INFZ24, Informatique et linguistique II

AUDIBERT Laurent [1]

22 octobre 2003

1. Jeune équipe DELIC - Université de Provence - 29 Avenue Robert SCHUMAN - 13621 Aix-en-Provence Cedex 1 - laurent.audibert@up.univ-aix.fr

La linguistique informatique ou, suivant l'appellation anglo-saxonne, la linguistique computationelle, est une discipline issue des développements de l'informatique dans le domaine des sciences du langage. Le domaine de la linguistique informatique recouvre toutes les applications informatiques qui ont trait au langage naturel. Par langage naturel on entend le langage tel qu'il est parlé spontanément par les êtres humains (le français, l'anglais, le russe, le wolof ...), par opposition aux langages formels utilisés en logique, en mathématique, en informatique, ...

L'objet premier de la linguistique informatique est le traitement automatique du langage (TAL), c'est-à-dire l'analyse et la génération automatique du langage, plus précisément encore, l'élaboration de modèles computationnels d'analyse et de génération à partir desquels on peut réaliser des logiciels capables de comprendre ou de produire des énoncés en langue naturelle.

Cette unité d'enseignement présente les bases de la théorie des automates et des langages formels : automates à états finis, expressions régulières, grammaires régulières, grammaire indépendantes du contexte, analyseurs syntaxiques.

Table des matières

Cours 1

La linguistique informatique

Ce cours présente dans un premier temps quelques-unes des applications du traitement automatique des langues.

Nous parlerons également de la composante syntaxique d'un système de traitement automatique des langues.

Nous aborderons enfin la question de l'ambiguïté du langage qui constitue un problème fondamental pour l'analyse syntaxique et sémantique.

1.1 Domaines d'application

Les enjeux économiques liés au développement d'application du traitement des langues naturelles sont importants, comme en témoigne le recours de plus en plus fréquent au terme d'industrie de la langue. Les domaines d'application vont de la recherche d'informations à la traduction automatique en passant par les systèmes de communication homme/machine et les vérificateurs d'orthographe.

1.1.1 Systèmes de communication homme-machine

Le problème de la communication entre l'homme et la machine est un problème crucial pour un grand nombre d'applications informatiques. Il recouvre tous les aspects liés à l'interaction en langue naturelle entre un ordinateur et son utilisateur. Concrètement, il s'agit de développer des programmes capables de comprendre des ordres donnés en langue naturelle et d'y répondre également en langue naturelle écrite ou parlé. Le succès de ces programmes dépend largement des limites du domaine d'application et de la richesse d'expression qui lui est liée. Dans des domaines très restreints, avec des formes d'expressions très limitées et très simples, les résultats peuvent être impressionnants. Par contre, la conversation à bâtons rompus avec un ordinateur n'est sans doute pas pour demain.

A titre d'exemple, dans le domaine en pleine expansion des bases de données, les interfaces en langue naturelle permettent à un usager de formuler ses requêtes en langue naturelle plutôt que dans un langage formel d'interrogation de bases de données tel que `SQL`. C'est le système d'interface qui se charge de convertir la requête en français en une requête `SQL` interprétable par la base de données.

1.1.2 Recherche d'information

Par recherche d'information (*information retrieval*), on entend la conception de programmes capables de fouiller de gigantesques masses de documents en langue naturelle pour en extraire les informations désirées.

Les moteurs de recherche sur internet constituent de bons exemples d'application du type recherche d'information.

Les programmes disponibles à l'heure actuelle ne vont guère au-delà de la recherche de mots-clés (un peu comme les fonctions de recherches dans les éditeurs de texte courants) et de combinaisons de mots-clés. Ils ne tiennent pas compte des variations morphologiques, de la synonymie et ne cherche pas non plus à lever l'ambiguïté sémantique des mots de la requête.

1.1.3 Vérificateurs et correcteurs orthographiques

Un vérificateur orthographique est un programme capable de repérer les erreurs orthographiques dans un texte donné. Un correcteur propose en plus une correction. Les premiers systèmes de vérification orthographique se limitaient à signaler les formes suspectes en comparant tous les mots du texte à une liste préétablie. Des systèmes plus sophistiqués sont apparus au cours des quelques dernières années, qui cherchent à effectuer une analyse syntaxique locale de façon à affiner la vérification. Le recours à une analyse syntaxique, même partielle, des phrases à vérifier ou à corriger entraîne une augmentation considérable de la complexité de la tâche.

1.1.4 Dictée vocale

Un système de dictée vocale est une sorte de machine à écrire à laquelle on dicte un texte au lieu de la taper. Il s'agit donc d'un système capable de comprendre la parole et de la transcrire en texte. La difficulté première à laquelle doit faire face cette application est l'analyse de la parole, c'est à dire le découpage du flot de paroles en une séquence de mots. Etant donné les nombreuses incertitudes inhérentes au traitement de signal et à la reconnaissance phonétique[1] ainsi que les très nombreux cas d'homophonie[2] ainsi que ceux liés à l'articulation[3] une analyse linguistique s'avère indispensable pour tenter de déterminer les choix les plus plausibles sur la base des contraintes linguistiques. Les systèmes commercialisés sont encore très contraignants et nécessitent, entre autre, une phase d'apprentissage.

1.1.5 Synthèse de la parole

Les systèmes de synthèse de la parole (*text-to-speech*) sont des systèmes de lecture à haute voix à partir d'un texte écrit. Le passage du message écrit au message parlé se fait par l'intermédiaire d'un système de synthèse de la voix, auquel on envoie les séquences de phonèmes correspondant aux mots du message. Bien que des progrès importants aient

1. Distinction *gronde*/*grande* par exemple.
2. Mots qui se prononcent de la même façon mais s'écrivent différemment : *pain*/*peint*/*pin*, *pense*/*panse*, *mange*/*manges*/*mangent*, etc.
3. Par exemple *l'appareil*/*la pareille*, *ton nombre*/ *ton nombre*.

été accomplis au cours des quelques dernières années en matière de synthèse, les systèmes actuels présentent encore des lacunes importantes, principalement en ce qui concerne les aspects prosodiques (intonation, durée, etc.), la qualité de la voix synthétisée (trop monotone), le traitement des homographes hétérophones (mots qui s'écrivent de la même façon mais qui se prononcent différemment), et le traitement des liaisons entre mots. Ces lacunes limitent encore considérablement les possibilités d'utilisation de ces systèmes de synthèse.

1.1.6 Traduction automatique

Par traduction automatique, on entend la traduction d'un texte d'une langue à une autre langue effectuée par un ordinateur. Il est clair que les besoins en matière de traduction sont considérables et en constante augmentation, ce qui garantit un marché substantiel pour des systèmes capables de satisfaire les exigences des utilisateurs. Ainsi la traduction automatique occupe une place privilégiée dans le domaine de la linguistique informatique. Cela est d'autant plus vrai que la traduction fait intervenir toutes les composantes majeures de la linguistique informatique, comme l'analyse, les lexique, la génération et cela à un niveau très général et donc potentiellement très intéressant mais aussi très complexe. Dans tous les cas, la qualité de la traduction reste pour l'instant largement en deçà de l'attente de la majorité des usagers potentiels.

1.2 L'analyse linguistique

De façon schématique, on peut considérer que l'analyse linguistique d'un texte comprend les opérations qui suivent.

1. Découper le texte en phrases, et segmenter chacune des phrases en séquences d'unités lexicales (mots, expression, etc.).
2. Déterminer comment ces unités lexicales s'articulent les unes avec les autres pour former des groupes syntaxiques de niveau supérieur, les syntagmes.
3. Reconnaître les rapports fonctionnels entre syntagmes qui déterminent la structure sémantique (structure de prédicats / arguments) de chaque phrase.
4. Interpréter les structures sémantiques par rapport au contexte de l'énoncé et au modèle du discours.

Ces opérations relèvent de niveaux distincts, qui correspondent aux composantes traditionnelles d'une grammaire, soit la *morphologie* (qui traite de l'agencement des morphèmes), la *syntaxe* (qui traite de l'agencement des mots), la *sémantique* (qui traite de la signification des mots et des règles qui déterminent le sens littéral de la phrase à partir du sens de ses éléments constitutifs) et la *pragmatique* (qui détermine le sens particulier d'un énoncé en fonction des données contextuelles, telles que rapports avec les éventuels énoncés précédents, connaissances extra-linguistiques, etc.).

Un premier handicap pour le développement d'analyseurs linguistiques de qualité tient à l'insuffisance des théories linguistiques actuelles, principalement dans les domaines de la sémantique et de la pragmatique, et de la difficulté de mise en œuvre informatique des théories bien établies comme la syntaxe.

Une autre difficulté vient du fait que les différents niveaux d'analyse linguistique ne sont pas ordonnés de façon strictement linéaire, mais interagissent de façon complexe et encore mal comprise. Il existe un ordre intrinsèque dans la mesure où, par exemple, l'analyse sémantique s'articule sur les structures syntaxiques et ne saurait par conséquent précéder l'analyse syntaxique. Mais il ne s'agit là que d'un ordre partiel. En effet, un grand nombre d'ambiguïtés syntaxiques ne peuvent être résolues que sur la base de critères sémantiques, voire pragmatiques.

L'analyse syntaxique automatique des langues naturelles est une étape fondamentale dans le processus d'analyse automatique du langage, puisque c'est à elle qu'incombe la tâche cruciale de déterminer les structures syntaxiques des phrases. Ce sont ces structures, en effet, qui vont permettre ensuite de calculer les diverses interprétations sémantiques et pragmatiques. Une erreur au niveau du découpage des syntagmes ou un mauvais choix lexical de la part de l'analyseur syntaxique sera propagée aux autres niveaux d'analyse et de traitement.

Le problème de l'analyse syntaxique automatique a fait l'objet d'une attention toute particulière depuis les premières recherches sur le traitement automatique des langues naturelles, et tout particulièrement au cours des dix ou quinze dernières années. Des progrès importants ont été accomplis, mais de nombreux problèmes subsistent, de telle sorte qu'à l'heure actuelle on ne dispose pas de modèles d'analyse suffisamment puissants pour satisfaire aux exigences d'applications à large échelle comme la traduction automatique.

1.3 L'ambiguïté des langues naturelles

Le problème le plus fondamental sur lequel butent les analyseurs syntaxiques, et de façon plus générale tous les systèmes de traitement du langage, est celui de l'ambiguïté des langues naturelles. Même si les locuteurs d'une langue en sont rarement conscients, pratiquement n'importe quelle phrase fourmille d'ambiguïtés, que ce soit au niveau lexical, syntaxique, sémantique ou pragmatique.

1.3.1 Les ambiguïtés lexicales

En parcourant une phrase, un analyseur se trouve fréquemment confronté à des mots qui ont plus d'un sens, on parle alors de *polysémie*. Par exemple, *grève* signifie *arrêt de travail* ou *rivage*.

