence par rajouter une copie de V_t à V_n (on notera A la copie de a), puis l'ensemble de règles $\{A \to a \mid a \in V_t\}$. On remplace ensuite les symboles terminaux a figurant dans les membres droits de règles initiales par le non-terminal correspondant A, puis on remplace récursivement la règle $X \to X_1 \ldots X_n$ pour $n > 2$ par $X \to X_1 Y$ et $Y \to X_2 \ldots X_n$ en ajoutant un nouveau non-terminal Y à chaque fois. □

7.5 Dérivations gauches et droites

Nous allons maintenant décrire un langage présent dans tous les langages de programmation, le langage des expressions arithmétiques. Nous noterons par *Id* et *Cte* les tokens décrivant respectivement les noms de variables et les constantes entèères ou réelles (sans les exposants pour simplifier). Ces syntaxes seront supposées être celles des langages informatiques classiques, à l'exception près que les constantes et les variables pourront être de taille arbitraire. Pour simplifier nous considérons également que les seules opérations sont l'addition et la multiplication.

Une grammaire des expressions arithmétiques aura alors pour symboles terminaux $V_t = \{+, \times, (,), \mathit{Id}, \mathit{Cte}\}$ et pour non-terminaux l'unique symbole E qui sera donc la source :

$$E \to \mathit{Id} \mid \mathit{Cte} \mid E + E \mid E \times E \mid (E)$$

Cette grammaire permet d'engendrer l'expression arithmétique $x + 2.5 \times 4 + (y + z)$, en supposant que les mots 2.5 et 4 font partie du langage engendré par la grammaire des constantes, et que x, y, z font partie du langage engendré par la grammaire des variables. En effet :

$$\underline{E} \to \underline{E} + E \to \mathit{Id} + \underline{E} \to \mathit{Id} + \underline{E} \times E \to \mathit{Id} + \mathit{Cte} \times \underline{E} \to \mathit{Id} + \mathit{Cte} \times \underline{E} + E \to$$
$$\mathit{Id} + \mathit{Cte} \times \mathit{Cte} + \underline{E} \to \mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (\underline{E}) \to \mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (\underline{E} + E) \to$$
$$\mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (\mathit{Id} + \underline{E}) \to \mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (\mathit{Id} + \mathit{Id})$$

et il suffit de remplacer de manière appropriée les différentes occurrences de *Id* et *Cte* pour obtenir le résultat désiré. La dérivation ci-dessus est appelée *dérivation gauche*, car le non-terminal réécrit à chaque étape est celui qui est situé le plus à gauche dans le mot obtenu à l'étape précédente. La dérivation ci-dessus n'est pas unique, en voici une autre très semblable, qui n'est pas une dérivation gauche, mais est obtenue en permutant les deux dernières étapes de la dérivation gauche :

$$\underline{E} \to \underline{E} + E \to \mathit{Id} + \underline{E} \to \mathit{Id} + \underline{E} \times E \to \mathit{Id} + \mathit{Cte} \times \underline{E} \to \mathit{Id} + \mathit{Cte} \times \underline{E} + E \to$$
$$\mathit{Id} + \mathit{Cte} \times \mathit{Cte} + \underline{E} \to \mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (\underline{E}) \to \mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (E + \underline{E}) \to$$
$$\mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (\underline{E} + \mathit{Id}) \to \mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (\mathit{Id} + \mathit{Id})$$

En voici encore une autre, une *dérivation droite* cette fois :

$$\underline{E} \to E + \underline{E} \to E + (\underline{E}) \to E + (E + \underline{E}) \to E + (\underline{E} + \mathit{Id}) \to \underline{E} + (\mathit{Id} + \mathit{Id}) \to$$
$$E \times \underline{E} + (\mathit{Id} + \mathit{Id}) \to \underline{E} \times \mathit{Cte} + (\mathit{Id} + \mathit{Id}) \to E + \underline{E} \times \mathit{Cte} + (\mathit{Id} + \mathit{Id}) \to$$
$$\underline{E} + \mathit{Cte} \times \mathit{Cte} + (\mathit{Id} + \mathit{Id}) \to \mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (\mathit{Id} + \mathit{Id})$$

Comment être sûr que ces dérivations sont équivalentes, en le sens que les mots engendrés ont la même signification, c'est-à-dire donneront lieu au même calcul ? Il se trouve que le calcul qui est associé à un mot w dépend non seulement du mot lui-même, mais aussi de la dérivation qui a servi à le fabriquer ! Cette dépendance s'exprime par un parenthésage du mot, qui peut être implicite dans le mot w, mais devient explicite dans la dérivation. Par exemple, le parenthésage explicite du mot $x + 2.5 \times 4 + (y + z)$ tel qu'il est défini par la première dérivation est $(x + (2.5 \times (4 + (y + z))))$. La seconde dérivation donne le même parenthésage, mais il n'en est pas de même pour la troisième, qui correspond au parenthésage $(((x + 2.5) \times 4) + (y + z))$. En fait le parenthésage attendu n'est ni l'un ni l'autre. Il doit traduire les conventions de *désambiguïsation* habituelles, à savoir que la multiplication a priorité sur l'addition, et que multiplication et addition associent à gauche, c'est-à-dire que $x + y + z$ doit être interprété comme étant $((x + y) + z)$. Cela ne semble pas très important puisque l'addition et la multiplication sont associatives, mais cela le devient lorsque la soustraction et la division font partie du langage. Le bon parenthésage est donc $((x + (2.5 \times 4)) + (y + z))$, et une dérivation de notre exemple conforme à ce parenthésage est la suivante :

$$\underline{E} \to E + \underline{E} \to E + (\underline{E}) \to E + (E + \underline{E}) \to E + (\underline{E} + \mathit{Id}) \to \underline{E} + (\mathit{Id} + \mathit{Id}) \to$$
$$E + \underline{E} + (\mathit{Id} + \mathit{Id}) \to E + E \times \underline{E} + (\mathit{Id} + \mathit{Id}) \to E + \underline{E} \times \mathit{Cte} + (\mathit{Id} + \mathit{Id}) \to$$
$$\underline{E} + \mathit{Cte} \times \mathit{Cte} + (\mathit{Id} + \mathit{Id}) \to \mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (\mathit{Id} + \mathit{Id})$$

Cette dérivation est elle aussi une dérivation droite. Nous pouvons maintenant en donner une variante (à des permutations près) dans laquelle l'utilisation de la règle $E \to E \times E$ a lieu lorsqu'aucune autre règle ne peut être utilisée, ce qui est une manière de traduire la convention de désambiguïsation :

$$\underline{E} \to E + \underline{E} \to E + (\underline{E}) \to E + (E + \underline{E}) \to E + (\underline{E} + \mathit{Id}) \to \underline{E} + (\mathit{Id} + \mathit{Id}) \to$$
$$\underline{E} + E + (\mathit{Id} + \mathit{Id}) \to \mathit{Id} + \underline{E} + (\mathit{Id} + \mathit{Id}) \to \mathit{Id} + E \times \underline{E} + (\mathit{Id} + \mathit{Id}) \to$$
$$\mathit{Id} + \underline{E} \times \mathit{Cte} + (\mathit{Id} + \mathit{Id}) \to \mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (\mathit{Id} + \mathit{Id})$$

Notre mot possède donc plusieurs dérivations droites (et aussi plusieurs dérivations gauches) dans la grammaire, ce qui est la manifestation visible de l'ambiguïté. Nous posons donc la définition suivante.

Définition 7.12 *Une grammaire G est* ambiguë *s'il existe un mot $w \in \mathcal{L}ang(G)$ qui possède plusieurs dérivations droites dans G.*

Notre grammaire des expressions arithmétiques est donc ambiguë. Nous allons la transformer de manière à refléter les règles de désambiguïsation, ce qui nécessite l'utilisation de nouveaux non-terminaux F et G. L'idée de la transformation est la suivante : un nouveau symbole non-terminal est utilisé pour chaque niveau de priorité, et une règle récursive gauche est utilisée pour traduire l'associativité à gauche. Par exemple, la règle $E \to E + F$ va permettre d'engendrer une suite d'additions parenthésées à gauche. Le non-terminal F servira à engendrer les expressions arithmétiques qui ne sont pas des additions. Il faudra donc rajouter aussi la règle $E \to F$ de manière à terminer la séquence d'additions. Ainsi, si l'on dispose également de la règle $F \to a$, on engendrera l'expression $a + a + a$ par la dérivation suivante :

$$\underline{E} \to \underline{E} + F \to \underline{E} + F + F \to \underline{F} + F + F \to a + \underline{F} + F \to a + a + \underline{F} \to a + a + a$$

la règle de multiplication doit être changée conformément à ce principe, et les règles de notre nouvelle grammaire des expressions arithmétiques sont alors les suivantes :

$$E \to E + F \mid F \qquad F \to F \times G \mid G \qquad G \to (E) \mid \mathit{Cte} \mid \mathit{Id}$$

Le même mot est maintenant obtenu par une unique dérivation gauche ou droite, nous donnons ci-dessous la dérivation gauche :

$$\begin{aligned}
&\underline{E} \to \underline{E} + F \to \underline{E} + F + F \to \underline{F} + F + F \to \underline{G} + F + F \to \mathit{Id} + \underline{F} + F \to \\
&\mathit{Id} + \underline{F} \times G + F \to \mathit{Id} + \underline{G} \times G + F \to \mathit{Id} + \mathit{Cte} \times \underline{G} + F \to \mathit{Id} + \mathit{Cte} \times \mathit{Cte} + \underline{F} \to \\
&\mathit{Id} + \mathit{Cte} \times \mathit{Cte} + \underline{G} \to \mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (\underline{E}) \to \mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (\underline{E} + F) \to \\
&\mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (\underline{F} + F) \to \mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (\underline{G} + F) \to \mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (\mathit{Id} + \underline{F}) \to \\
&\mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (\mathit{Id} + \underline{G}) \to \mathit{Id} + \mathit{Cte} \times \mathit{Cte} + (\mathit{Id} + \mathit{Id})
\end{aligned}$$

La transformation opérée sur la grammaire a éliminé l'ambiguïté au profit d'une part d'une perte de simplicité, car l'écriture de la grammaire est plus complexe, et d'autre part d'une augmentation de la longueur de la dérivation des mots du langage.

7.6 Arbres syntaxiques

Nous avons vu que, dans une même grammaire G, un mot pouvait avoir plusieurs dérivations équivalentes, c'est-à-dire s'obtenant les unes des autres par des permutations adéquates. Nous allons donc maintenant exhiber une structure de donnée, les *arbres syntaxiques*, qui est invariante par ces permutations, de telle sorte qu'un mot de $\mathcal{L}ang(G)$ possède un arbre syntaxique unique lorsque G est non-ambiguë.

Définition 7.13 *Étant donnée une grammaire $G = (V_t, V_n, S, R)$, les arbres de syntaxe de G sont des arbres, les nœuds internes étiquetés par des symboles de V_n, et les feuilles étiquetés par des symboles de V, tels que, si les fils pris de gauche à droite d'un nœud interne étiqueté par le non-terminal N sont étiquetés par les symboles respectifs $\alpha_1, \ldots, \alpha_n$, alors $N \to \alpha_1 \cdots \alpha_n$ est une règle de la grammaire G.*

Un arbre de syntaxe dont la racine est étiquetée par l'axiome de la grammaire et dont le mot des feuilles u appartient à V_t^* est appelé arbre de dérivation *ou encore* arbre syntaxique *de u.*

Tout arbre de dérivation est donc un arbre de syntaxe, mais certains arbres de syntaxe ne sont pas des arbres de dérivation, soit parce que leur racine n'est pas étiquetée par l'axiome de la grammaire, soit parce que certaines de leur feuilles sont étiquetées par des non-terminaux.

FIG. 7.3 – Deux arbres de dérivation du mot *Id* + *Cte* × *Cte* + (*Id* + *Id*) dans la grammaire ambiguë des expressions arithmétiques

FIG. 7.4 – Parcours gauche et droit de l'arbre de dérivation du mot *Id* + *Cte* × *Cte* + (*Id* + *Id*) dans la grammaire des expressions arithmétiques désambiguïsée

L'arbre de dérivation définit tout un ensemble de dérivations de son mot des feuilles. Une dérivation est en fait obtenue dès que l'on a fixé un mode de parcours de l'arbre dans lequel un nœud ne peut être visité que si son père l'a déjà été. Ainsi, la figure 7.3 montre deux arbres de dérivation du mot *Id* + *Cte* × *Cte* + (*Id* + *Id*) dans la grammaire ambiguë des expressions arithmétiques. Partant de la racine, le parcours gauche consiste à descendre tant que cela est possible en choisissant en permanence la branche la plus à gauche. Lorsque l'on arrive à une feuille, il faut remonter au plus proche ancêtre dont un fils n'a pas encore été visité, puis recommencer la descente à partir du plus à gauche des fils non encore visités. La figure 7.4 indique les parcours gauches et droits de l'arbre de dérivation du mot *Id* + *Cte* × *Cte* + (*Id* + *Id*) dans la grammaire désambiguïsée, en numérotant les nœuds internes de l'arbre de dérivation dans l'ordre où ils sont visités.

La discussion précédente montre que le mot des feuilles d'un arbre de dérivation de la grammaire G appartient à $\mathcal{L}ang(G)$. Nous allons maintenant montrer la réciproque en construisant l'arbre de syntaxe associé à une dérivation. Notons que cette fois la dérivation est supposée quelconque : son origine est un non-terminal arbitraire, et le mot engendré peut comporter des non-terminaux. On va donc construire des arbres de syntaxe qui ne seront pas nécessairement des arbres de dérivation. On procède par récurrence sur la longueur des dérivations de la façon suivante :

- à une dérivation de longueur nulle, c'est-à-dire à un symbole $\alpha \in V$, on associe un arbre de syntaxe réduit à une feuille étiquetée par α ;
- à une dérivation de longueur $n+1$, de la forme $N \rightarrow^* w_1 \underline{M} w_2 \rightarrow w_1 \alpha_1 \dots \alpha_n w_2$, on associe l'arbre de syntaxe obtenu à partir de l'arbre de syntaxe de la dérivation $N \rightarrow^* w_1 M w_2$ en ajoutant, à la feuille étiquetée par M, n feuilles étiquetées respectivement par $\alpha_1, \dots, \alpha_n$.

On en déduit la propriété fondamentale suivante :

Théorème 7.14 *Étant donnée une grammaire $G = (V_t, V_n, S, R)$, un mot $u \in V_t^*$ appartient à $\mathcal{L}ang(G)$ si et seulement s'il possède un arbre de dérivation dans la grammaire G.*

On remarque facilement que des dérivations qui s'échangent par des permutations adéquates ont le même arbre de dérivation. On peut donc reformuler notre définition d'ambiguïté de la manière suivante :

Définition 7.15 *Une grammaire G est ambiguë s'il existe un mot $u \in \mathcal{L}ang(G)$ qui possède deux arbres de dérivation distincts dans la grammaire G.*

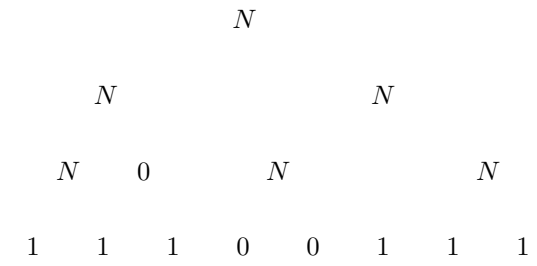
La figure 7.3 confirme que notre première grammaire des expressions arithmétiques était ambiguë, comme cela est toujours le cas avec des opérateurs binaires infixes, comme $+$ et $*$, à moins d'inclure les règles de priorité des opérateurs dans la grammaire, lorsqu'il y en a. Dans ce cas, la grammaire ambigue peut toujours être transformée en une grammaire non ambigüe comme nous l'avons fait pour la grammaire des expressions arithmétiques.

7.7 Exercices

Exercice 7.2 *On considère la grammaire suivante :*

$$G = (\{0,1\}, \{N\}, N, \{N \to 11 \mid 1001 \mid N0 \mid NN\})$$

1. *Dessiner les arbres de dérivation pour les mots* 11011 *et* 11110*.*
2. *Est-ce que la grammaire est ambiguë ?*
3. *On regarde l'arbre syntaxique suivant :*



Donner la dérivation gauche et la dérivation droite pour cet arbre.

Exercice 7.3 *On considère le langage des expressions arithmétiques avec addition, opérateur infixe noté* $+$*, soustraction, opérateur infixe noté* $-$*, opposé, opérateur unaire préfixe noté* $-$*, multiplication, opérateur binaire noté* $*$*, division, opérateur binaire infixe noté* $/$*, et exponentiation, opérateur binaire noté* $\uparrow$*. Les opérateurs binaires associent tous à gauche sauf l'exponentiation qui associe à droite, et l'ordre de priorité est* $\uparrow > -$*(opposé)* $> \{*, /\} > \{+, -$*(soustraction)*$\}$*.*

1. *Donner le bon parenthésage des mots* $3 - 2 - 1$ *et* $2 \uparrow 3 \uparrow 4$*.*
2. *Donner le bon parenthésage du mot* $x + -y \uparrow 2 - 3 * x$*.*
3. *Donner une grammaire non-ambiguë décrivant ce langage.*

Exercice 7.4
1. *Spécifier un type abstrait « arbre de syntaxe », avec ses fonctions de construction et d'accès.*
2. *Spécifier puis réaliser une fonction de parcours gauche et une fonction de parcours droit.*
3. *Coder ce type abstrait et ces fonctions de parcours en CAML puis en PASCAL.*

Exercice 7.5
1. *Construire l'arbre de dérivation du mot Id+Cte×Cte+(Id+Id) dans la grammaire ambiguë des expressions arithmétiques, correspondant au parenthésage* $((x + (2.5 \times 4)) + (y + z))$*.*
2. *Construire l'arbre de dérivation correspondant à la dérivation gauche du mot Id+Cte×Cte+(Id + Id) dans la grammaire désambiguïsée des expressions arithmétiques.*

Exercice 7.6 *Un palindrome non trivial est un mot de longueur au moins deux sur l'alphabet* $a - z$ *qui se lit aussi bien de droite à gauche que de gauche à droite. Par exemple,* $eve, anna, otto$ *sont des palindromes. Une suite-palindrome sera un mot sur ce même alphabet obtenu en concaténant plusieurs palindrômes, comme par exemple* $annaeveotto$*.*

1. *Donner une grammaire pour les suites-palindrome.*
2. *Dessiner les arbres de dérivation des mots* $annaeveotto$ *et* $aaaa$
3. *Est-ce que la grammaire est ambiguë ?*
4. *Peut-on trouver une grammaire non ambiguë pour le langage ?*

Chapitre 8

Automates à pile

Ce sont les machines utilisées pour la reconnaissance des langages hors-contexte.

8.1 Langages reconnaissables par automates à pile

Un automate à pile est une machine munie d'une bande de lecture, d'une bande de travail organisée en pile, et d'un contrôle comme indiqué à la figure 8.1.

FIG. 8.1 – L'automate à pile.