Si l'analyseur ne cherche qu'à construire une représentation syntaxique de la phrase, il peut choisir d'ignorer les cas de polysémies qui n'entraînent pas de distinction syntaxiques, comme dans l'exemple précédent. Toutefois, il arrive que les différents sens associés à un élément lexical s'accompagnent de traits morphologiques ou syntaxiques distincts. Par exemple, le mot *casse* peut être un substantif féminin (action de casser) ou un substantif masculin (cambriolage, ou casier contenant les caractères d'imprimerie).

Dans de tels cas, l'analyseur syntaxique ne peut ignorer l'ambiguïté. Il en est de même des nombreux cas où un mot appartient à plus d'une catégorie lexicale. Par exemple, *ferme* peut être un adjectif, un verbe ou un substantif.

1.3.2 Les ambiguïtés de structure

On appelle *ambiguïtés de structure* ou *ambiguïtés syntaxiques* les cas d'ambiguïté qui donnent lieu à des représentations syntaxiques différentes. Autrement dit, une phrase présente une ambiguïté structurelle si on peut lui associer deux structures ou plus.

Prenons, par exemple, la phrase :

Jean a vu l'homme sur la colline avec un télescope.

Cette phrase illustre un cas classique d'ambiguïté structurale qui est celui de l'attachement des syntagmes prépositionnels. Le syntagme prépositionnel *avec un télescope* peut être interprété comme se rapportant :

1. au mot *colline*, spécifiant ainsi que c'est sur la colline et avec un télescope que l'homme à été vu par Jean ;
2. au mot *homme*, précisant que, parmi tous les hommes sur la colline, c'est celui avec un télescope que Jean a vu ;
3. au verbe *voir*, spécifiant la façon qui a permis à Jean de voir l'homme sur la colline.

1.3.3 Les ambiguïtés sémantiques

La phrase suivante illustre un cas d'ambiguïté sémantique :

Tous les étudiants de cette école parlent deux langues.

L'ambiguïté de cette phrase est liée à l'interprétation des quantificateurs *tous* et *deux*. Une première interprétation peut être paraphrasée par « chacun des étudiants de cette école parle deux langues, pas nécessairement les mêmes ». Il existe une deuxième interprétation que l'on pourrait paraphraser par « il existe deux langues telles que chaque étudiant de cette école parle ».

1.3.4 Ambiguïtés locales et ambiguïtés globales

Une ambiguïté peut être locale ou globale. On appelle *ambiguïté locale*, ou *ambiguïté temporaire*, une ambiguïté qui peut être levée rapidement, par exemple grâce à l'examen du ou des mots suivants. Au contraire, une *ambiguïté globale* est une ambiguïté qui ne peut être levée sur un simple examen des mots de la phrase.

1.4 TP : Point sur quelques applications existantes

1.4.1 Objectifs du TP

L'objectif de cette séance de travaux pratiques est que vous fassiez, par vous même, un petit tour d'horizon de l'état d'avancement actuel des outils de traitement automatique des langues.

Utilisez la toile (*world wide web*) pour mener à bien votre recherche. Voici quelques domaines sur lesquels votre recherche peut porter :

- systèmes de communication homme-machine ;
- recherche d'information ;
- vérificateurs et correcteurs orthographiques ;
- dictée vocale ;
- synthèse de la parole ;
- traduction automatique ;
- résumé automatique.

1.4.2 Traduction automatique

Au niveau des outils de traduction automatiques, voici deux sites intéressants :

`http://www.reverso.net/textonly/default.asp`
`http://babelfish.altavista.com/`

Tentez quelques traductions avec ces deux traducteurs. Tentez des cycles du style *français* $\longrightarrow$ *anglais* $\longrightarrow$ *français* ou *français* $\longrightarrow$ *allemand* $\longrightarrow$ *français*. Observez l'altération du texte original, pensez à essayer d'effectuer la deuxième traduction avec un outil de traduction différent de celui utilisé pour la première.

1.4.3 Logiciels de discussion

Voici deux sites à visiter :

`http://www.agentland.fr/`
`http://www.pandorabots.com/pandora/talk?botid=f5d922d97e345aa1`

1.4.4 Logiciels de synthèse de la parole

Voici deux sites à visiter :

`http://www.icp.inpg.fr/ICP/`
`http://www.lpl.univ-aix.fr/~roy/cgi-bin/metlpl.cgi`

Cours 2

Langages formels et leurs représentations

Un langage est un ensemble de chaînes (séquences) de caractères ou mots. Puisqu'il n'y a pas de borne sur la longueur des séquences, le nombre total des séquences dans un langage peut être infini. Le but d'une grammaire est de donner une description finie qui spécifie quelles séquences font partie du langage.

2.1 Langages formels

2.1.1 Alphabet, mots, phrases, langages

Définition 2.1 -alphabet, symboles- *Un* alphabet *est un ensemble fini non vide dont les éléments sont appelés* lettres, symboles, mots *ou encore* vocabulaire *L'alphabet sera noté* Σ.

Il peut paraître surprenant que les termes d'alphabet et de vocabulaire puissent être utilisés pour faire référence à un même objet. Cela s'explique simplement par le fait que dans les langages formels un mot est souvent représenté par un caractère alphabétique.

Définition 2.2 -mot, phrase- *Un* mot *est une suite finie d'éléments de l'alphabet. Si l'on utilise le terme « mot » pour désigner les éléments de l'alphabets on utilisera le terme* phrase *pour désigner une suite finie d'éléments de l'alphabet. L'ensemble des mots formés d'éléments de* Σ *est noté* Σ^*. *Le mot composé de* 0 *symboles est appelé le mot vide et est noté* ε.

Pour éviter toute ambiguïté, nous essaierons de ne pas utiliser le terme de « mot » pour désigner un élément de l'alphabet : nous n'utiliserons le terme de mot que pour désigner une suite finie d'éléments de l'alphabet.

Définition 2.3 -langage formel- *On appelle* langage formel *sur* Σ *tout sous-ensemble de* Σ^*.

L'ensemble vide, noté $\varnothing$, est donc un langage formel, et ce, sur n'importe quel alphabet.

2.1.2 Opérations sur les langages

Définition 2.4 -concaténation de mots- *L'opération de* concaténation *consiste à placer deux mots A et B côte à côte pour former le mot noté $A \wedge B$ constitué de la suite des symboles de A suivie de la suite des symboles de B.*

Définition 2.5 -union de langage- *Soit deux langages L_1 et L_2, on appelle* union *de L_1 et L_2 et on note $L_1 \cup L_2$ le langage défini par :*

$$L_1 \cup L_2 = \{x : x \in L_1 \ ou \ \ x \in L_2\}$$

Définition 2.6 -concaténation de langage- *Soit deux langages L_1 et L_2, on appelle* concaténation *de L_1 et L_2 et on note $L_1.L_2$ le langage défini par :*

$$L_1.L_2 = \{x \wedge y : x \in L_1 \ et \ \ y \in L_2\}$$

Définition 2.7 -fermeture de Kleene- *La* fermeture de Kleene[1]*, ou* fermeture itérative*, d'un langage L est notée L^* et est définie par :*

$$L^* = \{x; \exists n \geq 0 : x = x_1 \wedge x_2 \ldots x_n \ et \ \ x_1,x_2,\ldots,x_n \in L\}$$

2.1.3 Langages réguliers

Définition 2.8 -Langage régulier- *Les* langages réguliers *sur Σ sont les langages définis de la manière suivante :*

- *$\varnothing$ et $\{\varepsilon\}$ sont des langages réguliers ;*
- *pour tout symbole $x \in \Sigma$, $\{x\}$ est un langage régulier ;*
- *si A et B sont des langages réguliers, alors : $A \cup B$, $A.B$ et A^* sont des langages réguliers.*

Remarque : tout langage fini est régulier.

2.1.4 Appartenance et représentation

Les deux questions fondamentales qui se présentent dans l'étude des langages formels sont d'une part celle de l'*appartenance* et d'autre part celle de la *représentation* des langages. Le problème de l'appartenance est celui de déterminer, pour un langage et un mot donnés, si le mot appartient ou non au langage. Le problème de la représentation concerne la spécification de l'ensemble des phrases d'un langage.

Dans le cas d'un langage de taille finie, il est toujours possible, en principe, de donner la liste exhaustive de tous les mots qu'il comporte. La question de l'appartenance est alors facile à résoudre, puisque l'on peut parcourir la liste des mots du langage et voir si le mot qui nous intéresse y figure. De même, la représentation du langage peut se faire par extension, en donnant la liste exhaustive des phrases du langage.

Bien sûr, il y a des cas où il n'est pas possible, ou peu désirable, d'énumérer toutes les phrases d'un langage. Dans le cas de langages non finis, l'énumération de toutes les

1. Steven C. Kleene est l'auteur du premier article décrivant l'algèbre des expressions régulières.

phrases est évidemment impossible. C'est donc pour ces langages que se posent avec le plus d'acuité les problèmes d'appartenance et de représentation. Disposer d'une représentation finie de ces langages est une condition nécessaire (mais pas suffisante) pour résoudre le problème de l'appartenance.

Il y a plusieurs façons de définir par des moyens finis un langage non fini. Les méthodes les plus courantes sont les expressions régulières, les automates et les grammaires. Une expression régulière est une expression qui sert à désigner un langage régulier. Un automate est un mécanisme abstrait capable de déterminer, pour un langage donné, si un mot donné appartient ou non à ce langage. Une grammaire est un système génératif basé sur un ensemble fini de règles de dérivation, qui permet d'engendrer de façon systématique toutes les phrases d'un langage.

2.2 Expressions régulières

Une expression régulière est une expression qui sert à désigner un langage régulier.

Définition 2.9 -expression régulière- *Les expression régulières sur Σ sont formées de la manière suivante :*

- *$\varnothing$, ε et toute les lettres de Σ sont des expressions régulières ;*
- *si α et β sont des expressions régulières, alors l'*union $(\alpha + \beta)$*, la* concaténation $(\alpha\beta)$*, la* fermeture $(\alpha)^*$ *sont des expressions régulières.*

On éliminera les parenthèses inutiles. On convient également d'un ordre de préséance pour les trois opérateurs union, concaténation et fermeture :

1. la fermeture est l'opérateur de plus forte priorité,
2. vient ensuite la concaténation,
3. vient enfin l'union (la plus faible priorité).

L'association entre un langage régulier et l'expression régulière qui le dénote se fait de la manière qui suit.

Soit $L(E)$ le langage décrit par l'expression régulière E :

- $L(\varnothing) = \varnothing$; $L(\varepsilon) = \{\varepsilon\}$;
- $L(x) = \{x\}$ pour toute lettre x de Σ ;
- $L((\alpha + \beta)) = L(\alpha) \cup L(\beta)$;
- $L((\alpha\beta)) = L(\alpha).L(\beta)$;
- $L((\alpha)^*) = L(\alpha)^*$.

Exemple : une expression régulière pour l'ensemble des mots sur $\{a,b,c\}$ qui contiennent au moins une fois quatre a consécutif est : $(a + b + c)^*aaaa(a + b + c)^*$.

Théorème 2.1 – *Un langage est régulier si et seulement si il est dénoté par une expression régulière.*

2.3 Automates

Un *reconnaisseur* ou *automate* est un mécanisme abstrait capable de reconnaître les phrases d'un langage, c'est-à-dire de déterminer, pour un langage L et une phrase w donnés, si la phrase w appartient ou non au langage L.

Les automates les plus simples, appelés automates à états finis, ou simplement automates finis, sont des reconnaisseurs pour les langages réguliers. Ils sont formellement définis de la manière suivante.

Définition 2.10 -automate fini- *Un* automate à états finis *(AEF) ou* automates finis *est défini par :*

- *Un* ***ensemble fini*** E ***d'états****,* $E = \{e_0,e_1,\ldots,e_n,\varnothing\}$*. A chaque moment dans le processus de reconnaissance, l'automate se trouve dans un état donné. L'état* $\varnothing$*, aussi appelé état trappe représente l'état dans lequel se trouve l'automate en cas de transition illicite.*
- *Un état* $e_0 \in E$ *distingué comme étant l'****état initial****. C'est l'état dans lequel se trouve l'automate au début du processus de reconnaissance.*
- *un ensemble fini* $F \subset E$ *d'états distingués comme* ***états finaux*** *(ou* ***états terminaux****).*
- *Un alphabet* Σ *des* ***symboles d'entrée****.*
- *Une* ***fonction de transition*** Δ *qui à tout couple formé d'un état de* E *et d'un symbole de* Σ *fait correspondre un ensemble (éventuellement vide) d'états :* $\Delta(e_i,a) = \{e_{i_1},\ldots,e_{i_n}\}$

Un mot est dit reconnue par un automate donné s'il existe une séquence de transitions qui permet à cet automate, à partir de l'état initial, d'avancer du premier au dernier symbole du mot et de se trouver alors dans un état final. Une phrase qui n'est pas reconnue est dite rejetée. L'ensemble des phrases reconnues par un automate définit le langage reconnu par cet automate.

Il est commode de représenter l'ensemble des transitions Δ au moyen d'une table à double entrée, appelée table de transitions, avec sur un axe les états du système et sur l'autre axe les symboles (cf. exemple présenté figure 2.1).

	a	b
e_0	e_1	$\varnothing$
e_1	$\varnothing$	e_0

FIG. 2.1 – *Exemple de table de transitions*

Il existe un autre type de représentation, appelé graphe, ou réseau de transitions, qui permet de représenter graphiquement l'ensemble des opérations licites d'un automate. Dans un réseau de transitions, les états sont dénotés par des cercles contenant le nom de l'état, et les transitions par des arcs orientés qui relient deux états et qui portent une étiquette correspondant à un mot du langage. Une petite flèche signale l'état initial e_0 et un double cercle permet de distinguer les états finaux (cf. exemple présenté figure 2.2).

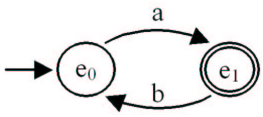


FIG. 2.2 – *Réseau de transitions pour le langage $a(ab)^*$.*

2.4 TD : langages formels, automates, expressions régulières

2.4.1 Alphabet, mots, phrases, langages

2.4.1.1 Alphabet

Proposez un alphabet qui devrait permettre d'écrire des mots correspondants à :

1. des nombres entiers ;
2. des nombres binaires ;
3. les mots de la langue française.

2.4.1.2 Mots

Soit les alphabets suivant :

- $\Sigma_1 = \{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z\}$;
- $\Sigma_2 = \{0,1\}$;
- $\Sigma_3 = \{p,0,1,\neg,\vee,\wedge\}$;
- $\Sigma_4 = \{\clubsuit,\spadesuit,✠\}$;

Dites à partir de quel alphabet(s) peut on générer un langage contenant les mots suivants :

4. 00000111001000100 11 ;
5. aujourdhuicestunbeaujourcarcestledeuxiemetp ;
6. pppppppppp ;
7. aujourdhuicestunbeaujourcarcestledeuxiemetpinfz24 ;
8. $\clubsuit\clubsuit\clubsuit$;
9. $\neg p \vee 0 \wedge \wedge p$;
10. aujourd hui c est un beau jour ;
11. $\clubsuit\spadesuit\clubsuit\clubsuit✠\clubsuit✠✠✠\spadesuit$.

2.4.1.3 Langages

Dites sur quel alphabet de la section précédente sont formés les langages suivants :

12. $\{✠,✠✠,✠✠✠✠,✠✠✠,✠✠✠✠✠\}$;
13. $\{a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z\}$;
14. $\varnothing$;
15. $\{cheval,canal,chacal,bocal,fanal,banal\}$;

16. $\{\varepsilon\}$;
17. ε .

2.4.2 Opérations sur les langages

Soit les langages suivants :
- $L_1 = \{a,b,c\}$;
- $L_2 = \{d,e\}$;
- $L_3 = \{\nabla,\triangle\}$.

18. Quel est le langage résultant de l'union de L_1 et L_2 ?
19. Quel est le langage résultant de l'union de L_2 et L_3 ?
20. Quel est le langage résultant de la concaténation de L_1 et L_2 ?
21. Quel est le langage résultant de la concaténation de L_2 et L_3 ?
22. Donnez 10 mots distincts de la fermeture de Kleene du langage L_3.

2.4.3 Expression régulière

23. Effectuez un parenthèsage correct et complet de l'expression régulière : $a + bc^*d$
24. Ecrivez les expressions régulières qui définissent :
    - les chaînes de 0 et de 1 se terminant par un 0 ;
    - les chaînes de 0 et de 1 contenant au moins un 1 ;
    - les chaînes de 0 et de 1 contenant au plus un 1.
25. Décrire la conjugaison du verbe *prendre* à l'imparfait comme une expression régulières.
26. On admet qu'un système d'exploitation λ impose que le nom des fichiers réponde aux conditions suivantes :
    - ils sont formés sur l'alphabet $\{a,b,c,\ldots,z,0,1,2,\ldots,9,.\}$;
    - ils sont constitué par une chaîne de 1 à 8 caractères qui sont des lettres ou des chiffres, suivi du caractère « . » suivit de 3 caractères qui sont des lettres ou des chiffres ;
    - ils ne peuvent commencer par un chiffre.

    Représentez par une expression régulière la syntaxe de ces noms de fichier.

2.4.4 Automates

27. Concevez le graphe d'un automate à états finis capable de reconnaître si un mots correspond à une conjugaison du verbe *prendre* à l'imparfait.
28. Représentez l'ensemble des transitions de cet automate au moyen d'une table à double entrée, appelée table de transitions, avec sur un axe les états du système et sur l'autre axe les symboles.
29. Donnez la représentation formelle de cet automate comme indiquée dans la définition 2.10.

2.5 Grammaires formelles

Une grammaire est un système formel défini comme un ensemble de règles parfaitement explicites, applicables de façon mécanique, qui transforment une certaine chaîne de symboles, dite chaîne d'entrée, en une autre chaîne de symboles, dite chaîne de sortie.

Définition 2.11 -grammaire formelle- *Nous appellerons grammaire formelle la donnée du quadruplet* (V_N,V_T,S,R) *où :*

- V_N *est un ensemble fini et non vide de symboles appelés* ***symboles non-terminaux*** *qui sont utilisés dans les étapes intermédiaires de la dérivation des phrases ;*
- V_T *est un ensemble fini de symboles appelés* ***symboles terminaux*** *;*
- $S \in V_N$ *est le symbole initial de la grammaire appelé* ***axiome*** *;*
- R *est un ensemble fini de couple* (φ,ψ) *tels que* $\varphi \in (V_N \cup V_T)^* - \varepsilon$, *et* $\psi \in (V_N \cup V_T)^*$, *en général notés* $\varphi \longrightarrow \psi$ *(lire* φ *se réécrit* ψ*) et appelés* ***règles de réécriture*** *ou* ***règles de production****.*

Les ensemble V_N et V_T sont disjoints. Ensemble, ils forment le vocabulaire Σ de la grammaire.

Définition 2.12 -dérivation immédiate- *Etant donnée une grammaire* G *de vocabulaire non-terminal* V_N, *de vocabulaire terminal* V_T *et d'ensemble de règles* R, *et deux mots* w_1 *et* w_2 *tels que* $w_1 \in (V_N \cup V_T)^* - \{\varepsilon\}$ *et* $w_2 \in (V_N \cup V_T)^*$, *on dit que* w_1 *permet de* dériver immédiatement w_2 *dans* G, *et on écrit* $w_1 \vdash_G w_2$, *si et seulement si il existe des mots* α, β, u_1 *et* u_2 *(* $u_1 \in (V_N \cup V_T)^* - \{\varepsilon\}$, $\alpha,\beta,u_2 \in (V_N \cup V_T)^*$ *) tels que :*

- $w_1 = \alpha u_1 \beta$
- $w_2 = \alpha u_2 \beta$
- $(u_1 \longrightarrow u_2) \in R$

Définition 2.13 -dérivation- *Dans les mêmes conditions que précédemment, on dira qu'un mot* w_1 permet de dériver *un mot* w_2, *et on écrira* $w_1 \vdash_G^* w_2$, *si et seulement si il existe une suite de mots* $u_0,u_1,\ldots,u_n$ *telle que :*

- $u_0 = w_1$
- $u_n = w_2$
- *pour tout* $i = 0 \ldots n-1$, $u_i \vdash_G u_{i+1}$

On appelle dérivation *toute suite d'expressions* $u_0,u_1,\ldots,u_n$ *satisfaisant la dernière contrainte (pour tout* $i = 0 \ldots n-1$, $u_i \vdash_G u_{i+1}$*).*

Définition 2.14 -langage engendré par une grammaire- *Etant donnée une grammaire* G *sur* $V_N \cup V_T$, *on appelle* langage engendré *par* G, *et on note* $L(G)$ *le langage :* $L(G) = \{\sigma \in V_T; S \vdash_G^* \sigma\}$

2.6 Types de règles, types de langages

2.6.1 Introduction

La définition des grammaires formelles que nous venons de donner est extrêmement générale puisque l'on peut montrer qu'elle permet de caractériser tous les langages récur-

sivement énumérables, c'est-à-dire tous les langages pour lesquels il existe une procédure capable d'énumérer les phrases de ce langage.

La classe des langages récursivement énumérables est beaucoup trop vaste, et surtout trop peu contrainte, pour constituer un modèle intéressant pour la plupart des applications basées sur la théorie des langages. En particulier, il faut noter que le problème des l'appartenance n'a pas toujours de solution pour les langages récursivement énumérables. Cela signifie que l'on ne peut garantir l'existence d'un algorithme capable de déterminer si une chaîne donnée fait partie ou non du langage.

D'un point de vue linguistique, la classe des langages récursivement énumérables ne présente également qu'un intérêt très limité. On s'attend, en effet, à ce que le langage naturel ait un certain nombre de propriétés spécifiques limitant la variété des structures syntaxiques possibles. Comme la classe des langages récursivement énumérables constitue le modèle le moins contraint (le plus faible), elle n'est guère attractive. Les langages récursifs, qui constituent un sous-ensemble propre des langages récursivement énumérables, sont de ce point de vue beaucoup plus intéressants, puisque l'on peut garantir l'existence d'un algorithme capable de décider en un nombre fini d'étapes si une chaîne donnée fait partie ou non du langage.