Définition 8.1 *Un* automate à pile non-déterministe $\mathcal{A}$ *est un quadruplet* (V_t, Q, V_P, T) *où*

1. V_t *est le* vocabulaire *de l'automate ;*
2. Q *est l'ensemble des* états *de l'automate ;*
3. V_P *est le* vocabulaire *de pile de l'automate ;*
4. $T\ :\ Q \times (V_t \cup \{\varepsilon\}) \times V_P \to \mathcal{P}(V_P^* \times Q)$ *est la* fonction de transition *de l'automate.*

L'automate sera dit déterministe *s'il vérifie la condition :*
$\forall q \in Q\ \forall X \in V_P\ \forall a \in V_t,\ |T(q, \varepsilon, X) \cup T(q, a, X)| \leq 1$ *;*
L'automate sera dit complet *s'il vérifie la condition :*
$\forall q \in Q,\ X \in V_P,\ a \in V_t,\ |T(q, \varepsilon, X) \cup T(q, a, X)| \geq 1$ *;*
On notera $q \xrightarrow{a,X,\alpha} q'$ *pour* $q' \in T(q, a, X)$ *avec* $X \in V_P,\ \alpha \in V_P^*$ *et* $a \in (V_t \cup \{\varepsilon\})$.

Lexicographie : on utilisera autant que faire se peut les lettres $a, b, c, \ldots$ pour les lettres de V_t, les lettres $u, v, w, \ldots$ pour les mots de V_t^*, les lettres X, Y, Z pour les lettres de V_P et les lettres $\alpha, \beta, \gamma, \ldots$ pour les mots de V_P^*. Il arrivera toutefois dans les exemples que l'on utilise les même lettres pour V_t et V_P.

La notion de calcul pose un petit problème : celui de faire tomber la pile à gauche ou à droite. Dans ce cours, on la fera généralement tomber à droite, et on écrira donc le mot (non vide) de pile sous la forme αX, mais on ne s'interdira pas de la faire tomber à gauche lorsque cela se révèlera plus commode, en l'écrivant sous la forme $X\alpha$. La différence restera bien sûr invisible lorsque l'on ne fait pas apparaître explicitement le sommet de pile.

Définition 8.2 *Étant donné un automate* $\mathcal{A} = (V_t, Q, V_P, T)$*, on appelle* configuration *toute paire* (q, α) *formée d'un état* $q \in Q$ *de l'automate et d'un mot de pile* $\alpha \in V_P^*$*, et* transition *de l'automate la relation entre configurations notée* $(q, \alpha) \xrightarrow{a,X,\beta} (q', \alpha')$*, où* $a \in V_t, X \in V_P, \beta \in V_t^*$*, telle que (i)* $(q', \beta) \in T(q, a, X)$*, (ii)* $\alpha = \gamma X$*, et (iii)* $\alpha' = \gamma\beta$

Dans le cas où l'on fait tomber la pile à gauche, on écrira une transition sous la forme $(q,\alpha)\xrightarrow{a,X,\beta}(q',\alpha')$, la condition (ii) devenant (ii') $\alpha = X\gamma$, et la condition (iii) devenant $\alpha' = \beta\gamma$.

Définition 8.3 *Étant donné un automate $\mathcal{A} = (V_t, Q, V_P, T)$, on appelle* calcul *d'origine $(q_0 \in Q, \alpha_0 \in V_t^+)$, toute suite $(q_0,\alpha_0)\xrightarrow{a_1,X_1,\beta_1}(q_1,\alpha_1)\ \ldots\ (q_{n-1},\alpha_{n-1})\xrightarrow{a_n,X_n,\beta_n}(q_n,\alpha_n)$ éventuellement vide de transitions.*
L'entier n est la longueur *du calcul et $a_0a_1\ldots a_n$ est le mot* lu *par le calcul, ce que l'on notera en écriture abrégée sous la forme $(q_0,\alpha_0)\xrightarrow{a_0\ldots a_n}(q_n,\alpha_n)$.*

Il sera parfois commode de noter un calcul sous une forme un peu différente, le contenu de pile apparaissant au dessus de la flèche figurant la transition, une transition reliant alors des états plutôt que des configurations :
$q_0\xrightarrow{a_1,\alpha_1X_1,\beta_1}q_1\ldots\ldots q_{n-1}\xrightarrow{a_n,\alpha_nX_n,\beta_n}q_n$.
Dans le cas où l'on fait tomber la pile à gauche, cette écriture commode devient
$q_0\xrightarrow{a_1,X_1\alpha_1,\beta_1}q_1\ldots\ldots q_{n-1}\xrightarrow{a_n,X_n\alpha_n,\beta_n}q_n$.
On peut maintenant passer à la notion de reconnaissance. En fait, il y en a plusieurs, toutes équivalentes, qui utilisent différentes conditions d'acceptation.

Définition 8.4 *On dit que le mot $u = u_1\ldots u_n$ est* reconnu *par l'automate $\mathcal{A}^F_{i,\phi} = (V_t, Q, i, F, V_P, \phi, T)$, où $\phi \in V_P$ est appellé* fond de pile *de l'automate, i est son état initial et $F \subseteq Q$ est l'ensemble des états acceptants, s'il existe un calcul $(i,\phi)\xrightarrow{u_1\ldots u_n}(q_n,\alpha_n)$ d'origine (i,ϕ) tel que :*

1. *reconnaissance par état final : $q_n \in F$;*
2. *reconnaissance par pile vide : $\alpha_n = \varepsilon$;*

On note par $\mathcal{L}ang(\mathcal{A})$ le langage des mots reconnus par l'automate $\mathcal{A}^F_{i,\phi}$ qui sera souvent noté simplement A.

On peut également combiner les deux conditions d'acceptation ce qui fournit 3 conditions d'acceptation différentes. Notons aussi l'importance du symbole de fond de pile : la première transition de l'automate nécessite la lecture d'un symbole dans la pile, qui est donc initialisée avec ϕ.

Exemple 8.5 $L = \{a^nb^ma^n \mid n,m \geq 1\} \cup \{a^mb^na^n \mid n,m \geq 1\}$ *L'automate reconnaissant ce langage par pile vide est représenté à la figure 8.2.*

FIG. 8.2 – Automate non déterministe reconnaissant le langage $\{a^nb^na^m \mid n,m \geq 1\} \cup \{a^nb^ma^n \mid n,m \geq 1\}$ par pile vide.

L'automate représenté à la figure 8.3 reconnait ce même langage par état final à seul.

FIG. 8.3 – Automate non déterministe reconnaissant le langage $\{a^nb^na^m \mid n,m \geq 1\} \cup \{a^nb^ma^n \mid n,m \geq 1\}$ par état final et fond de pile.

Notons qu'il est toujours possible de supposer que le symbole de fond de pile se trouve en permanence en fond de pile (sauf éventuellement tout à la fin en cas de reconnaissance par pile vide) et jamais ailleurs. En effet, étant donné un automate $\mathcal{A} = (V_t, Q, i, F, V_P, \phi, T)$, on construit aisément un automate $\mathcal{A}' = (V_t, Q \cup \{i'\}, i', F, V_P \cup \{\phi'\}, \phi', T')$ qui a la propriété tout en reconnaissant le même langage. On définit T' comme suit :

$$\begin{array}{rlcll} & T'(i',\varepsilon,\phi') & = & (i,\phi'\phi) & \\ \forall q \in Q\ \forall a \in V_t \cup \{\varepsilon\} \forall X \in V_P & T'(q,a,X) & = & T(q,a,X) & \\ \forall q \in Q & T'(q,\varepsilon,\phi') & = & (q,\varepsilon) & \text{si reconnaissance par pile vide} \end{array}$$

Théorème 8.6 *Les modes de reconnaissance sont équivalents pour les automates non déterministes. On note par $\mathcal{A}lg$ la classe des langages reconnaissables par un automate à pile.*

On choisit la reconnaissance par état final pour les automates déterministes.

Preuve: Soit $\mathcal{A} = (V_t, Q, i, F, V_P, \phi, T)$ un automate à pile reconnaissant par état final. On construit aisément un automate reconnaissant à la fois par état final et pile vide : il suffit pour cela d'ajouter de nouvelles transitions en état final de manière à vider la pile. Toutefois, cette transformation ne préserve pas le déterminisme.

Soit maintenant $\mathcal{A} = (V_t, Q, i, F, V_P, \phi, T)$ un automate à pile reconnaissant par pile vide. On construit aisément un automate reconnaissant à la fois par état final et pile vide, en ajoutant un nouvel état f, un nouveau symbole de fond de pile comme précédemment, puis les transitions :

(i) $T'(i, \phi') = (i, \phi'\phi)$;

(ii) $\forall q \in Q\ \forall a \in V_t \cup \{\varepsilon\}\ \forall X \in V_P\ T'(q, a, X) = T(q, a, X)$;

(iii) $\forall q \in Q\ T'(q, \phi') = (f, \varepsilon)$.

Notons que ces transformations conservent le déterminisme. □

Comme le déterminisme est conservé en passant de la reconnaissance par pile vide à celle par état acceptant, la bonne notion de reconnaissance par un automate déterministe, c'est-à-dire celle qui qui autorise la plus grande classe de langages reconnus, est basée sur la reconnaissance par état final.

8.2 Automates à pile et grammaires hors-contexte

Théorème 8.7 *Un langage est hors-contexte si et seulement si il est reconnaissable par un automate à pile.*

Afin de bien comprendre la correspondance entre la génération par une grammaire et la reconnaissance par un automate, nous allons étudier un exemple. On se donne tout d'abord une grammaire en forme normale de Chomsky qui engendre le langage $\{a^n b^n \mid n \geq 0\}$, par exemple la grammaire

$$G = (\{a, b\}, \{S, A, B, C\}, S, \{S \to \varepsilon, S \to AC, C \to SB, A \to a, B \to b\})$$

et nous allons construire une dérivation qui engendre le mot a^3b^3 :

$$\begin{gathered} S \to AC \to aC \to aSB \to aACB \to aaCB \\ \to aaSBB \to aaACBB \to aaaCBB \to aaaSBBB \\ \to aaaBBB \to aaabBB \to aaabbB \to aaabbb \end{gathered}$$

Pour reconnaitre le langage engendré, l'idée est de construire un automate à pile non-déterministe qui calque le calcul fait par la grammaire : à chaque étape, par exemple à l'étape qui termine la première ligne de notre dérivation, le préfixe maximal $v \in V_t^*$ du mot engendré à cette étape, dans notre cas le mot a^3, sera très exactement le préfixe déja reconnu à cette étape du mot u à reconnaitre, dans notre exemple le mot a^3b^3, alors que le reste $w \in V_n^*$ du mot, dans notre cas le mot CBB, sera le contenu de pile à cette étape de la reconnaissance. Notre automate, en adoptant la convention de la pile qui tombe à gauche, sera donc

$$\mathcal{A} = (\{a, b\}, \{q\}, q, \{S, A, B, C\}, S, \{q \xrightarrow{\varepsilon,S,\varepsilon} q, q \xrightarrow{\varepsilon,S,AC} q, q \xrightarrow{\varepsilon,C,SB} q, q \xrightarrow{a,A,\varepsilon} q, q \xrightarrow{b,B,\varepsilon} q\}).$$

et la reconnaissance du mot a^3b^3 donnera les calculs suivants :

$$\begin{gathered} (q, S) \xrightarrow{\varepsilon,S,AC} (q, AC) \xrightarrow{a,A,\varepsilon} (q, C) \xrightarrow{\varepsilon,C,SB} (q, SB) \xrightarrow{\varepsilon,S,AC} (q, ACB) \xrightarrow{a,A,\varepsilon} (q, CB) \\ \xrightarrow{\varepsilon,C,SB} (q, SBB) \xrightarrow{\varepsilon,S,AC} (q, ACBB) \xrightarrow{a,A,\varepsilon} (q, CBB) \xrightarrow{\varepsilon,C,SB} (q, SBBB) \\ \xrightarrow{\varepsilon,S,\varepsilon} (q, BBB) \xrightarrow{b,B,\varepsilon} (q, BB) \xrightarrow{b,B,\varepsilon} (q, B) \xrightarrow{b,B,\varepsilon} (q, \varepsilon) \end{gathered}$$

Notons qu'il n'est pas utile de choisir la grammaire de départ en forme normale de Chomsky, on peut faire le même raisonnement avec une grammaire quelconque, dans laquelle les ocurrences des symboles terminaux des membres droits de règles sont transformées en non-terminaux, à l'exception de l'occurrence du terminal par lequel débute évenrtuellement le membre droit.

$$G = (\{a, b\}, \{S, B\}, S, \{S \to \varepsilon, S \to aSB, B \to b\})$$

$$\begin{gathered} S \to aSB \to aaSBB \to aaaSBBB \to aaaBBB \\ \to aaabBB \to aaabbB \to aaabbb \end{gathered}$$

$$\mathcal{A} = (\{a, b\}, \{q\}, q, \{S\}, S, \{q \xrightarrow{\varepsilon,S,\varepsilon} q, q \xrightarrow{a,S,SB} q, q \xrightarrow{b,B,\varepsilon} q\}).$$

$$\begin{gathered} (q, S) \xrightarrow{a,S,aSB} (q, SB) \xrightarrow{a,S,aSB} (q, SBB) \xrightarrow{a,S,aSB} (q, SBBB) \xrightarrow{\varepsilon,S,\varepsilon} (q, BBB) \\ \xrightarrow{b,B,\varepsilon} (q, BB) \xrightarrow{b,B,\varepsilon} (q, B) \xrightarrow{b,B,\varepsilon} (q, \varepsilon) \end{gathered}$$

On peut maintenant faire la preuve qu'à toute grammaire hors-contexte correspond un automate à pile qui reconnait le même langage. Cet automate a un unique état duquel partent et auquel arrivent toutes les transitions, que l'on appellera l'automate marguerite.

Preuve: On suppose tout d'abord donnée la grammaire $G = (V_t, V_n, S, R)$, et nous allons construire un automate à pile $\mathcal{A} = (V_t, Q = \{q\}, q, V_n, S, T)$ qui reconnait par pile vide le langage des mots engendrés par la grammaire.

À toute règle de la forme $N \to \varepsilon$, on fait correspondre la transition $\xrightarrow{\varepsilon,N,\varepsilon}$.
À toute règle de la forme $N \to a\alpha$, on fait correspondre la transition $\xrightarrow{a,N,\alpha}$.
À toute règle de la forme $N \to X\alpha$, on fait correspondre la transition $\xrightarrow{\varepsilon,N,X\alpha}$.

L'automate obtenu reconnait bien le langage $\mathcal{L}(G)$. Il a sa pile qui est tombée à gauche. Une récurrence simple montre que les dérivations dans la grammaire correspondre très exactement aux calculs de l'automate. □

La transformation est plus complexe dans le sens inverse, et sans intérêt pratique : nous l'omettrons.

8.3 Exercices

Exercice 8.1 *Peut-on compléter un automate non déterministe sans changer le langage reconnu ? un automate déterministe ? l'automate obtenu dans ce cas est-il encore déterministe ?*

Exercice 8.2 *Montrer que pour tout automate $\mathcal{A}$ reconnaissant par pile vide et état final en même temps, il existe un automate $\mathcal{A}'$ possédant un unique état final qui reconnait le même langage par pile vide et état final en même temps.*

Chapitre 9

Propriétés de clôture des langages algébriques

9.1 Vide et finitude des langages algébriques

On opère à partir des grammaires hors-contexte associées, pour lesquelles le problème a déja été abordé.

Lemme 9.1 *Le langage reconnu par un automate à pile $\mathcal{A}$ est non vide ssi la grammaire associée mise sous forme propre est non vide.*

Preuve: C'est une conséquence du théorème 8.7 et du Corollaire 7.10. □

9.2 Pompage des langage algébriques

9.2.1 Propriété de pompage des algébriques

Revenons à notre grammaire de règles $S \to \varepsilon, S \to aSb$ engendrant les mots de la forme $a^n b^n$, et considérons l'arbre syntaxique du mot ab :

Le long du chemin central, le non-terminal S apparait deux fois. On peut donc obtenir de nouveaux arbres syntaxiques pour la même grammaire en supprimant, ou en répetant un nombre arbitraire de fois le motif central $S(a, , b)$, par exemple

On en déduit alors que le mot a^2b^2 appartient au langage.

Théorème 9.2 *Si L est algébrique, alors il satisfait la propriété*

$P_{alg}(L) \stackrel{\text{def}}{=} \exists n \in \mathbb{N}$ *tel que* $\forall m \in L\ |m| \geq n \implies \exists u, x, v, y, w \in V_t^*$ *tel que les propriétés suivantes soit satisfaites*

1. $m = uxvyw$
2. $|xy| > 0$
3. $|xvy| < n$
4. $\forall i \geq 0\ ux^ivy^iw \in L$

Preuve: Contrairement à ce que nous avons fait pour l'exemple, nous alons choisir une grammaire $G = (V_t, V_n, S, R)$ en forme normale de Chomsky qui engendre le langage L.

Soit N le nombre de non-terminaux dans la grammaire. Sachant que pour un arbre syntaxique de hauteur h, le mot des feuilles est de longueur 2^h au plus □

Comme toujours, les propriétés de pompage ne sont pas caractéristiques d'une classe de langage. Il existe donc des langages non-algébriques qui satisfont la propriété de pompage précédente. Il est bien-sûr possible d'élaborer des propriétés de pompage plus complexes, que des langages non-algébriques ont plus de difficultés à satisfaire, cela fait l'objet d'exercices.

9.2.2 Un peu de logique

L'utilisation essentielle d'une propriété de pompage est de montrer que certains langages ne sont pas algébriques. Puisque tout langage algébrique L satisfait la propriété $P_{alg}(L)$, un langage qui ne la satisfait pas ne peut pas être algébrique.

Ce raisonnement, dit par contraposition, utilise la propriété logique suivante :

$$A \Longrightarrow B \text{ si et seulement si } \neg B \Longrightarrow \neg A$$

Et donc, le théorème de pompage peut également s'exprimer sous la forme équivalente :

Si le langage L satisfait la propriété $\neg P_{alg}(L)$, alors il n'est pas algébrique.