2.6.2 Type de grammaires

Définition 2.15 -grammaire de type 1- *Une grammaire G est dite* de type 1*, ou contextuelle, si et seulement si pour toute règle $\phi \longrightarrow \psi$ qui lui appartient, on a $|\phi| \leq |\psi|$ où $|\chi|$ désigne la longueur du mot χ en nombre de symboles.*

Cette condition exige que, pour chaque règle de production, la partie droite de la règle soit de longueur supérieure ou égale à la partie gauche. On dira que la grammaire ne peut contenir de règle de production *raccourcissante*, ou ne peut contenir que des règles *non-raccourcissantes.*

Définition 2.16 -grammaire de type 2- *Une grammaire est dite* de type 2*, ou* hors-contexte*, ou encore* indépendante du contexte *(*context-free *en anglais), si et seulement si elle est de type 1 avec de plus la partie gauche de toute règle réduite à un seul symbole non terminal. Autrement dit, les règles sont de la forme : $\chi \longrightarrow \psi$ avec $\chi \in V_N$ et $\psi \in (V_N \cup V_T)^*$*

Définition 2.17 -grammaire de type 3- *Une grammaire G est dite* de type 3*, ou* linéaire*, ou encore* régulière*, si et seulement si elle est linéaire à gauche ou à droite. Une grammaire G est dite* linéaire à gauche *(respectivement* linéaire à droite*) si et seulement si elle est de type 2 et de plus que toutes ses règles sont de la forme : $X \longrightarrow xY$ ou $X \longrightarrow x$ avec X,$Y \in V_N$ et $x \in V_T$ (respectivement : $X \longrightarrow Yx$ ou $X \longrightarrow x$ avec X,$Y \in V_N$ et $x \in V_T$).*

Définition 2.18 -grammaire de type 0- *Le* type 0 *sera accordé aux grammaires « en général », c'est-à-dire sans restriction particulière sur la forme des règles.*

Les grammaires de type 1, 2, 3 ne permettent pas d'engendrer le mot vide (ε). Les langages qu'elles engendrent ne contiennent donc pas le mot vide (puisqu'engendrer celui-

ci supposerait qu'on ait au moins une règle raccourcissante, celle qui réécrit quelque chose en rien). Nous définissons donc les grammaires de types étendus 1, 2, 3.

Définition 2.19 -grammaires de type étendu- *Les grammaires de types étendus 1, 2, 3 sont les grammaires de types respectifs 1, 2, 3 auxquelles on rajoute la possibilité de règles ayant ε en partie droite.*

Etant donné le caractère de plus en plus restrictif de ces conditions sur la forme des règles de grammaires, on notera que les grammaires de type 3 constituent un sous-ensemble des grammaires de type 2 qui elles-mêmes constituent un sous-ensemble des grammaires de type 1 qui elles-mêmes constituent un sous-ensemble des grammaires de type 0.

2.6.3 La hiérarchie des langages

La classification des grammaires détermine une classification des langages que ces dernières engendrent. Ainsi, on appellera langage de type 3 un langage engendré par une grammaire de type 3, langage de type 2 un langage engendré par une grammaire de type 2, etc. La hiérarchie des langages, appelée *hiérarchie de Chomsky*, ainsi déterminée est illustrée figure 2.3.

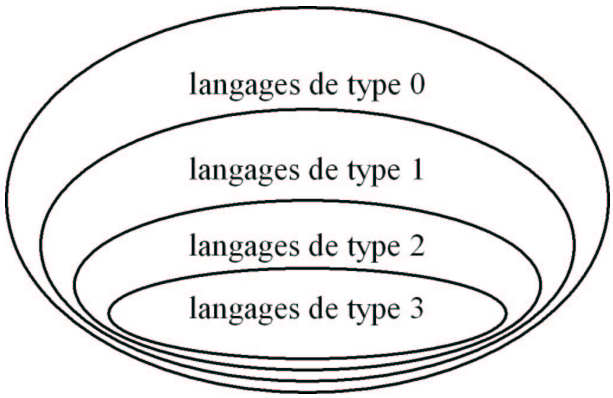


FIG. 2.3 – *Hiérarchie des langages.*

Il faut remarquer, cependant, que l'on ne peut exclure la possibilité qu'un langage engendré par une grammaire de type 2 soit en fait un langage de type 3. Pour établir l'appartenance strict à un langage de type 2 (donc sans inclure les langages de type 3), il faut montrer d'une part qu'il existe une grammaire de type 2 capable d'engendrer ce langage, et d'autre part qu'il n'existe aucune grammaire plus contrainte capable de l'engendrer.

La hiérarchie de langage ainsi établie correspond également à une hiérarchie de complexité. En effet, il est plus facile de déterminer si un mot w appartient à un langage de type 3 qu'à un langage de type 2 et qu'il appartient à un langage de type 2 qu'à un langage de type 1.

2.7 Structures engendrées par les dérivations

2.7.1 Arbres de dérivations

Pour les grammaires de type 2 et de type 3, les dérivations déterminent des structures arborescentes dans lesquelles :

- l'axiome de la grammaire correspond à la racine de l'arbre ;
- les éléments non-terminaux de la grammaire correspondent à des nœuds de l'arbre ;
- les éléments terminaux, c'est à dire les symboles du mot, correspondent aux feuilles (extrémités) de l'arbre.

A partir du nœud racine S, on construit l'arbre dérivationnel comme suit : à chaque pas de la dérivation, la réécriture d'un symbole non terminal revient à attacher au nœud correspondant à ce symbole autant de sous-arbres qu'il y a de symboles dans la partie droite de la règle utilisée.

2.7.1.1 Exemple de dérivation

Soit la grammaire suivante :

- $V_N = \{P, SN, SV, Det, N\}$
- $V_T = \{le, la, chat, mange\}$
- Axiome : P
- Règles de réécriture :
  $S \longrightarrow SN\ SV$
  $SN \longrightarrow Det\ N$
  $SV \longrightarrow V\ SN$
  $Det \longrightarrow le$
  $Det \longrightarrow la$
  $N \longrightarrow chat$
  $V \longrightarrow mange$

Soit la dérivation suivante :

$$\begin{aligned} P &\longrightarrow SN\ SV \\ &\longrightarrow Det\ N\ SV \\ &\longrightarrow le\ N\ SV \\ &\longrightarrow le\ chat\ SV \\ &\longrightarrow le\ chat\ V\ SN \\ &\longrightarrow le\ chat\ mange\ SN \\ &\longrightarrow le\ chat\ mange\ Det\ N \\ &\longrightarrow le\ chat\ mange\ la\ N \\ &\longrightarrow le\ chat\ mange\ la\ souris \end{aligned}$$

Cet dérivation détermine alors la structure arborescente de la figure 2.4.

2.7.2 Notion d'ambiguïté

Définition 2.20 -dérivation la plus à gauche- *On appelle* dérivation la plus à gauche *une dérivation dans laquelle à chaque pas de dérivation on réécrit le symbole non terminal le plus à gauche du mot.*

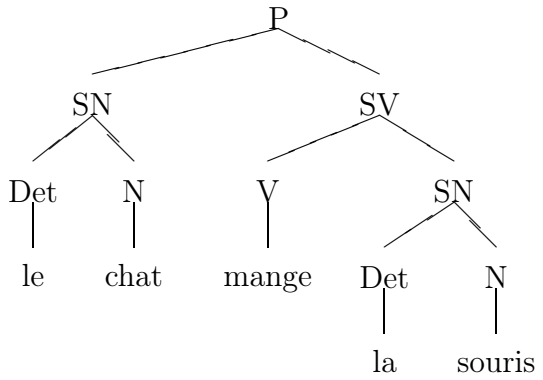


FIG. 2.4 – *Structure arborescente associée à la dérivation.*

Un exemple de dérivation la plus à gauche est donnée dans l'exemple de dérivation de la section précédente (section 2.7.1).

Définition 2.21 -dérivation la plus à droite- *On appelle* dérivation la plus à droite *une dérivation dans laquelle à chaque pas de dérivation on réécrit le symbole non terminal le plus à droite du mot.*

Définition 2.22 -grammaire ambiguë- *On dira qu'une grammaire est une* grammaire ambiguë *si le langage engendré par cette grammaire contient au moins un mot pour lequel il existe plus d'une dérivation la plus à droite.*

Définition 2.23 -langage ambigu- *Un langage est dit* ambigu *si toutes les grammaires qui l'engendrent sont ambiguës.*

2.7.2.1 Exemple de grammaire ambiguë

Soit la grammaire suivante :
- $V_N = \{P,SN,SV,SP,Det,Prep,N\}$
- $V_T = \{l,le,un,avec,homme,chien,télescope,observe\}$
- Axiome : P
- Règles de réécriture :
  $P \longrightarrow SN\ SV$
  $SN \longrightarrow Det\ N$
  $SN \longrightarrow Det\ N\ SP$
  $SP \longrightarrow Prep\ SN$
  $SV \longrightarrow V\ SN$
  $SV \longrightarrow V\ SN\ SP$
  $Det \longrightarrow l$
  $Det \longrightarrow le$
  $Det \longrightarrow un$
  $Prep \longrightarrow avec$
  $N \longrightarrow homme$
  $N \longrightarrow chien$
  $N \longrightarrow télescope$
  $V \longrightarrow observe$

Le mot « *l'homme observe le chien avec un télescope* » peut être obtenue à partir de deux dérivations les plus à gauche distinctes.

Ces dérivations déterminent les deux structures arborescentes représentées dans les figures 2.5 et 2.6.

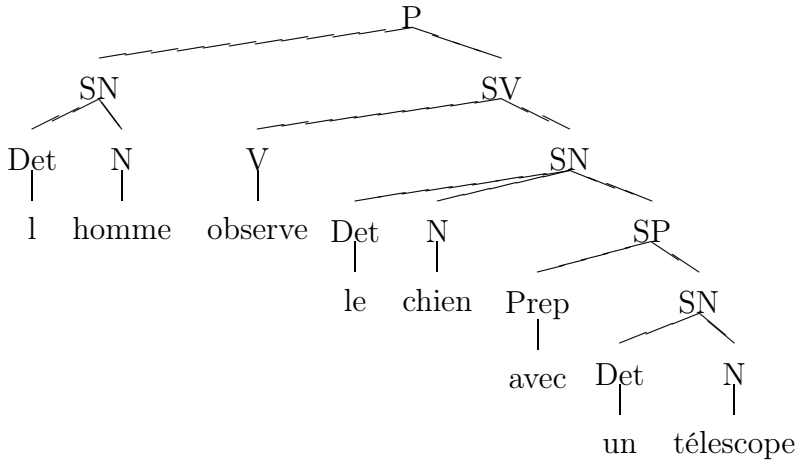


FIG. 2.5 – *Structure arborescente associée à la 1re possibilité de dérivation.*

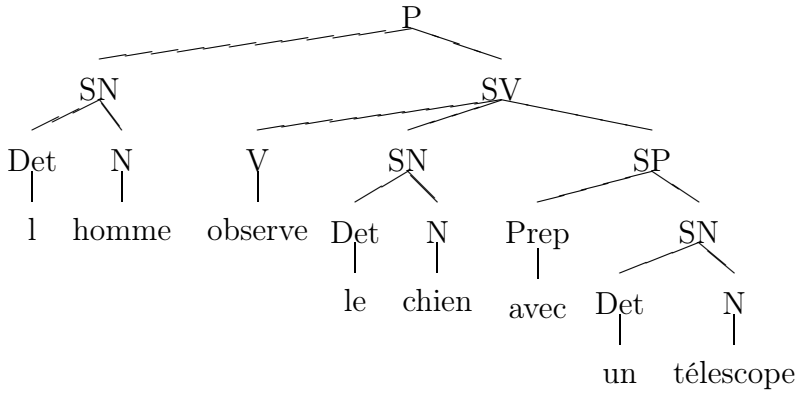


FIG. 2.6 – *Structure arborescente associée à la 2e possibilité de dérivation.*

2.7.3 Capacité générative et équivalence de grammaires

Définition 2.24 **-capacité générative faible-** *On parle de la* capacité générative faible *d'une grammaire lorsque l'on fait référence à l'ensemble des phrases engendrée par cette grammaire, sans s'intéresser à la façon dont ces phrases ont été dérivées.*

Définition 2.25 **-capacité générative forte-** *La* capacité générative forte *d'une grammaire correspond à l'ensemble des phrases et des structures associées engendrées par une grammaire.*

Sur la base de cette distinction, on peut définir les notions d'*équivalence faible* et d'*équivalence forte*.

Définition 2.26 -équivalence faible- *On dit que deux grammaires sont* faiblement équivalentes *si elles engendrent le même langage.*

Définition 2.27 -équivalence forte- *On dit que deux grammaires sont* fortement équivalentes *si elles engendrent le même langage et les mêmes structures.*

L'équivalence forte implique donc l'équivalence faible, mais non l'inverse.

2.8 TD : grammaires

2.8.1 Grammaire de type 3

La grammaire suivante est de type 3 :
- $V_N = \{S,Q,B\}$;
- $V_T = \{a,b\}$;
- axiome : S ;
- règles de réécritures :

$$S \longrightarrow aQ$$
$$Q \longrightarrow aQ$$
$$Q \longrightarrow bB$$
$$B \longrightarrow b$$

1. Exercez-vous à faire des dérivations de cette grammaire.
2. Dites si les mots suivants appartiennent au langage engendré par cette grammaire, si oui donnez la structure arborescente associée à leur dérivation :
   - $baabab$;
   - ab ;
   - $aaabb$;
   - $aaab$.
3. Après la suite de symboles a consécutifs on désire pouvoir avoir une suite de plus de 2 symboles b consécutif. Ajoutez une règle qui permette cela.
4. Comment faire pour que la suite de symboles b puisse se réduire à un seul b?

2.8.2 Grammaire de type 2

Ecrivez une grammaire capable de générer des mots contenant autant de symbole a que de symbole b de la forme suivante :

$$\underbrace{a \ldots a}_{1 \text{ à n } a}\underbrace{b \ldots b}_{1 \text{ à n } b}$$

2.8.3 Grammaire de type 1

La grammaire suivante est de type 1 car toutes ces règles sont non-raccourcissantes :

- $V_N = \{S,B,C\}$;
- $V_T = \{a,b,c\}$;
- axiome : S ;
- règles de réécritures :

$$S \longrightarrow aSBC$$
$$S \longrightarrow abC$$
$$CB \longrightarrow BC$$
$$bB \longrightarrow bb$$
$$bC \longrightarrow bc$$
$$cC \longrightarrow cc$$

1. Exercez-vous à faire des dérivations de cette grammaire.
2. Dites si les mots suivants appartiennent au langage engendré par cette grammaire :
   - $baabab$;
   - abc ;
   - $aaabbbccc$;
   - $abbccc$;
   - $aabcc$.

Cours 3

Langages réguliers (type 3)

3.1 Extensions Unix pour les expressions régulières

Le système Unix dispose de plusieurs commandes qui utilisent une notation dérivée des expressions régulières pour désigner des ensembles de chaînes de caractères.

La différence majeure entre une expression régulière classique et une expression régulière sous Unix est qu'une expression sous Unix est généralement interprétée comme décrivant toutes les chaînes qui contiennent une sous-chaîne décrite par l'expression régulière.

Par exemple, l'expression régulière classique *rena* ne décrit que le mot « rena » tandis que cette même expression interprétée sous Unix décrit toutes les chaînes contenant des occurrences comme «... j'app**rena**is ...», «... comp**rena**it ...», «... **rena**rd ...», etc. Comme nous le verrons plus loin, l'équivalent sous Unix de l'expression régulière classique *rena* sera ^*rena*$.

Dans ce qui suit, *<exp_reg>*, *<exp_reg_1>*, *<exp_reg_2>* désignent des expressions régulières.

<caractère> un caractère est une expression régulière qui désigne lui-même, excepté pour les caractères `.`, `?`, `+`, `*`, `{`, `|`, `(`, `)`, `^`, `$`, `\`, `[`, `]` qui sont des méta-caractères et ont une signification spéciale ; pour désigner ces méta-caractères, il faut les faire précéder d'un antislash (`\.`, `\?`, `\+`, `\*`, `\{`, `\|`, `\(`, `\)`, `\^`, `\$`, `\\`, `\[`, `\]`).

`[`***<liste_de_caractères>***`]` est une expression régulière qui décrit l'un des caractères de la liste de caractères, par exemple `[abcdf]` décrit le caractère `a`, le `b`, le `c`, le `d` ou le `f` ; le caractère `-` permet de décrire des ensembles de caractères consécutifs, par exemple `[a-df]` ≡ `[abcdf]` ; la plupart des méta-caractères perdent leur signification spéciale dans une liste, pour insérer un `]` dans une liste, il faut le mettre en tête de liste, pour inclure un `^`, il faut le mettre n'importe où sauf en tête de liste, enfin un `-` se place à la fin de la liste.

`[^`***<liste_de_caractères>***`]` est une expression régulière qui décrit les caractères qui ne sont pas dans la liste de caractères.

`[:alnum:]` à l'intérieur d'une liste, décrit un caractère alpha-numérique (`[[:alnum:]]` ≡ `[0-9A-Za-z]`) ; sur le même principe, on a également `[:alpha:]`, `[:cntrl:]`, `[:digit:]`, `[:graph:]`, `[:lower:]`, `[:print:]`, `[:punct:]`, `[:space:]`, `[:upper:]` et `[:xdigit:]`.

`.` est une expression régulière et un méta-caractère qui désigne n'importe quel caractère.

`\w` est une expression régulière qui est synonyme de `[[:alnum:]]`.

`\W` est une expression régulière qui est synonyme de `[^[:alnum:]]`.

`^` est une expression régulière et un méta-caractère qui désigne le début d'une chaîne de caractères.

`$` est une expression régulière et un méta-caractère qui désigne la fin d'une chaîne de caractères.

`\<` est une expression régulière qui désigne le début d'un mot.

`\>` est une expression régulière qui désigne la fin d'un mot.

<exp_reg>`?` est une expression régulière qui décrit zéro ou une fois *<exp_reg>*.

<exp_reg>`*` est une expression régulière qui décrit *<exp_reg>* un nombre quelconque de fois, zéro compris.

<exp_reg>`+` est une expression régulière qui décrit *<exp_reg>* au moins une fois.

<exp_reg>`{`n`}` est une expression régulière qui décrit *<exp_reg>* n fois.

<exp_reg>`{`n`,}` est une expression régulière qui décrit *<exp_reg>* au moins n fois.

<exp_reg>`{`n`,`m`}` décrit *<exp_reg>* au moins n fois et au plus m fois.

<exp_reg_1><exp_reg_2> est une expression régulière qui décrit une chaîne constituée de la concaténation de deux sous-chaînes respectivement décrites par *<exp_reg_1>* et *<exp_reg_2>*.

<exp_reg_1>`|`*<exp_reg_2>* est une expression régulière qui décrit toute chaîne décrite par *<exp_reg_1>* ou par *<exp_reg_2>*.

`(`*<exp_reg>*`)` est une expression régulière qui décrit ce que décrit *<exp_reg>*.

La concaténation de deux expressions régulières (*<exp_reg_1><exp_reg_2>*) est une opération prioritaire sur l'union (*<exp_reg_1>*`|`*<exp_reg_2>*).

3.2 TD : expressions régulières

3.2.1 Expressions régulières

Pour cet exercice nous restons dans le cadre des expressions régulières classiques décrites section 2.2.

1. Donnez tous les mots décrits par l'expression régulière suivante :

$$(aa + bb)(ba + ab)(aa + bb)$$

2. Donnez tous les mots de longueur 6 décrits par l'expression régulière suivante :

$$(aa + bb)^*$$

3. Donnez tous les mots de longueur inférieure ou égale à 8 décrits par l'expression régulière suivante :

$$(ab + ba)(aa + bb)^*ab$$

Pour les questions 4 à 8 trouvez une expression régulière, toujours de type classique, qui définisse chacun des langages suivants sur l'alphabet $\{a,b\}$.

4. Tous les mots d'exactement deux symboles.
5. Tous les mots qui contiennent 2 ou 3 b, ni plus, ni moins.
6. Tous les mots qui se terminent par deux symboles identiques.
7. Tous les mots qui ne se terminent pas par deux symboles identiques.
8. Tous les mots de longueur paire.
9. Dans le cours, nous avons défini (définition 2.9) les expressions régulières de manière récursive. De la même manière, donnez une définition récursive à la syntaxe des expressions arithmétiques qui portent sur des entiers et qui utilisent les opérateurs $+$, $-$, $/$, $*$, et les parenthèses ().

3.2.2 Parallèle entre expressions régulières et extensions Unix

Les expressions qui suivent ont été écrites en utilisant l'extension Unix des expressions régulières. Réécrivez-les en restant dans le cadre des expressions régulières classiques décrites section 2.2. Nous nous plaçons dans le cadre de l'alphabet $\Sigma = \{a,b,c,d,e,f\}$.

10. $\hat{}de\$$
11. $\hat{}de$
12. $de\$$
13. de
14. $\hat{}[a-df]\$$
15. $\hat{}(abc)?\$$
16. $\hat{}(abc)*\$$
17. $\hat{}(abc)+\$$
18. $\hat{}(abc)\{2\}\$$
19. $\hat{}(abc)\{2,\}\$$
20. $\hat{}(abc)\{2,3\}\$$

3.2.3 Extensions Unix des expressions régulières

21. Donnez une expression régulière qui décrive les mots possibles pour une horloge numérique sur 24 heures où les heures, les minutes et les secondes sont séparées par « : ».
22. On admet qu'un système d'exploitation λ impose que le nom des fichiers réponde aux conditions suivantes :
    - ils sont formés sur l'alphabet $\{a,b,c,\ldots,z,0,1,2,\ldots,9,.\}$;
    - ils sont constitués par une chaîne de 1 à 8 caractères qui sont des lettres ou des chiffres, suivi du caractère « . » suivi de 3 caractères qui sont des lettres ou des chiffres ;
    - ils ne peuvent commencer par un chiffre.