Il nous faut donc calculer la propriété $\neg P_{alg}(L)$, comme nous l'avions fait pour les langages reconnaissables. Pour cela, nous utiliserons plusieurs équivalences logiques élémentaires, qui permettent en particulier de pousser les négations vers l'intérieur d'une formule :

$$\begin{gathered}
A \Longrightarrow B \equiv B \vee (\neg A) \\
\neg\neg A \equiv A \\
\neg(A \wedge B) \equiv (\neg A) \vee (\neg B) \\
\neg(A \vee B) \equiv (\neg A) \wedge (\neg B) \\
\neg(A \Longrightarrow B) \equiv A \wedge (\neg B) \\
\neg\forall A \equiv \exists\neg A \\
\neg\exists A \equiv \forall\neg A
\end{gathered}$$

Ces équivalences permettent de dériver deux autres équivalences simples qui interviennent de manière cruciale dans le calcul qui nous intéresse :

$$\begin{gathered}
(A \wedge B \Longrightarrow C) \equiv A \Longrightarrow (B \Longrightarrow C) \\
\neg(A \wedge B \wedge C \wedge D) \equiv A \wedge B \wedge C \Longrightarrow (\neg D)
\end{gathered}$$

Notons

$$\begin{aligned}
A &\stackrel{\text{def}}{=} (m = uxvyw) \\
B &\stackrel{\text{def}}{=} (|xy| > 0) \\
C &\stackrel{\text{def}}{=} (|xvy| < n) \\
D &\stackrel{\text{def}}{=} (\forall i M) \\
M &\stackrel{\text{def}}{=} N \Longrightarrow O \\
N &\stackrel{\text{def}}{=} (i \geq 0) \\
O &\stackrel{\text{def}}{=} ux^i vy^i w \in L \\
E &\stackrel{\text{def}}{=} (A \wedge B \wedge C \wedge D) \\
F &\stackrel{\text{def}}{=} (\exists u, x, v, y, w \in V_t^*\ E) \\
G &\stackrel{\text{def}}{=} (m \in L) \\
H &\stackrel{\text{def}}{=} (|m| \geq n) \\
I &\stackrel{\text{def}}{=} (G \wedge H) \\
J &\stackrel{\text{def}}{=} \forall m (I \Longrightarrow F) \\
K &\stackrel{\text{def}}{=} (n \in \mathbb{N}) \\
P_{alg}(L) &\stackrel{\text{def}}{=} \exists n\ (K \Longrightarrow J)
\end{aligned}$$

Nous sommes maintenant armés pour calculer $\neg P_{alg}(L)$:

$$\begin{array}{rcl} \neg P_{alg}(L) & = & \forall n\ \neg(K \Longrightarrow J) \\ & = & \forall n\ K \wedge (\neg J) \\ (\neg J) & = & \exists m\ \neg(I \Longrightarrow F) \\ & = & \exists m\ I \wedge (\neg F) \\ (\neg F) & = & (\forall u,x,v,y,w \in V_t^*\ (\neg E)) \\ (\neg E) & = & (A \wedge B \wedge C) \Longrightarrow (\neg D) \end{array}$$

On peut maintenant calculer la propriété $\neg P_{alg}(L)$:

$$\begin{array}{rcl} \neg P_{alg}(L) & = & (\forall n\ (n \in \mathbb{N}) \wedge (\exists m\ I \wedge (\forall u,x,v,y,w \in V_t^*\ (A \wedge B \wedge C) \Longrightarrow (\neg D)))) \\ & = & (\forall n\ (n \in \mathbb{N}) \Longrightarrow (\exists m\ I \Longrightarrow (\forall u,x,v,y,w \in V_t^*\ (A \wedge B \wedge C) \Longrightarrow (\neg D)))) \end{array}$$

D'où finallement :

$\neg P_{alg}(L) = (\forall n\ (n \in \mathbb{N}) \Longrightarrow (\exists m\ (m \in L \wedge |m| \geq n) \Longrightarrow (\forall u,x,v,y,w \in V_t^*\ ((m = uxvyw) \wedge (|xy| > 0) \wedge (|xvy| < N)) \Longrightarrow (\exists i \geq 0\ ux^ivy^iw \notin L))))$

ce qui se récrit de la manière suivante :

$\neg P_{alg}(L) = \forall n \in \mathbb{N}\ \exists m \in L$ tel que $|m| \geq n \Longrightarrow (\forall u,x,v,y,w \in V_t^*$ tel que $m = uxvyw \wedge |xy| > 0 \wedge |xvy| < n,\ \exists i \geq 0$ tel que $ux^ivy^iw \notin L)$

Pour prouver qu'un langage n'est pas algébrique, il suffit donc de prouver qu'il satisfait la propriété $\neg P_{alg}(L)$. C'est ce que nous allons faire pour le langage $L = \{a^ib^ic^i \mid i \geq 0\}$, dont nous allons prouver qu'il satisfait la propriété $\neg P_{alg}(L)$:

Soit n un entier positif arbitraire, $m = a^nb^nc^n$ et $m = uxvyw$ un découpage de m tel que $|xy| > 0$ et $|xvy| < n$. La condition $|xvy| < n$ implique que $xvy = a^kb^l$ (ou bien, symétriquement, b^kc^l), avec $0 \leq k,\ 0 \leq l$ et $0 < k+l < n$, et donc, comme $|xy| > 0$, $v = a^{k'}b^{l'}$ avec $k'+l' < k+l$. Prenons alors $i = 0$: le mot $ux^0vy^0w = uvw = a^{n-k+k'}b^{n-l+l'}c^n$ n'appartient pas à L car si $k' = k$, alors $l' < l$. Donc, L satisfait la propriété $\neg P_{alg}(L)$, et n'est donc pas algébrique.

9.3 Propriétés de clôture

Nous allons regrouper ces propriétés en plusieurs catégories :

Union, produit et étoile de Kleene. Comme les rationels, les algébriques sont clos par ces trois propriétés, les preuves travaillant cette fois sur les grammaires.

Théorème 9.3 *Les langages algébriques sont clos par union.*

Preuve: On se donne un langage L_1 engendré par la grammaire hors-contexte $G_1 = (V_t, V_n^1, S_1, R_1)$ et un langage L_2 engendré par la grammaire hors-contexte $G_2 = (V_t, V_n^2, S_2, R_2)$. On suposera, au prix d'un renommage éventuel des non-terminaux de G_2 que $V_n^1 \cap V_n^2 = \emptyset$, et que $S \notin (V_n^1 \cup V_n^2)$.

On définit alors la grammaire $G = (V_t, V_n^1 \cup V_n^2 \cup \{S\}, S, R_1 \cup R_2 \cup \{S \to S_1, S \to S_2\})$, pour laquelle il est immédiat de montrer qu'elle engendre le langage union de L_1 et L_2. □

Théorème 9.4 *Les langages algébriques sont clos par produit.*

Preuve: On se donne un langage L_1 engendré par la grammaire hors-contexte $G_1 = (V_t, V_n^1, S_1, R_1)$ et un langage L_2 engendré par la grammaire hors-contexte $G_2 = (V_t, V_n^2, S_2, R_2)$. On suposera, au prix d'un renommage éventuel des non-terminaux de G_2 que $V_n^1 \cap V_n^2 = \emptyset$, et que $S \notin (V_n^1 \cup V_n^2)$.

On définit alors la grammaire $G = (V_t, V_n^1 \cup V_n^2 \cup \{S\}, S, R_1 \cup R_2 \cup \{S \to S_1S_2\})$, pour laquelle il est immédiat de montrer qu'elle engendre le langage produit de L_1 par L_2. □

Théorème 9.5 *Les langages algébriques sont clos par étoile.*

Preuve: On se donne un langage L_1 engendré par la grammaire hors-contexte $G_1 = (V_t, V_n^1, S_1, R_1)$. On suposera, au prix d'un renommage éventuel des non-terminaux de G_1 que $S \notin (V_n^1 \cup V_n^2)$.

On définit alors la grammaire $G = (V_t, V_n^1 \cup \{S\}, S, R_1 \cup \{S \to \varepsilon, S \to S_1 S\})$, pour laquelle il est immédiat de montrer qu'elle engendre l'itéré de Kleene L_1^* du langage L_1. □

Intersection et complémentation. À l'inverse des rationels, les algébriques ne sont clos ni par intersection, ni par complémentation. Seraient-ils clos par l'une de ces deux opérations qu'ils seraient clos par l'autre, puisqu'ils sont clos par union, et qu'union, complémentation et intersection sont liées par la célèbre loi de Morgan rappellée plus loin.

Théorème 9.6 *Les langages algébriques ne sont pas clos par intersection ni par complémentation.*

Preuve: On montre la non-clôture par intersection. Pour cela, on considère les deux langages $L_1 = \{a^i b^j c^j \mid i, j \geq 0\}$ et $L_2 = \{a^i b^i c^j \mid i, j \geq 0\}$ dont l'intersection est le langage $L = \{a^i b^i c^i \mid i \geq 0\}$ dont a vu au paragraphe 9.2 qu'il n'était pas algébrique.

Il nous suffit donc de montrer que les langages L_1 et L_2 sont algébriques, ce qui a déja été fait.

Supposons maintenant (c'est un raisonnement par l'absurde) que les algébriques soient clos par complémentation. Alors, étant donnés deux langages algébriques quelconques L_1 et L_2 (par exemple, les deux langages précédents), leurs complémentaires $\overline{L_1}$ et $\overline{L_2}$ le seraient également, et donc, par application de la clôture par union et de la loi de De Morgan

$$\overline{L_1 \cap L_2} = \overline{L_1} \cup \overline{L_2}$$

le langage $\overline{L_1 \cap L_2}$ le serait également. Par application de notre hypothèse, son complémentaire $\overline{\overline{L_1 \cap L_2}} = L_1 \cap L_2$ le serait aussi, ce qui contredit le résultat précédent. □

Par contre,

Théorème 9.7 *Les langages algébriques sont clos par intersection avec un rationel.*

Preuve: La preuve est très similaire à celle faite pour l'intersection de deux rationnels : étant donnés un automate à pile $A_1 = (V_t, Q_1, i_1, F_1, V_P, \phi, T_1)$ reconnaissant par état final, et un automate fini $A_2 = (V_t, Q_2, i_2, F_2, T_2)$ sur le même alphabet V_t, on construit l'automate à pile produit

$A = (V_t, Q_1 \times Q_2, (i_1, i_2), F \times F', T)$ où $T((q_1, q_2), a, X) = \{(T(q_1, a, X), T(q_2, a)\}$

Notons que l'automate produit a pour effet de synchroniser les transitions, donc les calculs des automates de départ, il reconnait $\mathcal{L}ang(A_{q_0}^F) \cup \mathcal{L}ang(A_{q_0'}^{'F'})$. □

9.3.1 Substitution et homomorphisme

Théorème 9.8 *Les langages algébriques sur un alphabet V_t sont clos par substitution des lettres de l'alphabet V_t par des langages algébriques, chaque lettre $a \in V_t$ étant remplacée par un langage albébrique L_a sur un alphabet V_t^a.*

Preuve: La preuve se fait aisément sur les grammaires. □

On en déduit le corollaire :

Théorème 9.9 *Les langages algébriques sont clos par homophisme.*

9.3.2 Homomorphisme inverse

Théorème 9.10 *Les langages algébriques sont clos par homomorphisme inverse.*

Preuve: La peuve a cette fois lieu sur les automates. □

9.3.3 Applications

Comme pour les rationnels, les propriétés de clôture servent tout autant à montrer que certains langages sont algébriques sans avoir à en donner une grammaire les engendrant ou un automate à pile les reconnaissant, qu'à montrer que certains langages sont ne sont pas algébriques alors même qu'ils satisfont le Lemme de la pompe.

9.4 Exercices

Chapitre 10

Analyse syntaxique

Le but de ce chapitre est de continuer l'étude entreprise dans le chapitre précédent, concernant le passage, appelé *analyse syntaxique*, d'une grammaire hors-contexte à un automate qui reconnait le langage engendré.

Plus précisément l'objectif est d'introduire le lecteur aux techniques d'analyse syntaxique et à l'utilisation du logiciel YACC. L'analyse syntaxique est un processus techniquement plus complexe que l'analyse lexicale, dont les objets mathématiques sous-jacents sont les automates à pile.

Il existe en fait deux façons bien différentes de concevoir l'analyse syntaxique d'un mot d'un langage décrit par une grammaire hors contexte. Lors de la lecture du mot (de la gauche vers la droite), l'arbre syntaxique peut être construit de haut en bas, ou de bas en haut, conformément à la figure 10.1. Dans le premier cas, l'analyse conctruit des arbres de syntaxe de racine S dont les feuilles sont étiquettées par les symboles de V. Dans le second, elle engendre des suites d'arbres de syntaxe dont les racines sont des mots sur V et les feuilles des mots sur VT.

Nous allons maintenant illustrer ces deux méthodes sur des exemples et en tirer de méthodes générales.

10.1 Analyse syntaxique descendante

Exemple 10.1 *Considérons la grammaire de Dick :*

$$S \quad \rightarrow \quad \varepsilon \mid (S)S$$

L'analyse descendante correspond à une dérivation gauche. Par exemple, pour le mot $(((())))()$*, la dérivation gauche est la suivante :*

$$S \rightarrow (S)S \rightarrow ((S)S)S \rightarrow (((S)S)S)S \rightarrow ((()S)S)S \rightarrow (((()(S)S)S)S \rightarrow (((()()S)S)S$$

$$\rightarrow (((()))S)S \rightarrow ((()))(S)S)S \rightarrow ((()))()S)S \rightarrow ((()))())S \rightarrow ((()))())$$

L'analyse syntaxique descendante va construire la suite des arbres d'analyse correspondant à cette dérivation, représentée à la figure 10.2.

□

FIG. 10.1 – Méthodes d'analyse syntaxique

FIG. 10.2 – Analyse syntaxique descendante

10.1.1 Analyse descendante non-déterministe

Il s'agit en fait de construire l'automate déjà vu qui transforme une grammaire hors-contexte en un automate à pile non-déterministe, l'automate marguerite. Nous renvoyons donc au chapitre qui précède.

10.1.2 Analyse descendante déterministe

L'automate marguerite est déterministe en de rares occasions : si toutes les règles de membre gauche $N \in V_P$ sont de la forme $N \to \varepsilon$ ou $N \to a\alpha$ avec la condition que pour chaque $N \in V_n$ et $a \in V_t$, il existe au plus une règle de membre gauche N et dont le membre droit est vide ou commence par la lettre a.

Cette condition étant trop restrictive, on va essayer de prédire le bon choix à chaque étape de l'analyse, c'est pourquoi la méthode que nous allons discuter est aussi appellée analyse descendante prédictive.

Le choix entre plusieurs règles applicables peut se faire grâce à la connaissance du prochain caractère devant être engendré, appelé le *caractère d'avance*. Au départ, pour notre example, ce caractère est la parenthèse ouvrante, et la première règle à appliquer doit donc engendrer une parenthèse ouvrante. Il se trouve que la seconde règle engendre une parenthèse ouvrante, et ce sera donc elle que l'on appliquera chaque fois qu'une parenthèse ouvrante devra être engendrée. De son coté, la première règle fait disparaître le non-terminal S. Or, dans le membre droit de la seconde règle, la première occurrence du non-terminal S est placée devant une parenthèse fermante. L'application de la première règle à une telle occurrence de S engendre donc indirectement une parenthèse fermante. Il en est de même de la seconde occurrence de S (il faut pour s'en rendre compte regarder les arbres syntaxiques) sauf dans le cas où il s'agit de la dernière telle occurrence de S à remplacer avant de terminer (et alors il faut ne rien engendrer du tout, car le mot à lire est entièrement lu). Le choix de l'application d'une règle est donc entièrement déterminé par la connaissance du caractère d'avance. Une telle grammaire est dite LL(1), le premier L étant l'initiale du mot anglais Left, qui indique que la lecture du mot à analyser se fait de gauche à droite, le second L, initiale du même mot anglais, indiquant que l'on construit une dérivation gauche, et le 1 indiquant l'utilisation d'un unique caractère d'avance. Toutes les grammaires ne sont pas LL(1), certaines peuvent être LL(2), sans être LL(1), mais seul le cas LL(1) est utilisé en pratique. Se rendre compte qu'une grammaire est LL(1) demande le calcul des premiers caractères engendrés par l'utilisation de chaque règle. Par exemple, dans la grammaire de Dick, le premier caractère engendré par l'utilisation de la seconde règle est la parenthèse ouvrante. Notons qu'une grammaire LL(1) est nécessairement non-ambiguë, puisque le choix d'une règle lors de l'analyse est entièrement déterminé par la connaissance du caractère d'avance.

Afin d'obtenir un traitement uniforme de tous les cas, il est utile d'augmenter la grammaire considérée $G = (V_t, V_n, S, \mathcal{P})$ par un nouveau terminal f et un nouveau non-terminal S' qui devient la source de la grammaire augmentée

$$G' = (V'_t = V_t \cup \{f\}, V'_n = V_n \cup \{S'\}, S', \mathcal{P}' = \mathcal{P} \cup \{S' \to Sf\})$$

Définition 10.2 *Soit $S' \longrightarrow_g^* u\alpha \longrightarrow_g^* uav$ une dérivation gauche telle que $u \in V_t^*, \alpha \in V'^*, a \in V'_t$, et $v \in V_t'^*$. Alors a est appellé* caractère d'avance *de la sous-dérivation $\alpha \longrightarrow_g^* av$.*

Notons tout d'abord que le caractère d'avance existe toujours, puisque nous avons pris la précaution de prendre $S' \to Sf$ comme origine de la dérivation et que $u \in V_t^*$. Ce caractère d'avance a plusieurs origines possibles :

- si $\alpha = a\beta$ avec $a \in V_t'$, le caractère d'avance a est le premier caractère de la chaîne α ;
- si $\alpha = N\beta$ avec $N \in V_n$ et $N \longrightarrow_g^* a\gamma$, le caractère d'avance est engendré par N ;
- si $\alpha = N\beta$ avec $N \in V_n$, $N \longrightarrow_g^* \varepsilon$ et $\beta \longrightarrow_g^* a\gamma$, le caractère d'avance est engendré par β.

D'où les définitions :

Définition 10.3 *On définit les ensembles suivants :*

$$First(N \in V') = \{a \in V_t' \mid \exists \beta \in V'^* \ N \xrightarrow[g]{*} a\beta\}$$

$$First(\alpha \in V'^*) = \{a \in V_t' \mid \exists \beta \in V'^* \ \alpha \xrightarrow[g]{*} a\beta\}$$

$$Follow(N \in V) = \{a \in V_t' \mid P \to \alpha N \beta \text{ et } a \in First(\beta), \text{ ou } \beta \xrightarrow{*} \varepsilon \text{ et } a \in Follow(P)\}$$

On peut bien sûr calculer aisément $First(N)$ comme un point fixe sur l'ensemble fini V_t'. On peut ensuite en déduire $First(\beta)$ pour tout β, et donc pour tout ensemble fini de mots. On peut donc finalement également calculer $Follow(N)$ ou $Follow(a)$ à nouveau comme des points fixes sur V_t'. Ces calculs utilisent un algorithme pour décider si un mot $\beta \in V'^*$ peut engendrer le mot vide. Ce sont là les différentes briques de base de l'analyse descendante déterministe. Les méthodes de calcul par point fixes qui sont nécéssaires à leur obtention seront vues en exercice, on les supposera connues.

Le lemme qui suit justifie les notions précédentes :

Lemme 10.4 *Soit $S' \longrightarrow_g Sf \longrightarrow_g^* uN\alpha \longrightarrow_g^* uav$ une dérivation gauche de S' vers $uav \in V_t'^*$. Alors $a \in First(N)$ ou sinon $a \in Follow(N)$.*

Ce lemme eprime que l'échec est assuré si le caractère d'avance $a \notin (First(N) \cup Follow(N))$. La preuve est à faire en exercice. Il permet d'anticiper un échec de l'analyse, mais ne préjuge pas de la possibilité de réussir l'analyse de manière déterministe. Pour cela, nous devrons nous assurer que la dérivation qui engendre le caractère à l'avance est de la forme $N \to w \longrightarrow^* av$ où w est unique.

Exemple 10.5 *On considère la grammaire*

$$\begin{array}{rcl} S' & \to & Sf \\ S & \to & aSaTb \mid bT \mid c \\ T & \to & dTK \mid bKc \mid a \\ K & \to & SS \end{array}$$

Construisons la dérivation gauche (on récrit donc en permanence le non-terminal le plus à gauche) du mot $abdbcccbbccccaabf$:

$S' \to Sf \to aSaTbf \to abTaTbf \to abdTKaTbf \to abdbKcKaTbf \to abdbSSKaTbf \to abdbcScKaTbf \to abdbcccKaTbf \to abdbcccSSaTbf \to abdbcccbTSaTbf \to abdbcccbbKcSaTbf \to abdbcccbbSScSaTbf \to abdbcccbbcScSaTbf \to abdbcccbbcccSaTbf \to abdbcccbbccccaTbf \to abdbcccbbccccaabf$

On voit que la reconnaissance du mot u est rendue déterministe par la connaissance du caractère d'avance. Revenons à notre automate non-déterministe. Le choix de la règle à appliquer lorsque le non-terminal N est en sommet de pile va donc se faire par le caractère d'avance, c'est-à-dire le premier caractère de u

Notons que $First(aSaTb) = \{a\}, First(bT) = \{b\}, First(c) = \{c\}, First(a) = \{a\}, First(bKc) = \{b\}, First(dTK) = \{d\}$ et $First(SS) = \{a, b, c\}$. La déterminisation de la procédure initiale a pu avoir lieu parce que les initiales de deux membres droits distincts de règles de même membre gauche avaient une intersection vide. Si ce n'était pas le cas, la même démarche produirait un analyseur qui resterait partiellement non-déterministe.

Nous terminons ce paragraphe par un dernier exemple contenant des productions vides. Nous adopterons un mécanisme plus systématique que précedemment, consistant à empiler l'intégralité des membres droits. Ne pas empiler le premier caractère du membre droit dans le cas où il s'agit d'un terminal apparaît alors comme une optimisation.

Exemple 10.6 *On considère la grammaire*

$$\begin{array}{lcl} S & \rightarrow & UaTb \mid b \\ U & \rightarrow & c \mid \varepsilon \\ T & \rightarrow & YU \mid aTe \\ Y & \rightarrow & d \mid \varepsilon \end{array}$$

pour laquelle on obtient les calculs suivants :

$$First(UaTb) = \{c, a\}, First(YU) = \{d, c\}, First(aTe) = \{a\}, First(x) = \{x\} \text{ pour } x \in V_t$$

$$Follow(U) = \{a, e, b\}, Follow(T) = \{b, e\}, Follow(Y) = \{c, b, e\}$$

La définition précise de l'automate est laissé au lecteur.

10.1.3 Automate d'analyse prédictive descendante

L'utilisation d'une pile suggère que le programme d'analyse descendante est en fait un automate à pile. On peut construire cet automate de manière systématique dans la mesure où l'on accepte des transitions conditionnées par la lecture du caractère d'avance dans le cas de transitions vides (cela revient à dire que la lecture ne fait pas toujours avancer la tête de lecture sur la bande de l'automate). La justification du fait que l'on peut coder ces automates à pile "augmentés" par de simples automates à pile au prix d'une augmentation du vocabulaire de pile (qui devient formé de couples (a, X) où $a \in V_t$ et $X \in V_P$) est laissée au lecteur. On notera par

$$q \xrightarrow{a,a',X,\alpha} q'$$

une transition de l'automate de l'état q vers l'état q' avec a comme caractère d'avance, $a' \in \{a, \varepsilon\}$ comme symbole lu, X comme symbole de sommet de pile, et enfin α comme mot empilé. Les transitions vides laissent donc le caractère d'avance inchangé, le caractère "lu" dans ce cas étant de la forme (ε, a), alors que les transitions normales consomment le caractère d'avance, le caractère "lu" dans ce cas étant donc de la forme (a, a).

Étant donnée une grammaire $G = (V_t, V_n, S, \mathcal{P})$, on définit la condition d'application d'une règle arbitraire comme suit :

$$\mathcal{P}rediction(N \rightarrow \alpha) = \begin{cases} First(\alpha) \cup Follow(N) & \text{si} \quad \alpha \longrightarrow_G^* \varepsilon \\ First(\alpha) & \text{sinon} \end{cases}$$

en notant que $First(\varepsilon) = \{\}$.
On construit alors l'automate à pile

$$\mathcal{A} = (V_t \cup \{f\}, V_t \cup V_n \cup \{S', f\}, S', \{1, 2\}, 1, \{2\}, T)$$

qui reconnait $\mathcal{L}(G)$ avec deux états, 1 étant initial et 2 final (ce dernier état n'est pas nécessaire si l'on décide de reconnaître par pile vide), où l'ensemble T des transitions est défini comme suit :

À toute règle $M \rightarrow \alpha$ de la grammaire originelle, on associe les transitions :

$1 \xrightarrow{a \in \mathcal{P}rediction(M \rightarrow X\alpha), \varepsilon, M, X\alpha} 1$ encore notées $\quad M \xrightarrow{\mathcal{P}rediction(X\alpha), \varepsilon, M, X\alpha} X$

Aux terminaux $a \in V_t$ et f, on associe les transitions respectives :

$1 \xrightarrow{a,a,\varepsilon} 1$ et $1 \xrightarrow{f,f,\varepsilon} 2$

Le lecteur aura avantage à dessiner l'automate des exemples 10.5 et 10.6. Notons que de nombreuses variantes sont possibles, qui permettent en pratique d'optimiser les constructions données ci-dessus. Notons également que cette construction est très proche de celle donnée pour passer d'une grammaire à un automate, à la différence près du caractère d'avance. Notons enfin que la construction de l'automate de présuppose pas que la grammaire soit éffectivement $LL(1)$.

Il se peut bien sûr que l'automate obtenu soit non-déterministe. Il sera en fait déterministe si la grammaire considérée satisfait les conditions de déterminisme énoncées ci-après.

10.1.4 Condition de déterminisme

On peut maintenant donner formellement les conditions de déterminisme pour l'analyse prédictive descendante d'une grammaire G.

Définition 10.7 *Une grammaire est dite LL(1) si elle satisfait la condition :*
Pour chaque paire de règles $N \rightarrow \alpha \mid \beta$, $\mathcal{P}rediction(N \rightarrow \alpha) \cap \mathcal{P}rediction(N \rightarrow \beta) = \emptyset$.

La condition implique bien sûr qu'une seule règle de membre gauche N peut dériver le mot vide.

10.1.5 Factorisation des membres droits

Il sera parfois possible de transformer la grammaire afin de procéder à une analyse descendante déterministe. Ainsi, la grammaire

$$\begin{array}{rcl} S & \rightarrow & aSaTb \mid aSadT \mid dT \mid c \\ T & \rightarrow & dTK \mid bKc \mid a \\ K & \rightarrow & SS \end{array}$$

ne conduit pas à une analyse déterministe, car $First(aSaTb) = First(aSadT) = \{a\}$. Il est toutefois très simple de la transformer en mettant en facteur la partie commune des deux membres droits concernés, d'où la grammaire :

$$\begin{array}{rcl} S & \rightarrow & aSaL \mid dT \mid c \\ T & \rightarrow & dTK \mid bKc \mid a \\ K & \rightarrow & SS \\ L & \rightarrow & Tb \mid dT \end{array}$$

Un problème identique apparait maintenant avec les règles $L \rightarrow TB \mid dT$ puisque la règle $T \rightarrow dTK$ engendre un d, et donc $First(dT) \cap First(Tb) = \{d\}$. On va éliminer cette nouvelle ambiguité par une nouvelle transformation : on recopie tout d'abord les membres droits de la règle de membre gauche T à la place de T dans Tb, ce qui donne :

$$L \rightarrow dTKb \mid bKcb \mid ab \mid dT$$

puis on factorise à nouveau le membre gauche dT qui est préfixe commun à deux règles, d'où la grammaire finale :

$$\begin{array}{rcl} S & \rightarrow & aSaL \mid dT \mid c \\ T & \rightarrow & dTK \mid bKc \mid a \\ K & \rightarrow & SS \\ L & \rightarrow & dTM \mid bKcb \mid ab \\ M & \rightarrow & Kb \mid \varepsilon \end{array}$$

Le lecteur vérifiera que la même méthode que précédemment permet d'analyser la grammaire obtenue de manière déterministe.

Cette transformation ne permet pas toujours de se ramener à une grammaire pour laquelle une analyse déterministe est possible. Le cas pathologique est celui des règles *récursives gauches*, c'est-à-dire de la forme $S \rightarrow Sv$. Supposons en effet que l'on ait les règles

$$S \quad \rightarrow \quad Sv \mid w$$

où v, w sont des mots non vides dans V^+, n'ayant aucune occurrence de S pour simplifier. Les autres règles de la grammaire G sont quelconques ne contenant pas d'occurrence de S. On aura plusieurs cas suivant que $v, w \longrightarrow_G^* \varepsilon$ ou pas.

1. Si $w \not\longrightarrow^* \varepsilon$, alors $Sv \not\longrightarrow^* \varepsilon$ et donc :
   $\mathcal{P}rediction(S \rightarrow w) = First(w)$
   $\mathcal{P}rediction(S \rightarrow Sv) = First(Sv) = First(w)$
2. Si $w \longrightarrow^* \varepsilon$ et $v \not\longrightarrow^* \varepsilon$, alors $Sv \not\longrightarrow^* \varepsilon$ et donc :
   $\mathcal{P}rediction(S \rightarrow w) = First(w) \cup Follow(S) = First(w) \cup First(v)$
   $\mathcal{P}rediction(S \rightarrow Sv) = First(Sv) = First(S) \cup First(v) = First(w) \cup First(v)$
3. Si $v, w \longrightarrow^* \varepsilon$, alors $Sv \longrightarrow^* \varepsilon$ et donc :
   $\mathcal{P}rediction(S \rightarrow w) = First(w) \cup Follow(S) = First(w) \cup First(v)$
   $\mathcal{P}rediction(S \rightarrow Sv) = First(Sv) \cup FollowS = First(w) \cup First(v) \cup Follow(S) = First(w) \cup First(v)$

Dans tous les cas, l'ensemble des caractères d'avance sont identiques pour les deux règles. L'analyse descendante n'est donc pas adaptée aux grammaires comportant des règles récursives gauches : il faudrait tout d'abord transformer la grammaire afin de les éliminer. Quoiqu'une telle transformation existe, elle est peu utilisée en pratique car on lui préferre dans ce cas une analyse ascendante qui n'a pas le même inconvénient.

10.2 Analyse syntaxique ascendante

L'analyse descendante est très simple à mettre en œuvre, puisqu'il s'agit tout simplement de construire une dérivation gauche. Le fait qu'elle échoue chaque fois que la grammaire est récursive gauche en est la grande faiblesse dans la mesure où de telles règles sont fréquentes en pratique. Considérons par exemple la grammaire désambiguïsée des expressions arithmétiques :

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T \times F \mid F \\ F &\rightarrow (E) \mid \textit{Id} \mid Cte \end{aligned}$$

Les règles $E \rightarrow E + T$ et $T \rightarrow T \times F$ sont récursives gauches. De plus, la technique précédente ne permettrait visiblement pas de choisir entre les règles $E \rightarrow E + T$ et $E \rightarrow T$ qui admettent le même ensemble de caractères d'avance. L'analyse ascendante que nous allons voir maintenant n'a pas cet inconvénient, mais elle est d'une mise en œuvre plus complexe. Le principe est de construire une dérivation droite à l'envers. La construction d'une dérivation droite à l'endroit nécessiterait en effet une lecture de droite à gauche du mot à analyser. La lecture de gauche à droite est donc responsable de cette construction à l'envers de la dérivation. Ce mode opératoire a pour conséquence, on l'a déjà noté, que les objets manipulés au cours de l'analyse ne sont pas des arbres de syntaxe, mais des séquences d'arbres de syntaxe appelées *forêts*. Une forêt a pour *racine* la séquence des racines des arbres qui la composent, c'est-à-dire un mot sur l'alphabet $V_t \cup V_n$ de la grammaire. Par exemple, pour le mot *Id* $\times$ *Id* $+$ Cte, la suite des forêts construites est représentée à la figure 10.3. La suite des racines de ces forêts est la suite de mots *Id*, $F, T, T\times$, $T\times$*Id*, $T\times F, E, E+$, $E+$*Id*, $E+F, E+T, E$.

FIG. 10.3 – Analyse syntaxique ascendante

FIG. 10.4 – Shift et Reduce

10.2.1 Automate non-déterministe d'analyse ascendante

Les opérations effectuées au cours de l'analyse ascendante, comme on peut le remarquer sur la figure 10.3, sont de deux sortes :

(i) la lecture d'un caractère, opération appelée *shift*.

(ii) l'enracinement d'une règle, opération appelée *reduce*. Cette opération consiste à écrire la forêt courante g sous la forme de la juxtaposition de deux forêts ff', puis à construire une nouvelle forêt en juxtaposant la forêt f avec l'arbre de syntaxe de racine N, membre gauche d'une règle $N \rightarrow racine(f')$, et dont f' est la suite des sous-arbres.

Ces opérations sont décrites sur la figure 10.4.

La reconnaissance d'un mot du langage par une suite de shift et reduce va bien-sûr se faire à l'aide d'un automate à pile. Afin d'en faciliter la description, nous allons nous autoriser à lire plusieurs caractères d'un coup dans la pile, de manière à faciliter l'opération *reduce*. On s'autorise également à lire zéro caractères dans la pile, de manière à faciliter l'opération shift. Il aura un unique état, et reconnaitra par pile vide et état final à la fois. Soit $G = (V_t, V_n, S, R)$ la grammaire hors-contexte qui nous intéresse, et qui est cette fois augmentée de la règle $S' \rightarrow dSf$. La raison de cette nouvelle règle initiale est que la pile doit être initialisée par un caractère, toujours le même, qui ne peut donc être le premier caractère du mot à reconnaitre comme cela devrait être dans une analyse ascendante. On force donc ce caractère à être toujours le même, ici d. On fait bien sûr l'hy[pothèse que $d, f \notin V_t$.

$\mathcal{A} = (V_t \cup \{d, f\}, V_n \cup \{S'\}, d, \{1, 2\}, \{2\}, T)$, où la fonctiopn de transition T est définie comme suit :

shift : $1 \xrightarrow{a,\varepsilon,a} 1$ est une transition de l'automate pour tout $a \in V_t$.

reduce : $1 \xrightarrow{\varepsilon,w,N} 1$ est une transition de l'automate pour toute règle $N \rightarrow w \in R$

succès : $1 \xrightarrow{\varepsilon,S',\varepsilon} 2$.

Notons qu'il est facile de transformer cet automate à pile non conforme à la définition en un automate à pile conforme à la définition, c'est-à-dire qui lise exactement un caractère de pile à chaque transition. Pour cela, il suffit d'introduire de nouveaux états dont le rôle est de mémoriser la progression des membres droits de règles sur la pile. Une alternative permettant de mémoriser la progression des membres droits de règles sur la pile est de considérer un nouveau vocabulaire de pile, enrichi des membres droits de règles et de leurs préfixes. Nous nous autoriserons par la suite à utiliser la forme étendue d'automates.

L'automate à pile obtenu est bien sûr non-déterministe puisque shift et reduce sont deux opérations en général non-déterministes. On parle à ce propos de conflit shift-reduce, et de conflit reduce-reduce, car il se peut qu'il y ait plusieurs découpages $f_1 f_1'$ et $f_2 f_2'$ de f tels que les racines de f_1' et f_2' soient des membres droits de règles, de même qu'il peut y avoir plusieurs règles de même membre droit.

Il ne peut être déterministe qu'en l'absence de tels conflits.

L'absence de conflit reduce-reduce impose que 2 membres droits de règles ne puissent coexister en sommet de pile, c'est-à-dire, qu'aucun membre droit de règle ne soit suffixe d'un autre membre droit de règle. Dans le cas de la grammaire des expressions arithmétiques, par exemple, cette condition n'est pas vérifiée à cause des règles comme $E \rightarrow E + T \mid T$. Elle n'est bien sûr jamais vérifiée en présence de règles de la forme $N \rightarrow \varepsilon$.

L'absence de confli shift-reduce est plus gênante puisqu'un shift est toujours possible. Contrairement au cas de l'analyse descendante, l'automate obtenu ne peut donc jamais être déterministe.

10.2.2 Déterminisation de l'analyse ascendante

Nous allons donc le déterminiser grâce à la lecture d'un caractère d'avance. Lorsqu'il est possible de résoudre le non-déterminisme par la lecture d'un caractère d'avance, on dit que la grammaire est LR(1). Le premier L a la même signification que précédemment, il correspond à la lecture de gauche à droite. Initiale du mot anglais Right, R correspond à la dérivation droite (faite à l'envers), et le 1 au nombre de caractères d'avance. Dans notre exemple, nous avons rencontré de nombreux problèmes de choix shift-reduce. Par exemple, après la lecture du premier token « *Id* » du mot à analyser, nous avons effectué une réduction alors que nous aurions pu tout aussi bien faire une nouvelle lecture. Dans ce cas, il n'aurait plus jamais été possible d'obtenir un arbre syntaxique dont le premier caractère lu soit « *Id* ». Par contre, lors de la troisième étape, c'est un shift qui a été préféré à un enracinement : il aurait été possible d'enraciner avec la règle $E \rightarrow T$, mais il n'aurait jamais été possible d'enraciner ultérieurement $E\times$, qui n'apparaît pas dans un membre droit de règle. Des conflits reduce-reduce se présentent également, à chaque fois que la racine de la forêt courante se termine par $E + T$ ou $T \times F$. On peut alors enraciner avec les règles $E \rightarrow T$ et $E \rightarrow E + T$ dans le premier cas, et avec les règles $T \rightarrow F$ et $T \rightarrow T \times F$ dans le second. Dans les deux cas, on préférera la règle de plus long membre droit, c'est à dire $E \rightarrow E + T$ ou $T \rightarrow T \times F$. De nouveau, le choix contraire interdirait l'enracinement ultérieur de $+E$ dans le premier cas et de $\times T$ dans le second. Comme il est possible de résoudre tous les conflits, la grammaire des expressions arithmétiques est LR(1). Notons là-encore qu'une grammaire LR(1) est non-ambiguë.

10.2.3 Principes d'une analyse ascendante prédictive

Dans un premier temps, nous allons réduire le non-déterminisme par une méthode comparable à ce que nous avons fait pour l'analyse ascendante. La décision du choix entre un shift et un reduce, ou entre plusieurs reduce concurrents se fera en comparant le caractère en sommet de pile avec le caractère d'avance.

Dans le cas d'un shift, l'arbre syntaxique doit avoir la forme indiquée à la figure 10.5, ce qui implique l'existence d'une règle $P \rightarrow \beta X \gamma$ telle que $a \in First(\gamma)$. On notera dans ce cas $a \in Follow^0(X)$. Notons que la toute première action lors d'une analyse consiste à lire le premier caractère du mot à analyser, qui est en même temps le caractère d'avance. Il est également important de remarquer que le calcul de $First(\gamma)$ de fait pas intervenir les règles $N \rightarrow \varepsilon$, même si la grammaire en contient. En effet, le membre droit de la règle tout entier doit se trouver sur la pile au moment d'une réduction. Prenons l'exemple de la grammaire de Dick d'ordre 1, avec les règles

$$S \rightarrow \varepsilon \mid S(S)$$

Lorsque la pile contient une parenthèse ouvrante en son sommet -par exemple, la pile vaut $dS($-, alors il ne faut pas faire un shift, puisque cela amènerait une seconde parenthèse ouvrante sur la pile au dessus de la première, ce qui interdirait toute réduction ultérieure. Il faut au contraire faire une réduction avec la règle $S \rightarrow \varepsilon$.

Dans le cas d'un reduce, l'arbre syntaxique doit avoir la forme indiquée à la figure 10.6, ce qui implique la relation habituelle $a \in Follow(N_1)$, ce que nous noterons $a \in Follow^+(X)$ pour ne faire référence qu'au symbole de sommet de pile. Notons que cette notation n'est pas idéale, dans le cas où coexistent plusieurs règles ayant des membres droits se terminant par le même symbole X et des membres gauches distincts. Dans ce cas, on utilisera la version plus précise $a \in Follow(N_1)$.