    Représentez par une expression régulière (de type Unix) la syntaxe de ces noms de fichier.

3.3 Automates

3.3.1 Automate finis déterministes et non-déterministe

Les automates tels que nous les avons définis par la définition 2.10 sont non déterministes. Pour un tel automate, il est possible que, pour un état donné et un symbole donné, il y ait plusieurs transitions correspondantes. Il est possible d'étendre cette définition pour que l'automate accepte des transitions sur le mot vide (sans avancer sur le mot d'entrée), et des transitions sur des mots de longueur supérieure à 1.

Evidemment, d'un point de vue strictement informatique, de tels automates posent des problèmes de réalisation et ne sont donc pas souhaitables. Toutefois, comme nous le verrons plus loin, tout automate non-déterministe peut être rendu déterministe. L'intérêt du non-déterminisme est qu'il nous permet souvent d'avoir une description plus simple de certains langages.

Les automates finis déterministes sont plus faciles à simuler (pas de choix dans les transitions, donc jamais de retour en arrière à faire).

Définition 3.1 -automate fini déterministe- *Un* automate à états finis déterministe *(AEFD) est défini par :*

- *Un* ***ensemble fini*** E ***d'états***, $E = \{e_0,e_1,\ldots,e_n,\varnothing\}$. *A chaque moment dans le processus de reconnaissance, l'automate se trouve dans un état donné. L'état* $\varnothing$, *également appelé état trappe, représente l'état dans lequel se trouve l'automate en cas de transition illicite.*
- *Un état* $e_0 \in E$ *distingué comme étant l'****état initial****. C'est l'état dans lequel se trouve l'automate au début du processus de reconnaissance.*
- *un ensemble fini* $F \subset E$ *d'états distingués comme* ***états finaux*** *(ou* ***états terminaux****).*
- *Un alphabet* Σ *des* ***symboles d'entrée****.*
- *Une* ***fonction de transition*** Δ *qui à tout couple formé d'un état de* E *et d'un symbole de* Σ *fait correspondre un état de* E *:* $\Delta(e_i,a) = e_j$

3.3.2 Elimination du non-déterminisme

Il existe des algorithmes permettant de déterminiser un automate non déterministe (c'est à dire de construire un automate fini déterministe qui reconnaît le même langage que l'automate fini non-déterministe donné). L'automate fini déterministe obtenu comporte en général plus d'états que l'automate fini non-déterministe.

Une transition sur le mot vide, ou ε-transition, n'est autre qu'un arc étiqueté par le symbole ε. Un tel arc peut être franchi n'importe quand sans avancer sur le mot d'entrée.

L'opération qui consiste à déterminiser un automate non déterministe est sensiblement plus complexe quand l'automate non déterministe comporte des transitions sur le mot vide. Aussi considérerons-nous tout d'abord le cas où l'automate ne possède pas de transition sur le mot vide.

L'algorithme qui suit permet donc la déterminisation d'un automate non déterministe qui ne contient pas de transition sur le mot vide.

3.3.2.1 Algorithme de déterminisation d'un automate fini non déterministe sans transition sur le mot vide

1. Partir de l'état initial.
2. Rajouter dans la table de transition tous les nouveaux « états » produits, avec leur transition. Par exemple, si pour l'état e_i, toutes les transitions possibles sont $\Delta(e_i,s_j) = \{e_{ij_1},e_{ij_2}\}$ et $\Delta(e_i,s_k) = \{e_{ik_1}\}$, alors les deux « états » produits sont e_{ij_1},e_{ij_2} et e_{ik_1}.
3. Recommencer l'étape 2 jusqu'à ce qu'il n'y ait plus de nouvel « état ».
4. Tous les « états » contenant au moins un état terminal deviennent terminaux.
5. Renuméroter alors les états.

3.3.2.2 Exemple de déterminisation d'un automate fini non déterministe sans transition sur le mot vide

Nous nous proposons de construire un automate fini déterministe qui reconnaît le même langage que l'automate fini non-déterministe de la figure 3.1.

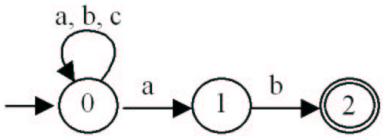


FIG. 3.1 – *Exemple d'automate fini non-déterministe.*

La table de transition de l'automate de la figure 3.1 est la suivante :

	a	b	c
0 (état initial)	0, 1	0	0
1		2	
2 (état final)			

Cet automate est bien non-déterministe puisque lorsqu'on est dans l'état 0 et que l'on rencontre le symbole a on peut passer dans l'état 1 ou rester dans le 0.

Pour générer la table de transition de l'automate fini déterministe, qui reconnaît le même langage que l'automate fini non-déterministe dont nous venons de voir la table de transition, nous appliquons l'algorithme précédent.

Nous partons donc de l'état initial :

	a	b	c
0 (état initial)	0, 1	0	0

On rajoute dans la table de transition tous les nouveaux « états » produits, avec leur transition. Le seul nouvel « état » produit est l'état 0,1. La table de transition devient :

	a	b	c
0 (état initial)	0, 1	0	0
0,1	0, 1	0, 2	0

Nous avons encore produit un nouvel « état », l'état 0,2 :

	a	b	c
0 (état initial)	0, 1	0	0
0,1	0, 1	0, 2	0
0,2	0, 1	0	0

Il n'y a plus de nouvel « état » produit. Tous les « états » contenant au moins un état terminal deviennent terminaux :

	a	b	c
0 (état initial)	0, 1	0	0
0,1	0, 1	0, 2	0
0,2 (état terminal)	0, 1	0	0

Il ne reste plus qu'à renuméroter les états :

	a	b	c
0 (état initial)	1	0	0
1	1	2	0
2 (état terminal)	1	0	0

L'automate fini déterministe produit est représenté figure 3.2.

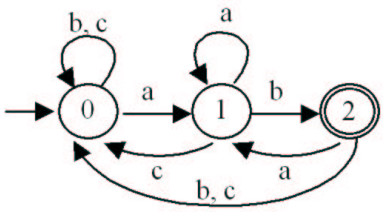


FIG. 3.2 – *Automate fini déterministe produit.*

3.3.2.3 Algorithme de déterminisation d'un automate fini non déterministe avec des transitions sur le mot vide

Dans un automate, si l'on se trouve dans un état s quelconque comportant des ε-transitions, on se trouve en réalité en même temps dans chaque état accessible à partir de s en suivant un chemin d'arcs étiquetés par ε.

L'algorithme est en fait le même que celui donné pour les automates finis non-déterministes sans transition sur le mot vide. La seule différence est qu'ici quand on parle d'un « état » on désigne également les états accessible par des ε-transitions.

1. Partir de l'« état » initial (ie. l'état initial et tous les états accessibles à partir de l'état initial en suivant un chemin d'arcs étiquetés par ε)
2. Rajouter dans la table de transition tous les nouveaux « états » produits, avec leur transition.
3. Recommencer l'étape 2 jusqu'à ce qu'il n'y ait plus de nouvel « état ».
4. Tous les « états » contenant au moins un état terminal deviennent terminaux.
5. Renuméroter alors les états.

3.3.2.4 Exemple de déterminisation d'un automate fini non déterministe avec des transitions sur le mot vide

On se propose de déterminiser l'automate fini non déterministe, comportant des transitions sur le mot vide, de la figure 3.3.

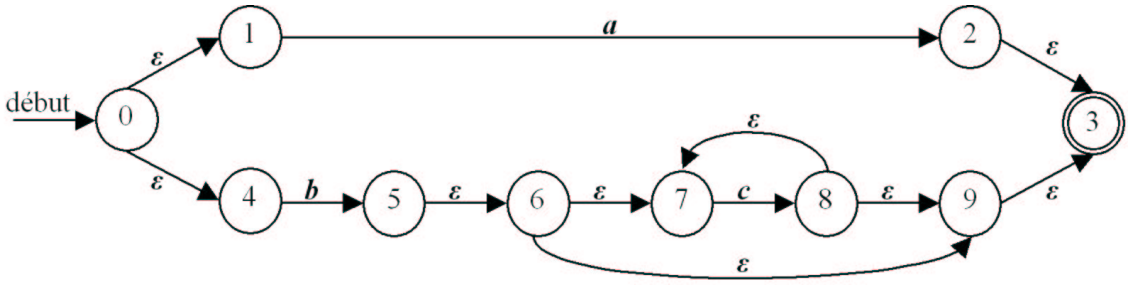


FIG. 3.3 – *Automate fini non déterministe.*

Voici la table de transition construite en utilisant l'algorithme, que nous venons de voir, sur cet automate :

	a	b	c
0,1,4 (état initial)	2,3	3,5,6,7,9	∅
2,3 (état terminal)	∅	∅	∅
3,5,6,7,9 (état terminal)	∅	∅	3,7,8,9
3,7,8,9 (état terminal)	∅	∅	3,7,8,9

Il ne reste plus qu'à renuméroter les états :

	a	b	c
0 (état initial)	1	2	∅
1 (état terminal)	∅	∅	∅
2 (état terminal)	∅	∅	3
3 (état terminal)	∅	∅	3

L'automate fini déterministe produit est représenté figure 3.4.

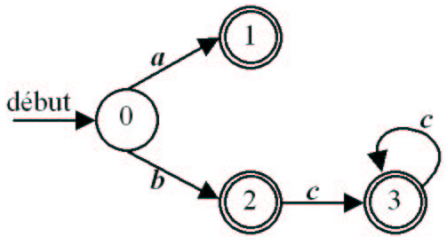


FIG. 3.4 – *Automate fini déterministe produit.*

3.4 Equivalence entre expression régulière et automate fini

3.4.1 Introduction

Théorème 3.1 – *Tout langage accepté par un automate à états finis peut être défini par une expression régulière.*

Théorème 3.2 – *Pour toute expression régulière, il existe un automate à états finis qui accepte le langage qu'elle dénote.*

Il existe une méthode systématique qui permet de construire l'automate fini qui décrit le même langage qu'une expression régulière donnée. Il existe également une méthode systématique qui permet de générer une expression régulière qui dénote le même langage qu'un automate fini donné.

Cependant, générer une expression régulière à partir d'un automate fini de manière systématique est une tâche plus complexe que l'inverse. Aussi, dans ce cours, ne présenterons-nous que la construction d'un automate fini équivalent à une expression régulière donnée.

L'algorithme de passage d'une expression régulière à un automate fini utilise massivement les ε-transitions.

3.4.2 Des expressions régulières aux automates

On peut transformer une expression régulière en automate au moyen d'un algorithme dérivé d'une récurrence complète sur le nombre d'occurrences d'opérateurs dans l'expression régulière.