Notons que les relations $Follow^0(X)$ et $Follow^+(X)$ sont très proches. Ce qui les différencie est que dans le second cas on s'oblige à repasser par le membre gauche N de la règle $N \rightarrow \beta X$ candidate pour une réduction (et peut-être également par d'autres membres gauches succesivement), alors que dans le premier on s'interdit ce passage par un tel membre gauche de règle.

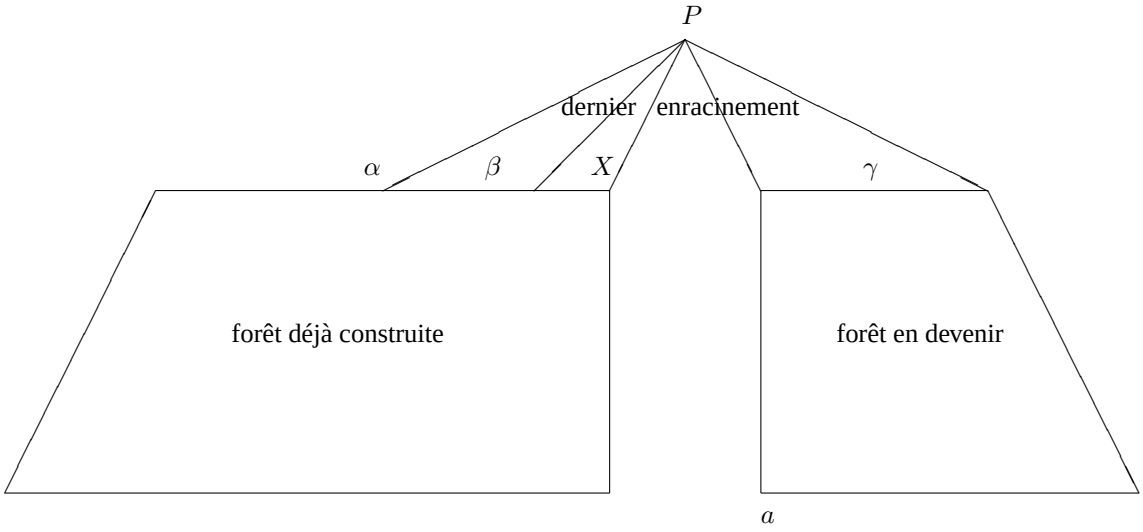


FIG. 10.5 – Réussite d'un shift

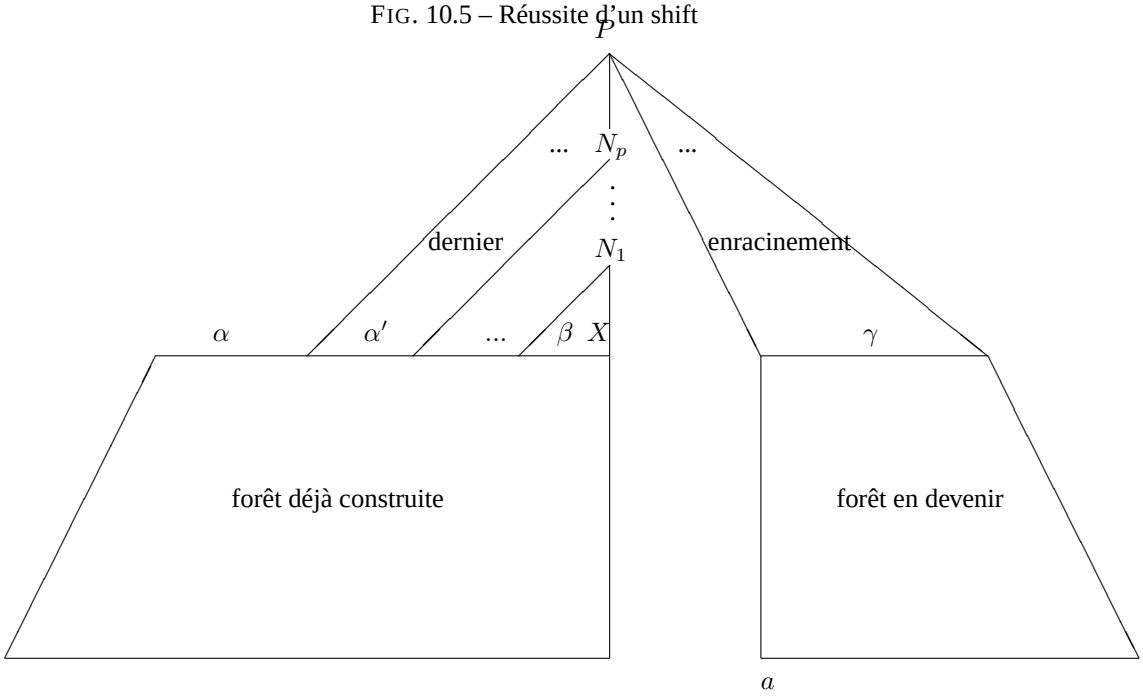


FIG. 10.6 – Réussite d'un reduce

Plutôt que de donner l'automate d'analyse ascendante prédictive pour la grammaire des expressions arithmétiques sous sa forme habituelle, nous allons donner le programme d'analyse ascendante déterministe qui code cet automate :

```
Proc Analyse-Ascendante-Deterministe(u)
 u:= u·f ; r := d ;
 Tantque r = r'X et u=av Faire
 Cas X = d Alors Empiler(a) ; u := v
     X = E Alors Si a ∈ {+,),f} Alors Empiler(a) ; u := v
                 Sinon Retourner(''echec'')
                 Fin Si
     X = T Alors Cas a = * Alors Empiler(a) ; u := v
                     a ∈ {+,),f} Alors Cas r = r'E+T Alors r := r'E
```

```
                                          r = r'T Alors r := r'E
                                       Fin Cas
                  Sinon Retourner(''echec'')
                  Fin Cas
      X = F Alors Si a ∈ {+,*,),f} Alors Cas r = r'T*F Alors r := r'T
                                           r = r'F Alors r := r'T
                                        Fin Cas
                  Sinon Retourner(''echec'')
                  Fin Si
      X = ( Alors Si a ∈ {(,Id,Cte} Alors Empiler(a) ; u:=v
                  Sinon Retourner(''echec'')
                  Fin Si
      X = * Alors Si a ∈ {(,Id,Cte} Alors Empiler(a) ; u:=v
                  Sinon Retourner(''echec'')
                  Fin Si
      X = + Alors Si a ∈ {(,Id,Cte} Alors Empiler(a) ; u:=v
                  Sinon Retourner(''echec'')
                  Fin Si
      X = ) Alors Si a ∈ {+,*,),f} Alors Si r = r'(E) Alors r := r'F
                                                     Sinon Retourner(''echec'')
                                                     Fin Si
                  Sinon Retourner(''echec'')
                  Fin Si
      X = Id Alors Si a ∈ {+,*,),f} Alors r := r'F Fin Si
                   Sinon Retourner(''echec'')
                   Fin Si
      X = Cte Alors Cas a ∈ {+,*,),f} Alors r := r'F Fin Si
                    Sinon Retourner(''echec'')
                    Fin Si
    Sinon Retourner(''echec'')
    Fin Cas
    Fin Tantque
    Si r=dEf Alors Retourner(''succes'') Sinon Retourner(''echec'')
  Fin Proc
```

Conditions de déterminisme Elles expriment l'impossibilité d'un conflit shift-reduce (condition (i)) comme d'un conflit reduce-reduce (conditions (ii) et (iii)) pour la grammaire G à analyser :

(i) Si la pile est de la forme $r = \alpha\beta X$ où $N \to \beta X$ est une règle de G, alors l'absence de conflit shift-reduce est exprimé par la condition :

$$Follow^0(X) \cap Follow(N) = \emptyset$$

(ii) Si $N \to \beta$ et $M \to \alpha\beta$ sont deux règles de G avec $M \neq N$, alors l'absence de conflit reduce-reduce est exprimé par la condition :

$$Follow(N) \cap Follow(M) = \emptyset$$

(iii) Si $N \to \beta$ et $N \to \alpha K\beta$ sont deux règles de G, alors une condition suffisante d'absence de conflit reduce-reduce est exprimé par la condition suivante, qui interdit d'empiler N au dessus de K :

$$N \notin Follow^0(K)$$

On pourrait choisir une condition plus générale, mais celle-ci nous suffira dans un premier temps.

On peut vérifier que les conditions de déterminisme sont satisfaites pour la grammaire G des expressions arithmétiques désambiguées.

Notons que cette méthode d'analyse revient à construire un automate à pile possédant un unique état, les transitions étant entièrement déterminées par le symbole en sommet de pile et le caractère d'avance.

10.2.4 Automate SLR d'analyse ascendante

L'analyse que nous venons d'éffectuer a de nombreuses limitations, car la connaissance du sommet de pile n'est pas suffisante pour en caractériser l'état. Elle ne permet donc pas d'anticiper certains échecs, et aboutit à des conflits shift/reduce dans des cas où un shift (ou un reduce) ne peut manifestement pas conduire à une analyse réussie. Considérons l'exemple suivant d'un fragment de grammaire pour lequel nous avons déjà éffectué l'analyse descendante prédictive :

$$\begin{array}{rcl} S' & \rightarrow & dSf \\ S & \rightarrow & aSa \mid bT \mid c \\ K & \rightarrow & SS \end{array}$$

Supposons que S est en sommet de pile. Nous ferons

- un shift si $a \in Follow^0(S) = \{a, b, c, f\}$;
- un reduce avec la règle $S \rightarrow SS$ si $a \in Follow^+(S) = \{c\}$.

Le non-déterminisme n'a donc pas été résolu. Il pourrait toutefois l'être grâce à une analyse un tout petit peu plus précise. En effet, supposons que le caractère d'avance soit c et que le sommet de pile contienne :

1. bS : dans ce cas, on ne peut réduire avec SS qui ne figure pas dans la pile. Il faut faire un shift, la pile contient alors bSc, ce qui permet de réduire avec $S \rightarrow c$, puis avec $k \rightarrow SS$ et la pile contiendra alors bK ce qui permet de poursuivre l'analyse, par exemple si le nouveau caratère d'avance est c.
2. bSS : dans ce cas, un shift amènerait $bSSc$ en sommet de pile, puis $bSSS$, puis bSK, ce qui conduit nécessairement à un échec. Au contraire, un reduce donne bK en sommet de pile, puis bKc par shift, ce qui amène la nouvelle réduction avec $t \rightarrow bKc$, permettant de poursuivre l'analyse.

Nous allons donc raffiner notre analyse en mémorisant plus d'information sur l'état de la pile. Pour cela, nous pouvons nous appuyer sur une propriété fondamentale de l'analyse ascendante : le langage des mots de pile obtenus au cours des analyses (non-déterministes) réussies des mots de $\mathcal{L}(G)$ est reconnaissable.

L'idée est de construire un automate dont les états décrivent la progression dans l'analyse en cours des membres droits de règles possibles. Ces états vont être constitués d'un ensemble d'*items*, où un item est une règle dont le membre droit est décoré par une marque adéquate, on utilise généralement un point. Par exemple, $E \rightarrow .E + T, E \rightarrow E. + T, E \rightarrow E + .T, E \rightarrow E + T.$ sont des items de la grammaire désambiguée des expressions arithmétiques qui décrivent les étapes successives de l'analyse de la règle $E \rightarrow E + T$. Un item seul ne suffit pas à décrire tous les calculs possibles à un moment donné, car les membres droits de deux règles différentes peuvent avoir un préfixe commun, ne serait-ce que le préfixe vide. Tant que ce préfixe commun est en cours d'analyse, on ne peut savoir laquelle des deux règles s'appliquera in fine. Les états seront donc des ensembles d'items représentant les futurs possibles du calcul en cours. Par exemple, $E \rightarrow E + .T, T \rightarrow .T * F, T \rightarrow .F, F \rightarrow .(E), F \rightarrow .Id, F \rightarrow .Cte$ constitue un ensemble d'items qui décrivent les évolutions possibles de l'analyse lorsque la pile contient $E+$ en son sommet. Notons que cet ensemble d'items contient des informations très précieuses :

- Le sommet de pile $E+$;
- Les caractères qui appartiennent à $First(T)$ tout comme ceux qui appartiennent à $First(F)$. Cela est du au fait que T et F figurent à droite du point dans l'un au moins des items de l'ensemble. Il s'ensuit que le calcul de la relation $Follow$ sera inutile.

Définition 10.8 *Étant donnée une grammaire G dont $N \to \alpha\beta$ est une règle, la règle pointée $N \to \alpha.\beta$ est appellée* item *ou* en-cours. *Un ensemble I d'item est*

- clos *si pour tout item $M \to \beta.N\gamma \in I$, alors $N \to .\alpha \in I$ pour toute règle $N \to \alpha$ de G ;*
- compatible *si tous les items qu'il contient ont des préfixes compatibles : étant donnés deux items quelconques $N \to \alpha.\beta$ et $N' \to \alpha'.\beta'$ appartenant à l'ensemble, α' est suffixe de α (ou le contraire) ;*
- terminal *s'il ne contient que des items* terminaux *de la forme $N \to \alpha.$, pour $N \to \alpha \in G$.*

La clôture d'un ensemble d'items compatible I est le plus petit ensemble d'items clos contenant I.

Intuitivement, un item $N \to \alpha.\beta$ indique donc que le sommet de pile contient le mot α, et que le calcul peut se continuer par β. Cela explique la notion de compatibilité (cohérence de la description du sommet de pile) et de cloture (présence de tous les futurs possibles).

Définition 10.9 *Étant donnée une grammaire $G = (V_t, V_n, S, \mathcal{P})$ augmentée par la règle $S' \to dSf$, on définit l'automate fini $\mathcal{A} = (V_t \cup V_n \cup \{d, f\}, Q, i, Q, T)$ dont les états sont les clôtures des ensembles d'items compatibles, l'état initial est la clôture de l'ensemble $\{S' \to d.Sf\}$, tous les états sont finaux, et les transitions sont définies comme suit :*

$$T(I, a \in V_t \cup \{f\}, X \in V_t \cup V_n \cup \{d\}) = I'$$

où I' est la clôture de l'ensemble $\{N \to \alpha X.\beta \mid N \to \alpha.X\beta \in I\}$.

Remarquons que l'ensemble d'items initial est compatible et que la compatibilité des ensemble d'items est conservée par la définition de la fonction de transition, ce qui assure que I' est bien un état de l' automate.

Lemme 10.10 *L'automate A reconnait le langage des mots de pile obtenus au cours des analyses non-déterministes (réussies) des mots de $\mathcal{L}(G)$.*

Calculons les ensembles d'items pour la grammaire des expressions arithmétiques :
$I0 = \{S' \to .Ef, E \to .E + T, E \to .T, T \to .T * F, T \to .F, F \to .(E), F \to .Id, F \to .Cte\}$
$I1 = \{S' \to E.f, E \to E. + T\}$
$I2 = \{E \to T., T \to T. * F\}$
$I3 = \{T \to F.\}$
$I4 = \{F \to (.E), E \to .E + T, E \to .T, T \to .T * F, , T \to .F, F \to .(E), F \to .Id, F \to .Cte\}$
$I5 = \{F \to Id.\}$
$I6 = \{F \to Cte.\}$
$I7 = \{S' \to Ef.\}$
$I8 = \{E \to E + .T, T \to .T * F, T \to .F, F \to .(E), F \to .Id, F \to .Cte\}$
$I9 = \{T \to T * .F, F \to .(E), F \to .Id, F \to .Cte\}$
$I10 = \{F \to (E.), E \to E. + T\}$
$I11 = \{E \to E + T., T \to T. * F\}$
$I12 = \{T \to T * F.\}$
$I13 = \{F \to (E).\}$

L'automate obtenu dans le cas des expressions arithmétiques est donné à la figure 10.7, où il est représenté sous forme d'une table dont les lignes sont indexées par les noms d'états, et les colonnes par les lettres du vocabulaire.

Notons qu'il n'y a pas de transitions sortante depuis les états constitués d'items tous terminaux, ce qui est parfaitement normal, puisque tous les items étant terminaux dans un tel état, continuer l'analyse est devenu impossible.

On peut transformer cet automate en un automate à pile reconnaissant les mots de $\mathcal{L}(G)$:

1. Les états de l'automate restent les même. Toutefois, l'état I7 correspondant à la fin d'une analyse réussie sera éliminé ;

État	Shift/Reduce							Transitions vides		
	f	$+$	$*$	$($	$)$	Id	Cte	E	T	F
$I0$				$I4$		$I5$	$I6$	$I1$	$I2$	$I3$
$I1$	$I7$	$I8$								
$I2$			$I9$							
$I3$										
$I4$				$I4$		$I5$	$I6$	$I10$	$I2$	$I3$
$I5$										
$I6$										
$I7$										
$I8$				$I4$		$I5$	$I6$		$I11$	
$I9$				$I4$		$I5$	$I6$		$I12$	
$I10$		$I8$			$I13$					
$I11$			$I12$							
$I12$										
$I13$										

FIG. 10.7 – Automate d'analyse des mots de pile engendrés au cours des analyses ascendantes réussies des expressions arithmétiques

2. Les transitions étiquettées par une lettre de V_t sont des shifts, ils provoquent la lecture d'un caractère du mot à analyser. On notera $S - Ik$ pour indiquer que le shift (c'est-à-dire l'empilement du caractère lu) est suivi d'une transition dans l'état Ik.
3. Les transitions étiquettées par une lettre $X \in V_n$ correspondent à un changement d'état induit par l'empilement d'un membre gauche de règle suite à une réduction ;
4. Les états contenant des items de la forme $N \rightarrow \alpha.$ deviennent source d'une transition au cours de laquelle α est dépilé et N est empilé. Cette réduction a bien sûr lieu au cas où le caractère d'avance appartient à $Follow(N)$. L'état d'arrivé doit donc être calculé en fonction de la réduction effectuée et de l'état de la pile. Par exemple, si la pile contient le mot $dE + T$ et qu'il faut faire la réduction $E \rightarrow E + T$, alors elle contiendra ensuite le mot dE et l'on devra donc être dans l'état $I1$. Mais si la pile contenait le mot $d(E + T$, la même réduction aboutira au mot de pile $d(E$, et l'on devra être dans l'état $I10$. Chaque configuration de la pile correspondant à un étatd e l' automate, on peut stocker dans la pile (en plus du caractère de pile) létat courant atteint au moment de cet empilement. Une réduction consistera alors à dépiler un nombre de fois égal à la taille du membre droit de le règle utlisée dans la réduction, puis à empiler le membre gauche ainsi que le nouvel état atteint (obtenu en lisant la table de transition de l'automate d'analyse des mots de pile, l'état de départ étant stocké dans la pile, et le caractère lu étant le membre gauche de la règle utilisée).
5. On considérera que la pile est initialisée avec le symbole d de manière à ne pas être vide, mais en fait ce symbole ne jouera aucun rôle. La figure 10.8 exprime cet automate en indiquant, lorsqu'une réduction doit être éffectuée, quelle règle s'applique, les règles étant numérotées à partir de 0, numéro de la règle $S \rightarrow dSf$.

On en déduit que les états contenant un item terminal et des items non-terminaux sont sources de conflits shift-reduce potentiels. En cas d'absence de tel état, on est assuré de pas avoir de tel conflit.

De même, les états contenant plusieurs items terminaux sont sources de conflits reduce-reduce potentiels. En cas d'absence de tel état, on est assuré de pas avoir de tel conflit.

Le cas de la grammaire des expressions arithmétiques traité à la figure 10.8 montre un exemple d'automate déterministe qui possède des états pouvant engendrer les deux catégories de conflit. Le caractère d'avance permet toutefois dans cet exemple de résoudre les conflits potentiels.

État	Shift/Reduce							Transitions vides		
	f	$+$	$*$	$($	$)$	Id	Cte	E	T	F
$I0$				$S-I4$		$S-I5$	$S-I6$	$I1$	$I2$	$I3$
$I1$	succès	$S-I8$								
$I2$	$R2$	$R2$	$S-I9$		$R2$					
$I3$	$R4$	$R4$	$R4$		$R4$					
$I4$				$S-I4$		$S-I5$	$S-I6$	$I10$	$I2$	$I3$
$I5$	$R6$	$R6$	$R6$		$R6$					
$I6$	$R7$	$R7$	$R7$		$R7$					
$I8$				$S-I4$		$S-I5$	$S-I6$		$I11$	
$I9$				$S-I4$		$S-I5$	$S-I6$		$I12$	
$I10$		$S-I8$			$S-I13$					
$I11$	$R1$	$R1$	$S-I12$	$R1$						
$I12$	$R3$	$R3$	$R3$	$R3$						
$I13$	$R5$	$R5$	$R5$	$R5$						

FIG. 10.8 – Automate d'analyse ascendante des expressions arithmétiques

Pile	Input	État	Action
d	Id * Id + Id f	I0	S-I5
d Id	* Id + Id f	I5	R-6
d F	* Id + Id f	I3	R-4
d T	* Id + Id f	I2	S-I9
d T *	Id + Id f	I9	S-I5
d T * Id	+ Id f	I5	R-6
d T * F	+ Id f	I12	R-3
d T	+ Id f	I2	R-2
d E	+ Id f	I1	S-I8
d E +	Id f	I8	S-I5
d E + Id	f	I5	R6
d E + F	f	I3	R4
d E + T	f	I11	R1
d E	f	I1	succès

FIG. 10.9 – Exemple d'analyse ascendante déterministe

Cet automate est appellé automate SLR de la grammaire des expressions arithmétiques. Un exemple d'analyse avec l'automate SLR est donné à la figure 10.9.

Le lecteur pourra utilement modifier les définitions précédentes de manière à effectivement stocker les états dans la pile de manière à pouvoir calculer automatiquement l'état atteint en cas de réduction.

Variantes On peut améliorer la méthode SLR, en enrichissant les items SLR (aussi appellés items LR(0)) comme suit :

Définition 10.11 *On appellera* item LR(1) *les couples* $(N \to \alpha.\beta, a)$ *formés d'un item SLR et d'un caractère du vocabulaire terminal tels que* $a \in First(\beta)a$ *appellé* caractère de retour *de l'item. Un ensemble I d'items LR(1) sera* clos *si pour tout item* $(N \to \alpha.X\beta, b) \in I$*, pour toute règle* $X \to \gamma$

et tout terminal $a \in First(\beta b)$, alors $(X \rightarrow .\gamma, a) \in I$. L'ensemble initial d'items est la clôture de l'ensemble $S' \rightarrow i.Sf, f$. Dans cette définition, le caractère de retour a deviendra le caractère d'avance lors de la réduction $X \rightarrow \gamma$, et donc, les caractères de retour sont conservés lors d'une transition de l'automate. Une réduction $N \rightarrow \gamma$ ne pourra être éffectuée dans un état I en présence du caractère d'avance a, qu'à la condition que I contienne l'item terminal $(N \rightarrow \gamma., a)$.

Une ultime variante, appellée LALR(1), a pour objectif de réduire le nombre d'états en regroupant les états LR(1) similaires, c'est-à-dire qui contiennent des items identiques, au caractère d'avance près. Cette méthode risque de faire apparaître des conflits inexistants dans la méthode LR, mais elle est en pratique celle qui est implémentée dans les générateurs d'analyseurs.

Une dernière variation de la méthode consiste à ne pas désambiguer la grammaire, mais à donner les règles de précédences ou d'associativité permettant de mener cette désambiguation à bien automatiquement lors du calcul de l'automate LR(1) ou LALR (sans transformer la grammaire à priori). Les générateurs d'analyseurs d'aujourd'hui permettent ce type de désambiguation.

10.2.5 Génération d'analyseurs syntaxiques avec YACC

On dispose aujourd'hui d'outils qui permettent la génération d'analyseurs syntaxiques à partir de grammaires hors-contextes, dont le plus connu est YACC. Étant donnée une grammaire, YACC essaye de trouver une stratégie pour une analyse ascendante en utilisant la méthode LALR (1) décrite dans la section précédente. En fait, YACC accepte comme données des grammaires qui peuvent être ambiguës (donc ne sont exactement pas LALR), pourvu qu'elles soient accompagnées des *règles de désambiguïsation* adéquates qui précisent les priorités respectives des différents opérateurs. Cette facilité permet de garder des grammaires simples, mais son utilisation nécessite une bonne compréhension du fonctionnement du logiciel.

Dans la section précédente, nous avons décrit le principe de l'analyse ascendante comme une construction explicite de l'arbre de dérivation. Pour la réalisation d'un tel algorithme il n'est pas du tout nécessaire de garder toutes les forêts construites : comme toutes les actions de l'analyseur dépendent seulement du prochain caractère lu et des *racines* des forêts construites, YACC va économiser l'utilisation de la mémoire et stocker seulement les racines des forêts. Autrement dit, l'analyseur engendré par YACC par défaut fournit seulement une fonction disant si un mot donné est dans le langage engendré par la grammaire ou pas. Cependant, l'analyse syntaxique n'est en principe qu'un premier pas dans un logiciel plus complexe, et on a donc généralement besoin d'obtenir un résultat plus informatif. Dans ce but, YACC permet d'associer aux règles de la grammaire des expressions permettant de construire le résultat souhaité de l'analyse syntaxique, arbre de syntaxe, résultat d'évaluation, etc. Ces expressions prennent des valeurs dans une domaine défini dans le langage de programmation, et utilisent des opération prédéfinies (ou définies par le programmeur) dans ce langage. L'exemple ci-dessous décrit une grammaire des expressions arithmétiques, ainsi que l'évaluation des expressions reconnues (notre syntaxe dérive de la syntaxe réelle de YACC, à trouver dans un manuel de référence) :

Non-terminaux : E, T
Axiome : E
Tokens : Cte avec valeur de type entier, $+$, $(,)$

$$\begin{array}{lcll} E & \rightarrow & E\ +\ T & \{E+T\} \\ & | & T & \{T\} \\ T & \rightarrow & (E) & \{E\} \\ & | & \text{Cte} & \{\text{val(Cte)}\} \end{array}$$

On fournit donc à YACC les informations concernant la grammaire d'une part, c'est-à-dire la liste des non-terminaux, l'axiome, la liste des tokens, celle des règles, et d'autre part un mécanisme d'évaluation donné sous la forme d'expressions accollées aux règles. L'expression $\{E+T\}$ accollée à la règle $E \rightarrow E + T$ spécifie un calcul à effectuer sur l'arbre de syntaxe. La règle $E \rightarrow E + T$

signifie que l'arbre de syntaxe de racine E est formé d'un arbre de racine E (le E du membre gauche de règle), dont les trois sous-arbres ont pour racines respectives E (le E du membre droit de règle), $+$ et T. Notons par A_1 et A_2 les deux sous-arbres de syntaxe dont les racines E et T sont des non-terminaux. Supposons (c'est notre hypothèse de récurrence) calculées les valeurs v_1 et v_2 des sous-arbres A_1 et A_2. Le calcul de la valeur de l'arbre tout entier s'exprime comme une fonction, qui ne dépend que de la règle $E \to E + T$, dont les arguments sont v_1 et v_2. Dans le cas présent, ce sera la fonction qui calcule la somme $v_1 + v_2$. Quand il y a plusieurs occurrences du même token ou non-terminal dans le membre droit d'une règle de la grammaire, on doit les numéroter pour les distinguer, comme dans l'exemple suivant, qui calcule le nombre de paires de parenthèses dans un mot de la grammaire de Dick à une parenthèse :

Non-terminaux : S
Axiome : S
Tokens : $($, $)$

$$\begin{array}{rcll} S & \to & \varepsilon & \{0\} \\ & | & (S)S & \{S_1 + S_2 + 1\} \end{array}$$

Ce mécanisme est utilisé de manière systématique pour calculer une représentation compacte de l'arbre de dérivation, comme expliqué dans le chapitre 11.

YACC est disponible dans un grand nombre d'implantations de langages de programmation, que ce soit C (la version originelle de YACC), OCaml ou JAVA. Chaque implantation possède ses particularités. En particulier, le langage d'expressions utilisé par YACC dans ces implantations est le langage de programmation hôte.

10.3 Exercices

Exercice 10.1 *On considère le langage des expressions arithmétiques avec addition, soustraction, opposé, multiplication, division, et exponentiation (notée $\uparrow$) :*

$$\begin{array}{rcl} E & \to & E + E \mid E - E \mid -E \mid E \times E \mid E/E \mid E \uparrow E \mid (E) \mid Int \\ Int & \to & [0-9]^+ \end{array}$$

Cette grammaire étant ambiguë, on adopte les règles de désambiguïsation suivantes : les opérateurs binaires associent à gauche sauf l'exponentiation qui associe à droite, et l'ordre de priorité décroissant est l'exponentiation, puis l'opposé, puis la multiplication et la division, puis l'addition et la soustraction.

1. *Donner le parenthésage implicite correct du mot $1 - 3 \times 4 + -5 \uparrow 2$.*
2. *Définir une grammaire non-ambiguë décrivant ce langage, respectant les règles de désambiguïsation. On utilisera un nouveau non-terminal par niveau de priorité.*
3. *Associer à chaque règle de cette grammaire un calcul permettant de déterminer le nombre d'opérateurs de l'expression reconnue.*
4. *Associer à chaque règle de la grammaire non ambiguë un calcul permettant d'évaluer l'expression reconnue.*
5. *Donner un algorithme d'analyse ascendante non-déterministe de cette grammaire.*
6. *Le déterminiser.*

Exercice 10.2 *On se pose le problème d'afficher un texte sur un terminal dont le nombre de colonnes est fixé C, de manière à ce qu'aucun mot ne soit à cheval sur deux lignes. On veut aussi « compresser » le texte en supprimant les occurrences consécutives des séparateurs (caractères blancs et passages à la ligne, notés respectivement _ et / ci-dessous). Ainsi un texte source peut être :*

`qui___veut__voyager_loin__/_menage_sa__monture`

La sortie d'un tel texte sur 20 colonnes doit être

`qui_veut_voyager/loin_menage_sa/monture`

Pour cela, on commence par donner une grammaire pour le texte source :

$$\begin{array}{rcl} Phrase & \rightarrow & Mot \mid Phrase\ Sep\ Phrase \\ Mot & \rightarrow & Car \mid Mot\ Mot \\ Car & \rightarrow & \texttt{a} \mid \cdots \mid \texttt{z} \\ Sep & \rightarrow & \texttt{\_} \mid \texttt{/} \mid Sep\ Sep \end{array}$$

1. *Montrer que cette grammaire est ambiguë.*
2. *Définir une grammaire non-ambiguë engendrant le même langage.*
3. *Faire l'analyse ascendante non-déterministe puis déterministe de cette grammaire.*

Exercice 10.3 *On veut piloter un traceur graphique au moyen de mots sur l'alphabet* $\{\uparrow, \downarrow, \rightarrow, \leftarrow\}$. *Le langage accepté est donné par la grammaire suivante :*

$$L \rightarrow LL \mid \uparrow \mid \downarrow \mid \rightarrow \mid \leftarrow \mid \varepsilon$$

Le traceur est situé initialement en position $(0, 0)$. *Chaque mouvement le déplace de 1mm.*

1. *Donnez une grammaire non ambiguë pour ce langage dont les règles soient récursives droites.*
2. *Déterminer les calculs effectués lors de l'analyse ascendante gauche du mot* $\uparrow\rightarrow\uparrow\leftarrow$. *On dessinera au préalable l'arbre de dérivation et l'on effectuera les calculs en remontant dans cet arbre, puis on montrera comment ces calculs sont effectués par l'analyse ascendante en écrivant à chaque étape les racines des arbres de dérivation intermédiaires.*
3. *Faire l'analyse ascendante de cette grammaire, d'abord non-déterministe, puis déterministe si possible.*
4. *Donnez une grammaire non ambiguë pour ce langage dont les règles soient récursives gauches.*
5. *Déterminer les calculs effectués lors de l'analyse ascendante gauche du mot* $\uparrow\rightarrow\uparrow\leftarrow$. *On dessinera au préalable l'arbre de dérivation et l'on effectuera les calculs en remontant dans cet arbre, puis on montrera comment ces calculs sont effectués par l'analyse ascendante en écrivant à chaque étape les racines des arbres de dérivation intermédiaires.*
6. *Faire l'analyse ascendante de cette grammaire, d'abord non-déterministe, puis déterministe si possible.*
7. *Donnez les expressions associées à cette grammaire pour calculer la position du pointeur en fin de lecture du mot. Ces expressions seront de type enregistrement* $\{x : int; y : int\}$.
8. *Déterminer les calculs effectués lors de l'analyse ascendante gauche du mot* $\uparrow\rightarrow\uparrow\leftarrow$. *On dessinera au préalable l'arbre de dérivation et l'on effectuera les calculs en remontant dans cet arbre, puis on montrera comment ces calculs sont effectués par l'analyse ascendante en écrivant à chaque étape les racines des arbres de dérivation intermédiaires.*

Exercice 10.4 *Les deux grammaires suivantes engendrent le langage* L_s *des séquences d'identificateurs :*

$$\begin{array}{rcl} S & \rightarrow & \varepsilon \mid Id\ S \\ S & \rightarrow & \varepsilon \mid S\ Id \end{array}$$

1. *Laquelle de ces deux grammaires permet-elle une analyse descendante du langage L_s ?*
2. *Supposons qu'on veuille réaliser une analyse ascendante pour le langage L_s avec un outil comme YACC. Laquelle des deux grammaires est préférable si on veut économiser la consommation de mémoire ?*

Exercice 10.5 *Dans le fichier YACC des expressions arithmétiques, ajouter des expressions pour calculer :*

1. *le nombre d'opérateurs ;*
2. *l'ensemble des identificateurs utilisés ;*
3. *l'arbre de dérivation.*

Dans chaque cas, spécifier d'abord les opérations auxiliaires nécessaires.

Chapitre 11

Syntaxe abstraite des langages

L'objectif de ce chapitre est de définir une structure de données permettant de représenter le résultat de l'analyse syntaxique sous une forme adaptée à l'analyse sémantique.

Le type et la signification d'une phrase d'un langage sont calculés à partir de son arbre de syntaxe. Mais la syntaxe (dite *concrète*) d'un langage est souvent encombrée de complications nécessaires à l'analyse syntaxique (visant en particulier à traiter les problèmes d'ambiguïté), mais nuisibles à la compréhension, donc à la définition de sa sémantique. Cela rend nécessaire la définition d'un niveau intermédiaire, celui de la *syntaxe abstraite*, dont le but est d'éliminer de l'arbre de dérivation les détails devenus inutiles à ce stade. Plus précisément, la construction d'un *arbre de syntaxe abstraite* a trois objectifs :

- éviter les ambiguïtés inhérentes aux grammaires en construisant une structure nécessairement non-ambiguë ;
- éliminer les complications inutiles introduites dans les arbres de dérivations par l'abondance de mots clés et de symboles de ponctuation ;
- réunir en une seule notion les arbres de dérivation décrivant la structure d'une phrase du langage et les valeurs associées aux tokens de cette phrase.

Dans le cas des expressions arithmétiques, les expressions de syntaxe abstraite que l'on souhaite obtenir sont de la forme $+(\text{Cte}, \times(\text{Cte}, \text{Cte}))$. Ces expressions se prêtent en effet particulièrement bien à une évaluation, car elles correspondent à des arbres dont les nœuds sont étiquetés par des opération arithmétiques. Une évaluation revient donc à propager les calculs d'opérations arithmétiques dans l'arbre.

11.1 Grammaires abstraites

Définition 11.1 *Soit T un ensemble (fini ou infini) appelé ensemble des* étiquettes abstraites *et soit V_t le vocabulaire formé par T augmenté des parenthèses et de la virgule. Une grammaire G sur le vocabulaire terminal V_t est dite* abstraite *si*

1. *toutes les règles sont de la forme $N \to t$ ou bien $N \to t(N_1, \ldots, N_n)$ ($n \geq 1$) où les N_i sont des non-terminaux et $t \in T$;*
2. *une étiquette t donnée n'apparaît qu'une seule fois dans l'ensemble des règles.*

Un mot engendré par une grammaire abstraite s'appelle une expression de syntaxe abstraite *ou, plus simplement,* expression abstraite.

Par exemple, si $T = \{+, \times, \text{Cte}\}$ alors la grammaire

$$E \to +(E, E) \mid \times(E, E) \mid \text{Cte}$$

sera notre grammaire abstraite des expressions arithmétiques formées d'additions, de multiplications et de constantes.

Grâce à la propriété qu'une étiquette t donnée n'apparaît qu'une seule fois dans l'ensemble des règles, on a la propriété fondamentale suivante :

Proposition 11.2 *Une grammaire abstraite est non-ambiguë.*

Notre premier objectif est donc atteint.