Cas de base : à $\varnothing$, on peut associer un automate ayant un état initial mais pas d'état final (ie. $F = \varnothing$, cf. figure 3.5) ; à ε, on peut associer une transition vide d'un état initial vers un état final (cf. figure 3.6), ou bien un automate consistant simplement en un seul état à la fois initial et final ; à une lettre $x \in \Sigma$, on peut associer une transition étiquetée x allant d'un état initial vers un état final (cf. figure 3.7).

Schéma inductif : soit M_α et M_β les automates associés à deux expressions régulières α et β.

- Pour $\alpha + \beta$ on ajoute un état initial relié par une transition vide aux états initiaux respectifs de M_α et M_β et un état final relié par une transition vide aux états finaux respectifs de M_α et M_β (cf. figure 3.8).
- Pour $\alpha\beta$ on relie tous les états finaux de M_α à l'état initial de M_β par une transition vide. L'état final est celui de M_α et les états finaux sont ceux de M_β (cf. figure 3.9).
- Pour $(\alpha)^*$ on introduit un état initial q_0 et un état final q_f. On ajoute des transitions vides de q_0 à l'état initial de M_α, de chaque état final de M_α à q_f, de q_0 à q_f et de q_f à l'état initial de M_α (cf. figure 3.10).

FIG. 3.5 – *Automate pour* $\varnothing$.

FIG. 3.6 – *Automate pour* ε.

FIG. 3.7 – *Automate pour* x.

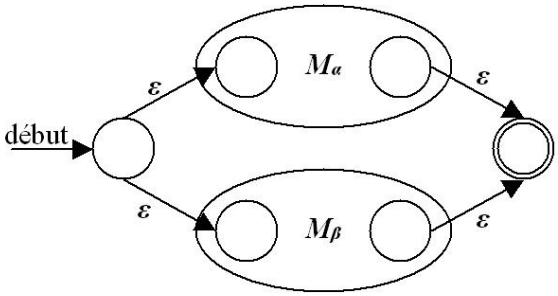


FIG. 3.8 – *Automate pour* $\alpha + \beta$.

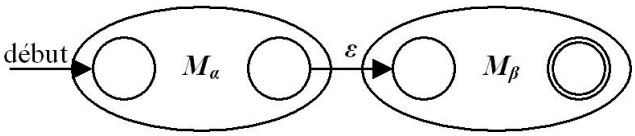


FIG. 3.9 – *Automate pour* $\alpha\beta$.

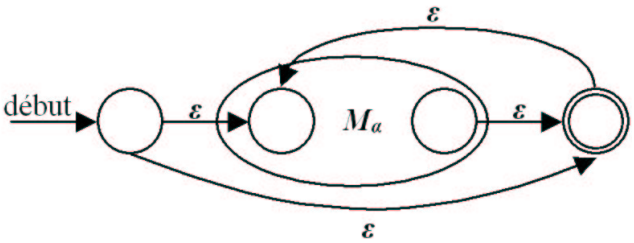


FIG. 3.10 – *Automate pour* $(\alpha)^*$.

3.4.3 Exemple de passage d'une expression régulière à un automate

La figure 3.11 montre l'automate fini qui décrit le même langage que l'expression régulière $(a(a+\varepsilon)b)^* + ba^*$.

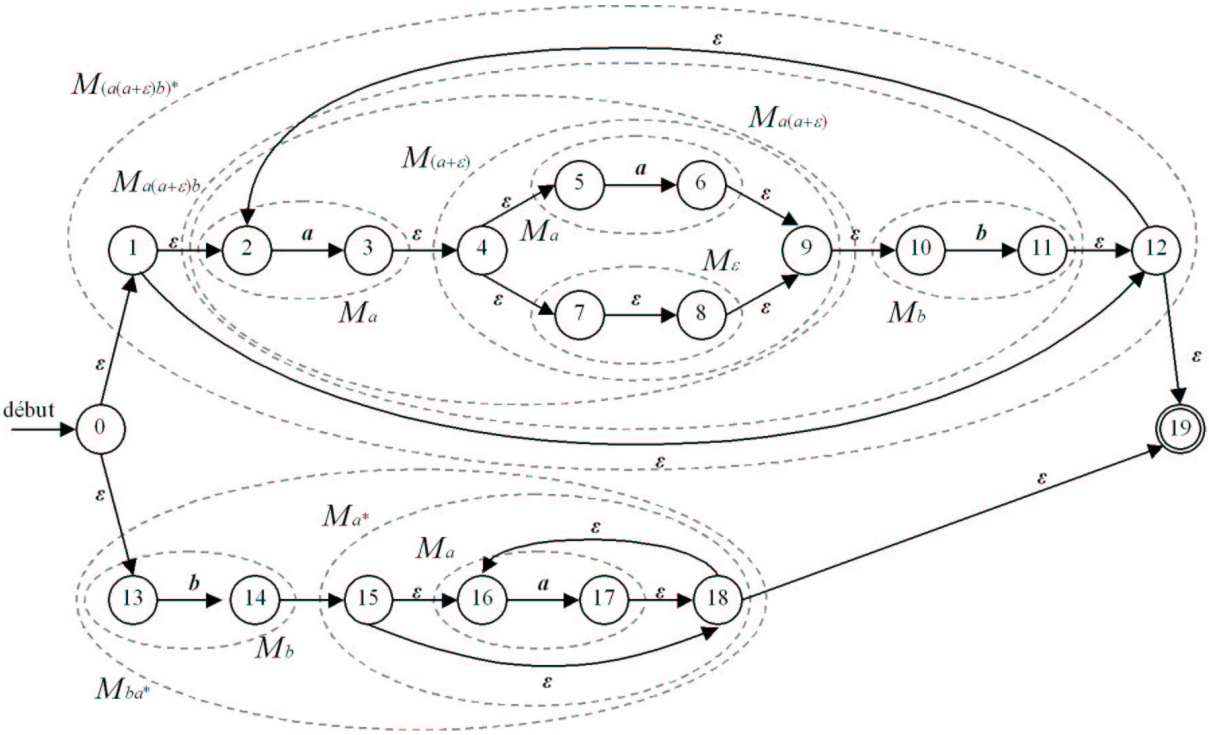


FIG. 3.11 – *Automate correspondant à l'expression régulière* $(a(a+\varepsilon)b)^* + ba^*$.

Bien entendu, il est possible de rendre déterministe l'automate de la figure 3.11 en supprimant les transitions sur le mot vide grâce à l'algorithme de la section 3.3.2 page 26.

3.5 TD : Automates

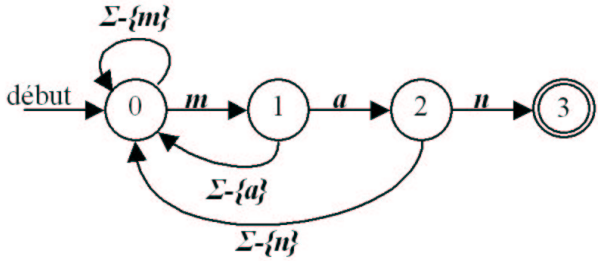
3.5.1 *Hey man !*

On se propose de trouver un automate déterministe capable de filtrer tous les textes contenant la portion de mot *man*. Cet automate sera défini sur le langage Σ qui correspond à l'ensemble des lettres contenues dans les textes à analyser.

Pour simplifier la tâche, nous dirons que la procédure de recherche sera terminée dès que la chaîne `man` aura été rencontrée. Nous cherchons donc un automate qui accepte les chaînes terminant en `man`. Ainsi, pour ce problème, nous dirons que le mot aura été reconnu par l'automate des que l'état final aura été atteint (donc que la chaîne `man` aura été rencontrée) sans attendre l'épuisement de tous les symboles du mot comme nous devrions

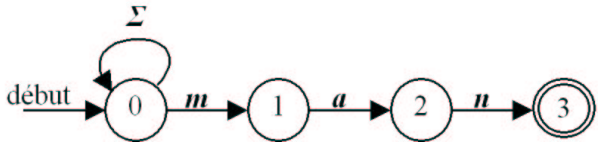
le faire en toute rigueur.

1. Une personne bien intentionnée vous propose l'automate suivant :



   Les portions de textes suivantes sont-elles acceptées par cet automate :
   - « l'analyse syntaxique et sémantique. » ;
   - « \pagenumbering{Roman} » ;
   - « Cet automate sera défini sur le langage Σ » ;
   - « sont les langages définis de la manière suivante » ;
   - « Figures, Sous figures et commandes complexes ».

   Cet automate rempli-t-il le cahier des charges ?
2. Une autre personne vous propose l'automate bien plus simple qui suit :



   Cet automate fonctionne-t-il mieux que le précédant ?
   Que pouvez-vous reprocher à cet automate ?
3. On cherche un automate déterministe équivalent à celui de la question 2. Pour cela, nous allons rendre déterministe cet automate en utilisant l'algorithme de détermi-nisation vu en cours. Pour la table de transition, vous considérerez les symboles m, a, n et $\Sigma - \{m,a,n\}$:

	m	a	n	$\Sigma - \{m,a,n\}$
0 (état initial)				

   Renumérotez les états et dessinez l'automate ainsi obtenu.

3.5.2 Mise à mal des transitions sur le mot vide

4. Soit l'expression régulière suivante : $(a + bc*)d$
   Construisez un automate fini non déterministe correspondant à cette expression régulière.
5. Simplifiez cet automate en supprimant les ε-transitions grâce à l'algorithme de la section 3.3.2 page 26.
6. Simplifiez l'automate de la figure 3.11 page 30 en supprimant les ε-transitions grâce à l'algorithme de la section 3.3.2 page 26.

3.6 JFLAP

3.6.1 Présentation de JFLAP

JFLAP[1] (*Java Formal Languages and Automata Package*) est un outil graphique pédagogique qui aide à la compréhension des concepts de base des langages formels et de la théorie des automates.

JFLAP permet de créer et de simuler un certain nombre d'automates dont les automates à états finis et les automates à pile. De plus, JFLAP offre des passerelles vers et depuis les expressions régulières et les grammaires.

L'application originale JFLAP a été programmée en C++ et fonctionne sous un environnement X-Window (Unix, Linux, ...). En raison de son succès en tant qu'outil pédagogique en théorie des langages formels et des automates, JFLAP a été réécrit en java de manière à pouvoir fonctionner sur n'importe quelle plateforme. Ainsi, la dernière version (JFLAP 3.1) a été écrite avec JDK 1.2 (*Java Development Kit*).

3.6.2 Installation

Etant programmé en java, JFLAP peut être utilisé comme une *applet* et fonctionne avec les navigateurs Netscape ou Internet Explorer. C'est cette façon de procéder que nous utiliserons en cours. Elle offre l'avantage de pouvoir utiliser JFLAP de n'importe quelle machine possédant un navigateur internet et permet de s'affranchir de toute installation. Cependant, il faut être conscient qu'utilisé de cette façon, JFLAP ne fonctionne pas aussi bien qu'en tant qu'application à part entière.

Il est possible d'utiliser JFLAP comme une application. Pour cela, il faut au préalable installer un interpréteur java, téléchargeable depuis le site de sun[2].