11.2 Arbres de syntaxe abstraite

Nous allons maintenant donner une représentation arborescente des expressions abstraites. Cela est possible grâce à la forme $N \to t(N_1, \ldots, N_k)$ des règles d'une grammaire abstraite. Une expression engendrée par le non-terminal N de cette règle sera représentée par l'arbre

$$\begin{array}{ccccc} & & t & & \\ & & & & \\ a_1 & & \ldots & & a_k \end{array}$$

où a_i est, récursivement, l'arbre associé à l'expression abstraite e_i engendrée par le non-terminal N_i. On dira alors que cet arbre est de *sorte* N. D'où la définition :

Définition 11.3 *Étant donnée une grammaire abstraite $G = (V_t, V_n, S, R)$, un* arbre de syntaxe abstraite *de* sorte $N \in V_n$ *(ou plus simplement* arbre abstrait*) est un arbre A dont les nœuds sont étiquetés par des symboles $t \in T$, tel que :*

- *soit il existe une règle $N \to t \in R$ et A est réduit à une feuille étiquetée par t ;*
- *soit il existe une règle $N \to t(N_1, \ldots N_n)$ telle que*
  - *(i) la racine de A soit étiquetée par t ;*
  - *(ii) A ait exactement n fils qui soient, récursivement, des arbres de syntaxe abstraite de sortes respectives N_1,…,N_n.*

Pour la grammaire abstraite précédente, on a par exemple les arbres de syntaxe abstraite suivants :

$$\begin{array}{ccc} \times & & \\ & & \\ \text{Cte} \quad \text{Cte} & & \end{array} \qquad\qquad \begin{array}{ccc} + & & \\ & & \\ \text{Cte} \quad \times & & \\ & & \\ \text{Cte} \quad \text{Cte} & & \end{array}$$

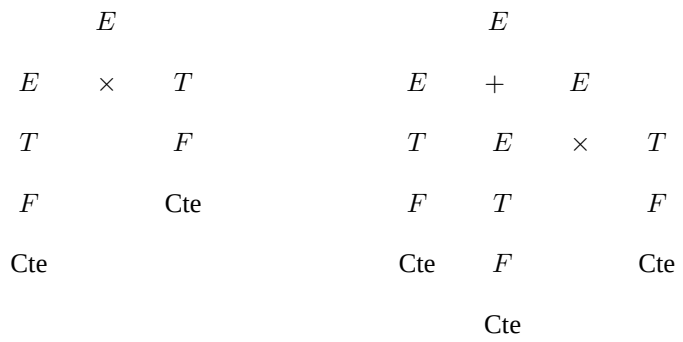
Il existe une correspondance biunivoque entre ces arbres et les mots reconnu par la grammaire abstraite : le parcours gauche d'un arbre de syntaxe abstraite permet de construire le mot correspondant. Ainsi, les exemples d'arbres précédents correspondent aux mots

$$\times(\text{Cte}, \text{Cte}) \qquad\qquad +(\text{Cte}, \times(\text{Cte}, \text{Cte}))$$

Réciproquement, on associe à chaque mot engendré par la grammaire abstraite un arbre de syntaxe abstraite par une construction analogue à la construction récursive élaborée au paragraphe 7.6 :

- à un mot engendré par une règle $N \to t$ on associe l'arbre réduit à une feuille étiquetée par t ;
- à un mot engendré par une règle $N \to t(N_1, \ldots, N_n)$ on associe l'arbre dont la racine est étiquetée par t et dont les fils sont, dans cet ordre, les arbres associés aux mots engendrés par N_1,…, N_n.

Si l'on compare les arbres de syntaxe abstraite des mots exemples précédents avec leurs arbres de dérivation dans la grammaire concrète non-ambiguë du langage des expressions arithmétiques :



on constate que l'on a atteint notre deuxième objectif : les non-terminaux introduits pour désambiguïser ont été éliminés. Remarquons aussi que si l'expression originale contenait des parenthèses, celles-ci ne sont plus présentes dans l'arbre : c'est la structure de l'arbre qui donne le parenthésage.

11.3 Représentation des valeurs des tokens dans l'arbre

Il nous reste à représenter les valeurs des tokens dans l'arbre de syntaxe abstraite lui-même. Pour ce faire, il n'est pas nécessaire de changer la notion d'arbre : il suffit d'utiliser la possibilité d'avoir un ensemble infini d'étiquettes abstraites. Cet ensemble sera donc en fait un langage régulier (les entiers, les chaînes de caractères, etc.) aussi cet ensemble d'étiquettes abstraites sera souvent informellement donné en extension avec des points de suspension, ou encore sous forme d'une expression rationnelle.

Ainsi, pour représenter les expressions arithmétiques avec les valeurs des constantes, il suffit de prendre $T = \{+, \times, \text{Cte}\} \cup \mathbb{N}$ et la grammaire abstraite suivante :

$$\begin{array}{rcl} E & \rightarrow & +(E,E) \mid \times(E,E) \mid \text{Cte}(\textit{Nat}) \\ \textit{Nat} & \rightarrow & (0 \mid 1 \mid \cdots \mid 9)^+ \end{array}$$

Le langage défini par cette grammaire contiendra donc par exemple $\times(\text{Cte}(1), \text{Cte}(2))$ et $+(\text{Cte}(1), \times(\text{Cte}(2), \text{Cte}(3)))$.

Pour résumer, la figure 11.1 montre un tableau donnant deux exemples d'expressions arithmétiques, accompagnées de leur arbre de dérivation et leur arbre abstrait associé, où l'on constate à nouveau la simplicité et la concision apportée par la notion d'arbre abstrait. Nous avons donc atteint notre dernier objectif.

11.4 Calcul des arbres de syntaxe abstraite

Le dernier point qui reste à traiter dans ce chapitre est la façon d'associer à chaque mot d'un langage son arbre abstrait. Cela se fait facilement dans le cadre du logiciel YACC comme expliqué au paragraphe 10.2.5. Ainsi, si l'on poursuit notre exemple des expressions arithmétiques, on aura :

$$\begin{array}{rcll} E & \rightarrow & E + T & \{+(E_1, T_1)\} \\ E & \rightarrow & T & \{T_1\} \\ T & \rightarrow & T \times F & \{\times(T_1, F_1)\} \\ T & \rightarrow & F & \{F_1\} \\ F & \rightarrow & \textit{Nat} & \{\text{Cte}(\text{val}(\textit{Nat}_1))\} \\ F & \rightarrow & (E) & \{E_1\} \end{array}$$

Noter l'utilisation de la fonction val pour accéder à l'unité lexicale représentée par le token *Nat*. Par ailleurs, la notation $+(E_1, T_1)$ représente ici la construction d'un arbre de syntaxe dont la racine est étiquettée par le symbole $+$, et les sous-arbres proviennent récursivement de lévaluation de E_1 et T_1.

syntaxe utilisateur	syntaxe concrète	arbre de dérivation	arbre de syntaxe abstraite
`1+3*2`	$Nat + Nat \times Nat$	E $E \quad + \quad T$ $T \quad T \quad \times \quad F$ $F \quad F \quad Nat$ $Nat \quad Nat$ $\text{val}(Nat_1) = 1$ $\text{val}(Nat_2) = 3$ $\text{val}(Nat_3) = 2$	$+$ Cte $\quad \times$ 1 $\quad$ Cte $\quad$ Cte $\quad$ 3 $\quad$ 2
`(1+3)*2`	$(Nat + Nat) \times Nat$	E T $T \quad \times \quad F$ $F \quad Nat$ $(\quad E \quad)$ $E \quad + \quad T$ $T \quad F$ $F \quad Nat$ Nat $\text{val}(Nat_1) = 1$ $\text{val}(Nat_2) = 3$ $\text{val}(Nat_3) = 2$	$\times$ $+$ $\quad$ Cte Cte $\quad$ Cte $\quad$ 2 1 $\quad$ 3

FIG. 11.1 – Comparaison des syntaxes abstraites et concrètes

11.5 Codages des arbres de syntaxe abstraite

Nous terminons ce chapitre par quelques indications sur la méthode que l'on peut utiliser pour programmer ces arbres de syntaxe abstraite.

Dans le langage CAML, ce codage est immédiat en utilisant des types sommes. Ainsi, on définira le type des expressions arithmétiques par :

```
type expr =
  Cte of int
| Plus of expr * expr
| Mult of expr * expr
```

D'une manière générale, il y aura autant de types à définir que de sortes d'arbres de syntaxe abstraite, et il y aura autant de constructeurs que de règles de grammaire abstraite. Les ensembles infinis d'étiquettes abstraites comme les entiers ou les identificateurs seront codés simplement par le type de base correspondant : `int` ou `string`. Cela est notre seule façon de coder des ensembles infinis d'étiquettes abstraites. Nous imposerons donc que tout ensemble infini d'étiquettes abstraites corresponde à l'un des types de base parmi `int`, `string`, et `real`.

Dans un langage comme PASCAL ou C n'offrant pas cette possibilité des types sommes et d'allocation automatique de mémoire, il est nécessaire de coder ces arbres à l'aide de pointeurs sur des enregistrements, contenant un champ pour l'étiquette abstraite et des champs pour les sous-arbres. Dans un langage avec objets comme C++ ou JAVA, on aura avantage à coder les arbres par des classes, la construction et la destruction de ces arbres en seront facilitées.

11.6 Exercices

Exercice 11.1

1. *Écrire une grammaire abstraite pour les expressions booléennes formées avec les opérations « ou », « et » et « non », et les constantes « vrai » et « faux ».*
2. *Donner une grammaire concrète non-ambiguë pour le langage des expressions booléennes avec toutes les opérations ci-dessus, et donner les règles de calcul de la syntaxe abstraite associées.*
3. *Donner un type CAML représentant cette syntaxe abstraite.*
4. *Écrire une fonction CAML permettant de transformer une chaine de caractères représentant une expression booléenne en une expression abstraite, en utilisant l'outil CAMLYACC.*

Exercice 11.2

1. *Écrire une grammaire abstraite pour les expressions arithmétiques sur l'addition, la multiplication, la soustraction, la division, l'opposé, l'exponentiation, les constantes entières ou réelles et les variables. (On ne distinguera pas les constantes entières, on les codera par des réels.)*
2. *Donner une grammaire concrète non-ambiguë pour le langage des expressions ci-dessus, et donner les règles de calcul de la syntaxe abstraite associées.*
3. *Donner un type CAML représentant cette syntaxe abstraite.*
4. *Écrire une fonction CAML permettant de transformer une chaîne de caractères représentant une expression arithmétique en une expression abstraite, en utilisant l'outil CAMLYACC.*

Chapitre 12

Machines de Turing

Conformément à la figure 12.1, les machines de Turing considérées sont dotées :

1. d'une bande de lecture bi-infinie, sur laquelle la donnée, mot sur le vocabulaire V_t, sera écrite entourée à l'infini de caractères blancs, considéré comme notre caractère spécial noté _ n'appartenant pas à V_t ;
2. d'un nombre fini de bandes de travail qui peuvent stocker des mots entourés à l'infini de blancs sur les vocabulaires respectifs $V_1, \ldots, V_n$ ne contenant pas le caractère blanc ;
3. d'un contrôle matérialisé par un ensemble fini Q d'états ;
4. d'un état initial q_0 ;
5. d'un ensemble $F \subseteq Q$ d'états acceptants ;
6. et d'une fonction de transition T, qui est une application de $Q \times V_t \times V_1 \times \ldots \times V_n$ dans
   - $Q \times \{L, R\} \times \{L, R\} \times \ldots \times \{L, R\}$ si la machine est déterministe,
   - et $\mathcal{P}(Q \times \{L, R\} \times \{L, R\} \times \ldots \times \{L, R\})$ si la machine est non déterministe.

Lors de chaque transition, la machine de Turing lit un mot sur chaque bande, éffectue une transition d'état accompagnée du déplacement du lecteur de chaque bande conformément à la fonction T. La notion de reconnaissance est ici un peu différente de celle que l'on a vue pour les automates : il suffit d'accéder à un état acceptant sans qu'il soit nécéssaire d'avoir lu le mot à reconnaitre en entier. La reconnaissance d'un mot arbitraire peut donc s'éffectuer en zéro transition, il suffit de déclarer que q_0 est acceptant. Cette machine, bien sûr, ne vérifie même pas que le mot est formé de lettres appartenant à l'alphabet autorisé pour la bande entrée.

On peut bien sûr prendre une autre définition où l'on force la lecture des données, mais ce modèle présente quelques inconvénients du point de vue de la théorie de la complexité, en particulier l'absence de complexité sub-linéaire.

12.1 Exercices

FIG. 12.1 – La machine de Turing.

Chapitre 13

Complexité en temps et en espace

On définit la complexité des langages relativement à un modèle de calcul particulier, les machines de Turing. Cela peut sembler limiter la portée des constructions, mais on s'aperçoit vite que ce modèle résiste à de nombreuses variations, et que tous les modèles de la calculabilité connus donnent peu ou prou les mêmes notions de complexité.

On peut se demander si la notion de langage est bien celle qui convient, puisque l'on est en général intéressé par des *problèmes*, et non des langages. En fait, tout problème peut se coder sous la forme d'un langage sur un alphabet finio, la précision est d'importance, par un codage appoprié des données et résultats du problème. On pourra donc parler indifféremment de langage ou de problème.

13.1 Classes de complexité

On dira qu'une machine de Turing M est bornée en temps par la fonction $T(n)$ si elle ne fait pas plus de $T(n)$ transitions pour une donnée de taille n. On s'intéressera tout particulièrement au cas où T est un polynôme. Dans ce cas, on dira que M travaille en espace polynomial.

On dira qu'une machine de Turing M est bornée en espace par la fonction $S(n)$ si elle ne balaie pas plus de $S(n)$ cellules sur chaque bande pour une donnée de taille n. On s'intéressera tout particulièrement au cas où S est un polynôme. Dans ce cas, on dira que M travaille en temps polynomial.

Ces notions servent à classifier les langages suivant le temps -ou l'espace- nécéssaire pour reconnaitre les mots qui en font partie. On dira ainsi qu'un langage L appartient à la classe $DTIME(T(n))$ s'il existe une machine de Turing déterministe M qui reconnait un mot $u \in L$ en temps au plus $T(\mid u \mid)$, et à la classe $NTIME(T(n))$ s'il existe une machine de Turing non déterministe M qui reconnait un mot $u \in L$ en temps au plus $T(\mid u \mid)$. La définition des classes $DSPACE(S(n))$ et $NSPACE(S(n))$ est similaire.

Finallement, on dira qu'un langage L appartient à la classe P (respectivement NP) s'il existe un polynôme $P(n)$ tel que L soit reconnu en temps $DTIME(P(n)$ (respectivement, $NTIME(P(n))$). La définition des classes $PSPACE$ et $NPSPACE$ est similaire.

De nombreuses autre classes jouent un rôle important, en particulier les classes logarithmiques en espace, $LOGSPACE$ qui est déterministe, et $NLOGSPSACE$ qui est non déterministe -l'espace autorisé est une puissance quelconque du logarithme de la taille de l'entrée-, ainsi que les classes exponentielles en temps $EXPTIME$ et $NEXPTIME$. Enfin, la classe de complexité élémentaire (en temps) caractérise les langages dont la reconnaissance est bornée en temps par une hauteur arbitraire d'exponentielles emboitées, comme n^{n^n}, de hauteur 3. On ne parle dans ce cas que de complexité déterministe. Cette classes interviendra par la suite.

Le nombre de bandes utilisées n'a guère d'importance pour les classes de complexité $P, NP, PSPACE, NPSPACE$: on peut le réduire à 2 en passant de $DTIME(T(n))$ à $DTIME(T(n)logT(n))$, et même à un en passant de $DTIME(T(n))$ à $DTIME(T(n)^2)$.

Le tableau qui suit récapitule les classes de complexité que nous venons de décrire.

	MT déterministe	MT non déterministe
Temps $T(n)$	$DTIME(T(n))$	$NTIME(T(n))$
Temps polynomial	P	NP
Temps exponentiel	EXP	$NEXP$
Temps élémentaire		
Temps non élémentaire		
Espace $S(n)$	$DSPACE(S(n))$	$NSPACE(S(n))$
Espace logarithmique	$LOGSPACE$	$NLOGSPACE$
Espace polynomial	$PSPACE$	$NPSPACE$

Une dernière classe importante est $CoNP$. On dit qu'un langage $L \subseteq V^*$ est dans $CoNP$ ssi son complémentaire $\overline{L}$ est dans NP. Cette définition est plus subtile qu'il n'y parait à première vue : pour reconnaitre $u \in L$ avec la machine non déterministe M reconnaissant $\overline{L}$, il est nécéssaire de rejeter u, c'est-à-dire d'essayer toutes les façons possibles de reconnaitre u avec M pour vérifier qu'elles échouent toutes. Peut-on reconnaitre u avec une machine de Turing non-déterministe M ? On n'en sait rien, le problème est ouvert.

13.2 Comparaisons entre mesures de complexité

Tout d'abord, notons que

$$DTIME(T(n)) \subseteq DSPACE(T(n))$$

puisque il faut au moins n transition pour visiter n cases dds bandes mémoires.

Pour donner une "réciproque", il faut d'abord s'assurer que borner l'espace implique un temps borné. Cela revient à montrer que si la machine M travaille en espace borné, alors il existe une machine de Turing M' qui reconnait le même langage, et travaille en espace et en temps borné à la fois. Plus précisément, on peut montrer que

$$DSPACE(S(n)) \subseteq DTIME(c^{S(n)})$$

où la constante c dépend de la machine M. On montre une relation similaire entre temps déterministe et non déterministe :

$$NTIME(T(n)) \subseteq DTIME(c^{T(n)})$$

où, à nouveau, la constante c dépend de la machine M. Par contre, la relation entre espace déterministe et non déterministe est bien différente, c'est le théorème de Savitch :

$$NSPACE(S(n)) \subseteq DSPACE(S(n)^2)$$

ce qui implique que

$$NPSPACE = PSPACE$$

et

$$NLOGSPACE = LOGSPACE.$$

Les inégalités connues entre classes de complexité sont les suivantes :

$$DSPACE(log\ n) \subseteq P \subseteq \{NP,\ CoNP\} \subseteq PSPACE,$$

$$DSPACE(log\ n) \subset PSPACE,$$

$$LOGSPACE \subseteq PSPACE$$

FIG. 13.1 – Inclusions (au sens large) essentielles.

et

$$P \neq LOGSPACE.$$

On voit que la pluspart de ces inégalités ne sont pas strictes ; par ailleurs, on ne sait pas si l'une des classes P et $LOGSPACE$ est incluse dans l'autre. La théorie de la complexité s'avère plus riche de questions que de réponses. La question de savoir si $P = NP$ ou pas et considérée comme l'une des grandes questions actuelles non seulement de l'informatique, mais de l'ensemble des mathématiques.

Les inclusions les plus importantes sont représentées à la figure 13.1.

13.3 Langages complets

De nombreux langages sont dans NP pour lesquels aucune solution polynomiale n'est connue. Le but de ce paragraphe et d'identifier parmi eux ceux qui sont difficiles, en ce sens que si l'un d'eux devait être polynomial, alors les classes P et NP coincideraient. La notion qui permet de comparer la difficulté de deux langages différents fait l'objet du paragraphe qui suit. Elle a une portée générale, qui dépasse le problème de comparer P et NP.

13.3.1 Réductions et complétude

Définition 13.1 *Le langage L' est réductible au langage L en temps polynomial, on dit aussi* polynomialement réductible en temps, *par la machine de Turing $M : L' \mapsto L$ si :*

1. *M fonctionne en temps polynomialement borné en fonction de la taille de son entrée ;*
2. *$M(x) \in L$ ssi $x \in L'$.*

Définition 13.2 *Le langage L' est réductible au langage L en espace logarithmique, on dit aussi* logarithmiquement réductible en espace, *par la machine de Turing $M : L' \mapsto L$ si :*

1. *M fonctionne en espace logarithmiquement borné en fonction de la taille de son entrée ;*
2. *$M(x) \in L$ ssi $x \in L'$.*

Lemme 13.3 *Supposons que L' soit polynomialement réductible en temps à L. Alors,*

1. *$L \in NP \Longrightarrow L' \in NP$;*
2. *$L \in P \Longrightarrow L' \in P$.*

Lemme 13.4 *Supposons que L' soit logarithmiquement réductible en espace à L. Alors,*

1. *$L \in P \Longrightarrow L' \in P$;*
2. *$L \in LOGSPACE \Longrightarrow L' \in LOGSPACE$.*

Ce sont les réductions qui vont nous permettre de comparer les langages entre eux :

Définition 13.5 *Soit $\mathcal{C}$ une classe de complexité. On dit que L est complet pour $\mathcal{C}$ vis-à-vis de la réduction polynomiale en temps (respectivement la réduction logarithmique en espace) si*

1. *$L \in \mathcal{C}$ ($\mathcal{C}$-difficulté) ;*
2. *tout langage $L' \in \mathcal{C}$ est polynomialement réductible en temps (respectivement, logarithmiquement réductible en espace) à L*

13.3.2 NP-complétude

On connait un grand nombre de problèmes $NP - complets$, c'est le cas de la pluspart de problèmes de nature combinatoire. On connait même des problèmes qui sont dans NP mais dont on sait pas s'ils sont dans P, ni s'ils sont complets. C'est le cas du test de primalité d'un entier.

Le problème paradigmatique de la classe NP est SAT : il s'agit de savoir si une formule propositionnelle donnée est satisfiable ou pas.

On considère le langage des formules logiques propositionnelles, c'est-à-dire baties à l'aide d'un ensemble donné $\mathcal{P}rop$ de symboles (ou variables) propositionnels pouvant pendre les deux valeurs $\{0, 1\}$ et des connecteurs logiques $true, false, \wedge, \vee, \neg$. Une formule propositionnelle est dite satisfiable s'il existe une application de $\mathcal{P}rop$ dans $\{0, 1\}$ telle qu'elle s'évalue en 1, $true, false$ étant des constantes logiques s'évaluant respectivement en 0 et 1.

Le problème SAT consiste à déterminer l'existence d'une telle application pour une formule donnée, ϕ, arbitraire.

On peut représenter le problème SAT comme le langage L_{SAT} construit comme suit : s'il y a n variables, la ième sera représentée par le mot formé de la lettre x suivie du code binaire de i. Notre alphabet sera donc $\{\wedge, \vee, \neg, (,), x, 0, 1\}$, permettant de coder une formule de longueur n par un mot de longueur (l'arrondi supérieur de) $n\ log\ n$. En effet, il y a au plus $n/2$ variables différentes dans une formule de taille n, et donc le code de chacune d'entre elles ne nécéssite pas plus de (l'arrondi supérieur de) $1 + log\ n$ lettres du vocabulaire. Comme nous allons utiliser une réduction en espace logarithmique, on va pouvoir faire comme si une formule de taille n était codée par un mot de même taille n, car $log(n\ log\ n) \leq log\ n^2 = 2\ log\ n$, et que toutes ls quantitées sont à une constante multiplicative près.

Lemme 13.6 *SAT est dans NP.*

Preuve: Pour cela, il suffit de construire de manière non-déterministe des valeurs de vérité pour toutes les variables, puis d'évaluer la formule, ce qui se fait en temps linéaire en la taille de la formule. □

D'une manière générale, les preuves d'appartenance à NP se décomposent en deux phases :

1. la première consiste à construire un arbre de choix de hauteur polynomiale, qui figure l'ensemble de tous les cas devant être examinés, en temps (non-déterministe) proportionnel à sa hauteur en utilisant le non-déterminisme des machine de Turing. Dans notre exemple, il s'agit de construire toutes les possibilités d'affecter des valeurs de vérité pour les variables ;
2. pour chaque choix, un calcul polynomial avec une machine déterministe. Lors de cette seconde phase, il s'agit de *vérifier* une propriété ; dans notre exemple, il s'agit de vérifier que les valeurs de vérité affectées aux variables rendent la formule vraie. C'est là une caractéristique essentielle des problèmes NP.

Il suffit pour terminer de collecter les résultats aux feuilles afin de vérifier s'il existe un choix d'affectation de valeurs de vérité aux variables qui rende la formule vraie, ce qui est fait par la définition de l'acceptation pour les machines non-déterministes.

Théorème 13.7 *SAT est NP-complet.*

Preuve: Il nous reste à montrer que tout problème de la classe NP est réductible à L_{SAT} en espace logarithmique en la taille de la donnée. Pour cela, étant donnée une machine de Turing non déterministe M -dont nous supposerons qu'elle possède une bande unique infinie à droite seulement pour plus de facilité- qui reconnait son entrée $x = x_1 \dots x_n$ de taille n en temps borné par un polynôme $p(n)$, nous allons définir un algorithme qui travaille en espace logarithmique pour construire une formule propositionnelle M_x qui est satisfiable ssi M accepte x. Cet algorithme pourra être facilement décrit sous la forme d'une machine de Turing prenant en entrée une description du calcul de M pour

sa donnée x, et écrivant la formule M_x sur sa bande de sortie, et utilisant par ailleurs des bandes de travail pour éffectuer ses calculs.

Soit $\#m_0\#m_1 \ldots \#m_{p(n)}$ la description d'un calcul de M formé de $p(n)$ transitions. Si la machine M accepte avant la $p(n)$ème transition, on répètera une transition vide sur l'état acceptant -ce qui ne change pas le langage reconnu de M ni le polynôme $p(n)$- de manière à ce qu'il y ait exactement $p(n)$ transitions et donc $(p(n)+1)^2$ caractères au total. Les $p(n)+1$ mots m_i de longueur $p(n)$ décrivent la configuration de la machine M après exactement i transitions. Ils sont de la forme $\alpha_i(q_i, a_i, b_i, D_i, q_{i+1})\beta_i$, où

- q_i désigne l'état courant, α_i est le mot écrit sur la bande depuis l'extrémité gauche jusqu'à la tête de lecture non comprise,
- a_i est le caractère lu,
- b_i est le caractère écrit lors de la transition à venir (choisi arbitrairement pour $i = p(n)$),
- D_i est la direction de déplacement de la tête de lecture lors de la transition à venir (choisie arbitrairement pour $i = p(n)$),
- q_{i+1} désigne l'état atteint dans la transition à venir (choisi arbitrairement pour $i = p(n)$),
- β_i est le mot écrit sur la bande à partir de la tête de lecture non-comprise jusqu'au premier blanc rencontré, et complété si nécéssaire par des blancs de manière à ce que le mot $\alpha_i a_i \beta_i$ ait exactement $p(n)$ caractères.

Chaque mot $\#\alpha(q, a, b, D, q')\beta$ est appellé une *description instantanée*, où les quintuplets (q, a, b, D, q') sont considérés comme les lettres d'un alphabet fini particulier W dont le sous-ensemble formé des quintuplets $(f \in F, a, b, D)$ est noté W_F. On utilisera la notation pointée de la programmation par objets pour récupérer les composantes des éléments de W. Notons par ailleurs que β peut être une séquence (éventuellement vide) formée de blancs uniquement si la tête de lecture pointe sur un blanc.

Á chaque caractère X qui peut figurer dans la descrition d'un calcul, on associe une variable propositionnelle $c_{i,j,X}$ qui servira à indiquer si le $(j+1)$ème caractère (on commence avec $j = 0$) de la ième description instantanée est un X. L'expression M_x cherchée doit assurer les propriétés suivantes :

1. Mot : les variables $c_{i,j,X}$ décrivent un mot, et donc l'une d'elle exactement est vraie pour chaque paire (i, j) ;
2. Init : m_0 décrit l'état initial de la machine M avec i sur la bande ;
3. Final : $m_{p(n)}$ possède un état acceptant ;
4. Trans : pour chaque $i \in [0, p(n)]$, m_{i+1} découle de m_i par une transition de la machine non déterministe M.

La formule M_x cherchée est donc la conjonction des formules décrivant ces quatre propriétés. Décrivons les dans notre langage logique :

$$\text{Mot} = \bigwedge_i \bigwedge_j \left(\bigvee_X c_{i,j,X} \wedge \neg \left(\bigvee_{X \neq Y} (c_{i,j,X} \wedge c_{i,j,Y}) \right) \right)$$

$$\text{Init} = c_{0,0,\#} \wedge \bigvee_{X \in W} c(0,1,X) \wedge \bigwedge_{j \in [1,n]} c_{0,j,x_j} \wedge \bigwedge_{j \in [n+1,p(n)]} c_{0,j,\_}$$

$$\text{Final} = \bigwedge_j \bigwedge_{X \in W_F} c_{p(n),j,X}$$

$$\text{Trans} = \bigwedge_{i \in [0,p(n)-1]} \bigwedge_{j \in [0,p(n)]} \bigvee_{\substack{W,X,Y,Z \text{ such} \\ \text{that} f(W,X,Y,Z)}} c_{i,j-2,W} \wedge c_{i,j-1,X} \wedge c_{i,j,Y} \wedge c_{i+1,j,Z}$$

où le prédicat de filtrage $f(W, X, Y, Z)$ décrit la possibilité pour le caractère Z d'apparaitre en position j de la $(i+1)$ème description instantanée, sachant que les caractères W, X et Y apparaissent aux positions respectives j_1, j et $j+1$ de la ième description instantanée. Il est important de noter que ce prédicat doit simplement être calculé à partir de la donnée de la machine M, sans qu'il soit nécessaire de le construire. Il sert en effet à éliminer les calculs impossibles de la formule. Il faudrait au contraire le construire dans notre langage si nous avions défini

$$\text{Trans} = \bigwedge_{i\in[0,p(n)-1]} \bigwedge_{j\in[0,p(n)]} \bigvee_{W,X,Y,Z} f(W,X,Y,Z) \Longrightarrow c_{i,j-2,W} \wedge c_{i,j-1,X} \wedge c_{i,j,Y} \wedge c_{i+1,j,Z}$$

Il faut maintenant montrer qu'il est possible de construire la formule M_x, ce qui inclue le calcul du prédicat f, en espace logarithmique. Cela résulte du fait que la taille de M_x est de l'ordre de $p(n)^2$, et qu'il suffit d'un espace logarithmique pour compter jusqu'à une valeur qui s'exprime comme un polynôme de n.

Il reste enfin à montrer que la machine M accepte la donnée x si et seulement si la formule M_x est satisfiable. Comme tout a été fait pour que ce soit vrai, la preuve en est laisée au lecteur. □

Les problèmes NP-complets sont très nombreux, comme la programmation linéaire en nombres entiers, le problème du sac à dos, le problème du voyageur de commerce, ou comme la recherche dans un graphe d'un circuit hamiltonien, d'un chemin de longueur minimale, d'un coloriage en 4 couleurs. La recherche d'un chemin eulérien, par contre est polynomiale.

On peut se demander où est la frontière entre le polynomial et le polynomial non-déterministe. Appelons $n - CNF$, la restriction du problème SAT au cas où la formule propositionnelle est une conjonction de clauses comprenant toutes le même nombre n de littéraux. On peut montrer que $2 - CNF$ est polynomial, alors que $3 - CNF$ est déja NP-complet.

Les problèmes NP-complets ont longtemps été considérés comme intractable. Qu'un problème soit NP-complet n'empêche pas de le résoudre, en général, du moins ses instances de taille pas trop grande. Il existe souvent de nombreuses sous-classes qui sont polynomiales, comme $2 - CNF$ dans le cas de SAT. La difficulté est généralement concentrée sur des problèmes très particuliers, qui font apparaître des phénomènes de seuils pour certains paramètres : loin du seuil, le problème est polynomial ; près du seuil, il devient difficile, mais il peut alors être possible de trouver une solution approximative en utilisant des algorithmes probabilistes.

13.3.3 PSPACE-complétude

La classe $PSPACE$ est tout aussi mystérieuse. Le problème $PSPACE$-complet paradigmatique de cette classe est un autre problème de logique, QBF pour Quantified Boolean Formulae, pour lequel il s'agit de décider si une formule propositionnelle quantifiée est satisfiable ou pas. Le langage est donc cette fois bâti à partir d'un nombre fini de variables propositionnelles, des connecteurs logiques habituels, et des quantificateurs universel et existentiel. C'est l'alternance des quantificateurs qui va nous propulser dans une classe dont on pense qu'elle et intrinsèquemnt plus complexe que la classe NP. On dit qu'une variable est libre dans une formule si elle n'est pas dans la portée d'un quantificateur.

Lemme 13.8 *QBF est dans NP.*

Preuve: La preuve utilise l'élimination des quantificateurs sur un ensemble fini, ici l'ensemble des valeurs de vérité. On a en effet les équivalences logiques :

$$\forall x \phi(x) \equiv \phi(0) \wedge \phi(1)$$

$$\exists x \phi(x) \equiv \phi(0) \vee \phi(1)$$

Le problème est qu'étant donnée une formule de taille n, la formule transformée peut être de taille exponentielle en n :

$$\forall^n\ x_1 x_2 \ldots x_n \text{ est de taille } 4n-1$$

FIG. 13.2 – La formule transformée de taille exponentielle.

mais sa transformée représentée à la figure 13.2 est de taille $2^{n-1} - 1 + 2^n \times (n-1) > (n-1)^2\, 2^n$.

On va donc parcourir l'arbre ci-dessus sans l'engendrer, en définissant une procédure d'évaluation récursive $EVAL$ qui utilise un espace de tavail polynomial dans la pile d'exécution pour évaluer ϕ :

1. $\phi = true$: EVAL retourne la valeur $true$;
2. $\phi = \phi_1 \wedge \phi_2$: EVAL retourne la conjonction des résultats obtenus récursivement pour ϕ_1 et ϕ_2 ;
3. $\phi = \phi_1 \vee \phi_2$: EVAL retourne la disjonction des résultats obtenus récursivement pour ϕ_1 et ϕ_2 ;
4. $\phi = \neg\psi$: EVAL retourne la négation du résultat obtenu récursivement pour ψ ;
5. $\phi = \exists x\psi$: EVAL construit les formules ψ_0 et ψ_1 en remplaçant respectivement la variable x par 0 et 1 dans ψ, puis retourne la disjonction des résultats obtenus récursivement pour ψ_0 et ψ_1 ;
6. $\phi = \forall x\psi$: EVAL construit les formules ψ_0 et ψ_1 comme précédemment, puis retourne la conjonction des résultats obtenus récursivement pour ψ_0 et ψ_1.

Comme le nombre de connecteurs ou quantificateurs est au plus égal à la taille n de la formule, la profondeur de récursion est au plus n. La taille des données à stocker dans la pile d'exécution étant linéaire en n, on en déduit que l'espace occuppé dans la pile (pour laquelle on utilisera par exemple une bande spécifique de la machine) est quadratique. Donc QBF est dans $PSPACE$. □

Les preuves d'appartenance à $PSAPCE$ patissent du fait que $NPSPACE = PSPACE$. On pourrait en déduire que le même schéma va à nouveau fonctionner, en construisant un arbre de choix en espace polynomial non-déterministe. En fait, cela n'est pas le cas. Les problèmes $PSPACE$, comme le problème QBF, ne se prètent pas à la décomposition en une première phase d'énumération non-déterministe qui contient toute la complexité suivie d'une phase de vérification polynomiale. Vérifier qu'un ensemble de valeurs de vérité est une solution d'un problème de QBF est tout ausi difficile que de les trouver.

Théorème 13.9 *QBF est PSPACE-complet.*

Preuve: Reste à réduire tout problème de $PSPACE$ à QBF par une réduction que l'on choisit polynomiale en temps. La réduction est similaire à la réduction précédente pour NP, en plus complexe. □

Nous terminons par un problème qui intervient de manière cruciale en vérification : l'accessibilité dans un graphe consiste, étant donnés un graphe G et deux sommets a et b, à déterminer s'il existe un chemin allant de a à b.

Théorème 13.10 *L'accessibilité dans un graphe est $NSPACE(log\ n)$-complet pour les réductions en espace logarithmique.*

Preuve: Le codage du problème comme un langage sur un alphabet fini est laissé au lecteur. Notons toutefois que le codage d'un noeud du graphe prendra $log\ n$ bits s'il y a n sommets.

On montre tout d'abord que le problème est dans la classe $NSPACE(log\ n)$. Pour cela, on va engendrer tous les chemins grâce au non-déterminisme, de sorte que la mémoire occuppée soit de taille logarithmique en n pour chacun d'eux. Pour cela, on va à chaque étape (il y aura au plus n étapes) engendrer tous les sommets et vérifier que l'on construit bien un chemin d'une part, et

si le sommet engendré est le sommet voulu d'autre part. Cela suppose de mémoriser pour chaque chemin partiel le sommet qui vient d'être engendré plus le sommet précédent, ce qui se fait en espace logarithmique. On ne conserve donc pas le chemin.

Il reste à réduire tout problème de $NSPACE(log\ n)$ au problème d'accessibilité par une machine de Turing déterministe fonctionnant en espace logarithmique. La réduction est esquissée, les détails sont laissés au lecteur.

Soit M une machine de Türing non-déterministe fonctionnant en espace logarithmique de la taille de son entrée, et x une donnée de taille n. La description de la machine à un moment donné se fait avec $log\ n$ bits, puisqu'elle fonctionne en espace logarithmique. Nous allons contruire un graphe M_x qui va coder le fonctionnement de la machine et aura un chemin du premier noeud au dernier ssi la machine M reconnait x. Les noeuds du graphe seront les descriptions instantanées de la machine plus un dernier pour l'acceptance. Le premier noeud est la description initiale. il y aura un arc d'un noeud à un autre si la transition correspondante est posible. Ce graphe répond bien à la question, et il faut montrer qu'on peut l'engendrer avec une machine qui travaille en espace logarithmique. Cela est du au fait que chaque description instantanée est codée sur approximativement $log\ n$ bits. □

Cette preuve est intéressante car elle mêle la génération aléatoire incrémentale d'un objet (ici, un chemin dans un graphe) avec le test que cette génération construit bien l'objet en question. Cela est nécéssité par le besoin de réduire la taille occuppée à l'incrément de l'objet, l'objet lui-même prenant trop de place.

13.4 Exercices

Exercice 13.1 *Montrer que le problème de satisfiabilité pour CNF est $NP - complet$.*

Exercice 13.2 *Montrer que le problème de satisfiabilité pour $3 - CNF$ est $NP - complet$.*

Pour montrer la complétude, on réduira le problème SAT au problème $3 - CNF$. On pourra dans un premier temps se débarasser des négations internes en les faisant descendre aux feuilles en utilisant pour ce faire un espace de travail logarithmique. Dans un second temps, il faudra créer la formule $3 - CNF$ à partir de la formule obtenue à l'issue de la première passe. La seconde passe consiste à engendrer les disjonctions de la clause les unes après les autres en montrant qu'il suffit de n clauses et d'un espace logarithmique pour les construire une par une.

Exercice 13.3 *Montrer que le problème de satisfiabilité pour $2 - CNF$ est dans P. Est-il complet pour P ?*

Exercice 13.4 *On appelle* couverture *d'un graphe G un sous ensemble A de ses sommets tel que pour toute arète a, b, $a \in A$ ou $b \in A$.*

Montrer que le problème, étant donné un graphe G et un nombre k, de déterminer si G a une couverture de taille k est NP-complet.

Exercice 13.5 *Montrer que le problème du coloriage d'un graphe avec un nombre de couleurs minimal est NP-complet.*

Exercice 13.6 *On appelle* circuit *hamiltonien d'un graphe G un circuit passant une fois et une seule par tout sommet de G.*

Montrer que le problème, étant donné un graphe G, de déterminer s'il possède un circuit hamiltonien est NP-complet.

Exercice 13.7 *On appelle* circuit *eulerien d'un graphe G un circuit passant une fois et une seule par tout arc de G.*

Montrer que le problème, étant donné un graphe G, de déterminer s'il possède un circuit eulerien est dans P.

Exercice 13.8 *Montrer que le problème, étant donné un entier n, de déterminer s'il est premier est dans NP.*

Bibliographie

[1] Béatrice Bérard, Michel Bidoit, François Laroussinie, Antoine Petit et Philippe Schnoebelen *Vérification de logiciels : Techniques et Outils du model-checking*, 1999. Vuibert Informatique. Philippe Schnoebelen, coordinateur.

[2] John E. Hopcroft et Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*, 1979. Addison Wesley.