Il faut ensuite décompresser le fichier contenant l'application JFLAP complète, par exemple le fichier :

http://www.cs.duke.edu/csed/rodger/tools/jflap11-99.zip

La ligne de commande permettant de lancer l'application est :

```
java -classpath "D:\Program Files\JFLAP\jflap" JFLAP %1 %2 %3 %4 %5 %6 %7 %8 %9
```

Bien entendu, « `D:\Program Files\JFLAP` » doit être remplacé par le chemin où se trouve JFLAP sur votre station de travail.

3.6.3 Principe de base d'utilisation

La souris est utilisée pour construire les états et les transitions. Attention, vous devez utiliser une souris à trois boutons :

- utilisez le bouton du milieu pour créer et déplacer des états ;
- utilisez le bouton de gauche pour dessiner des transitions ;

1. JFLAP 3.1 : Susan H. Rodger, Magda Procopiuc, Octavian Procopiuc, Eric Gramond, Ted Hung ; Computer Science Department, Duke University ; Supported by National Science Foundation DUE-9596002, DUE-9555084, and DUE-9752583. Copyright (c) 1999 All rights reserved.
site : http://www.cs.duke.edu/∼rodger/tools/jflap/index.html
2. Site http://java.sun.com/ ; fichier j2re-1_4_1-windows-i586-i.exe ; taille : 9700Ko.

- utilisez le bouton de droite pour faire apparaître un menu contextuel permettant de
  - rendre un état final ;
  - rendre un état non-final ;
  - rendre un état initial ;
  - supprimer un état ;
  - supprimer une transition.

3.7 TP : JFLAP

Le but de cette séance de TP est, dans un premier temps, de construire et de tester un automate fini, et dans un deuxième temps, de convertir l'expression régulière $(a + bc*)d$ en un automate déterministe le plus simple possible.

La page « JFLAP DEMO APPLET » de l'*applet* de JFLAP se trouve à l'adresse :

`http://www.cs.duke.edu/~rodger/tools/jflap/jflap99/demo.html`

C'est depuis cette page que vous pourrez créer des automates à états finis ou encore convertir des expressions régulières. Cette page vous permet également d'accéder à l'aide (en anglais) de JFLAP. N'hésitez pas à consulter cette aide pour éclaircir ou approfondir certaines fonctionnalités de JFLAP.

3.7.1 Lexique et flexion

L'ensemble des mots d'un lexique peut être organisé sous la forme d'un automate à états finis, où chaque arc correspond à une lettre.

L'objectif est de construire un automate à états finis capable de représenter le langage de l'extrait de lexique suivant :

Forme fléchie	**Base**	**Cat.**	**Morphèmes**
ami	ami	N	ms
amis	ami	N	mp
amie	ami	N	fs
amies	ami	N	fp
amant	amant	N	ms
amants	amant	N	mp
amante	amant	N	fs
amantes	amant	N	fp
chien	chien	N	ms
chiens	chien	N	mp
chienne	chien	N	fs
chiennes	chien	N	fp

En fait, notre automate fini véhiculera toute l'information de la première colonne du tableau ci-dessus puisque l'ensemble des chemins possibles à travers le graphe correspondra exactement aux 12 formes du lexique. Cet automate pourra donc être utilisé en génération (il énumérera alors toutes les formes possibles) ou en reconnaissance (il « acceptera » alors les formes qui font partie du lexique, et « rejettera » les autres).

Le travail à effectuer est le suivant.

1. Réalisez cet automate. Pour cela, utilisez le bouton « Finite State Automaton » (ie. automate à états finis) de la page « JFLAP DEMO APPLET ».
2. JFLAP ne permet pas d'utiliser cet automate en génération. Par contre, vous pouvez l'utiliser un reconnaissance de trois manières différentes :

   Step run : saisissez préalablement la chaîne à tester dans la zone d'édition « Input String : » ; l'action *Step run* du menu *Option* permet d'effectuer une reconnaissance pas à pas.

   Fast run : saisissez préalablement la chaîne à tester dans la zone d'édition « Input String : » ; l'action *Fast run* du menu *Option* vous dit si la chaîne est acceptée ou reffusée par l'automate.

   Multiple run : l'action *Multiple run* du menu *Option* vous permet de tester simultanément plusieurs mots.

   Testez votre automate en reconnaissance.
3. L'automate peut être augmenté d'information morphologiques de façon à servir pour des traitements subséquents. Dans le menu *Option* validez *Show Name Labels*. Saisissez maintenant dans les états finaux l'information de la dernière colonne de notre lexique. Notre automate fini véhicule maintenant toute l'information de la première et de la quatrième colonne de notre index.

3.7.2 Ca se fait tout seul !

L'objectif de cet exercice est de convertir l'expression régulière $(a + bc*)d$ en un automate déterministe le plus simple possible. Cet exercice se décompose en plusieurs étapes.

1. La première étape consiste à saisir notre expression régulière. Pour cela, utilisez le bouton « Regular Expression Conversion » (ie. conversion d'expression régulière) de la page « JFLAP DEMO APPLET ». Saisissez votre expression régulière dans la zone d'édition : $(a + bc*)d$. Cliquez ensuite sur le bouton « Create FSA ».
2. Pour observer la création pas à pas de votre automate utilisez alternativement les actions *Show* et *Next Step* du menu *Solve*. Pour obtenir directement votre automate utilisez l'action *Show All* du menu *Solve*. Une fois votre automate terminé, remettez de l'ordre dans votre automate en déplaçant intelligemment les états (bouton du milieu de la souris).
3. Tester en reconnaissance cet automate. Utilisez également le mode pas à pas pour observer les chemins parcourus pour accepter ou rejeter un mot.
4. Pour rendre déterministe cet automate utilisez l'action *Convert to DFA* du menu *Options*. Cliquez sur *Show* du menu *Solve* pour afficher l'automate obtenu.
5. Pour simplifier cet automate utilisez l'action *Minimize* du menu *Options*

3.8 Conclusions sur les langages réguliers

3.8.1 Utilité des langages réguliers

Il est facile de démontrer que tout langage fini peut être engendré à partir d'une grammaire de type 3, donc tout langage fini est un langage régulier.

Les langages réguliers (expressions régulières, automates finis, grammaires régulières, ...) sont très utiles pour la mise au point de grammaires locales (par exemple pour les combinaisons pronominales pré-verbales en français) et dans le domaine de la morphologie.

Les grammaires régulières n'apportent pas grand chose par rapport aux expressions régulières et aux automates finis. Aussi ne nous étendrons-nous pas sur ces dernières.

3.8.2 Limitations des langages réguliers

Etant donné la forme des règles d'une grammaire de type 3, il est clair qu'à chaque pas de dérivation, la chaîne de symboles contient au plus un symbole non terminal. De plus, ce symbole est soit le dernier symbole de la chaîne (si la grammaire est linéaire à droite), soit le premier symbole de la chaîne (si la grammaire est linéaire à gauche). Il s'ensuit que la structure arborescente est elle-même soit linéaire à droite, soit linéaire à gauche.

Les langages réguliers ne sont donc pas adéquats pour modéliser des langages ayant la propriété d'auto-enchâssement (que nous verrons section **??**) comme le langage de *Dyck*.

Le langage de Dyck est défini sur l'alphabet $\{(,)\}$ et est constitué de mots dont la structure du parenthèsage est bien équilibrée (par exemple $(()(()))$, $(())(()())$, ...). Les langages réguliers ont la capacité d'engendrer tous les mots du langage de Dyck, par exemple l'expression régulière $(\ \backslash\ )\ |\ \backslash)\ )*$, mais ils ne peuvent engendrer que les mots de ce langage. Les langages réguliers ne peuvent donc pas modéliser un langage comme celui de Dyck ou comme celui des expressions arithmétiques (en mathématiques).

De la même manière, les langages réguliers ne sont pas adéquats pour modéliser le langage naturel.

Au niveau de la capacité générative forte, l'inadéquation des grammaires régulières est facile à établir. Prenons par exemple la structure arborescente de la figure 2.4. Cette structure représente la syntaxe de la phrase « le chat mange la souris » mais elle n'est ni linéaire à gauche, ni linéaire à droite. On peut donc conclure à l'inadéquation des grammaires régulières pour représenter la syntaxe des langues naturelles.

Au niveau de la capacité générative faible, la preuve de l'inadéquation des grammaires régulières pour le langage naturel est plus difficile à établir et sort du cadre de ce cours. Cette démonstration fait également intervenir la notion d'auto-enchâssement, propriété fondamentale des langages hors contexte (de type 2).

Cours 4

Langages hors contexte (type 2)

4.1 L'auto-enchâssement

La propriété d'*auto-enchâssement* est la caractéristique fondamentale des langages non réguliers. En effet, tout langage qui possède cette propriété d'auto-enchâssement est non régulier.

Définition 4.1 -grammaire auto-enchâssante- *Soit une grammaire G de vocabulaire non-terminal V_N et de vocabulaire terminal V_T. Cette grammaire est dite* auto-enchâssante *s'il existe un symbole non terminal A et des séquences de symboles x et y, x,$y \in \{V_N \cup V_T\}^* - \{\varepsilon\}$, tels que $A \vdash_G^* xAy$.*

En d'autres termes, une grammaire est auto-enchâssante s'il existe un élément non terminal A à partir duquel on peut dériver une chaîne d'éléments contenant A, avec un contexte gauche x et un contexte droit y non nuls, c'est-à-dire sans que A soit le premier ou le dernier élément de la chaîne.

Définition 4.2 -langage auto-enchâssant- *Un langage est dit* auto-enchâssant *si toutes les grammaires qui l'engendrent sont auto-enchâssantes.*

Il n'est donc pas suffisant qu'une grammaire soit auto-enchâssante pour que le langage engendré par cette grammaire soit lui-même auto-enchâssant.

4.1.0.1 Exemples de grammaires auto-enchâssantes

La grammaire suivante possède la propriété d'auto-enchâssement et engendre un langage auto-enchâssant (langage que l'on note généralement $a^n b^n$) :

- $V_N = \{S\}$;
- $V_T = \{a,b\}$;
- axiome : S ;
- règles de réécritures :

  $S \longrightarrow aSb$
  $S \longrightarrow ab$

La grammaire suivante possède la propriété d'auto-enchâssement et engendre un langage qui n'est pas auto-enchâssant mais régulier :

- $V_N = \{S\}$;
- $V_T = \{a\}$;
- axiome : S ;
- règles de réécritures :

$$S \longrightarrow aSa$$
$$S \longrightarrow aa$$
$$S \longrightarrow a$$

4.2 Automates à pile

4.2.1 Introduction

Le langage $\{a^n b^n ; n > 1\}$, qui est non régulier, n'est pas accepté par un automate à états finis. Pouvons-nous étendre la notion d'automate de manière à accepter de tels langages ?

Si nous réfléchissons intuitivement à cette question, nous voyons que ce qui manque, c'est un dispositif qui nous permettrait dans le cas particulier de ce langage, de mémoriser le nombre de a rencontrés dans un mot, de manière à ce qu'on puisse ensuite vérifier que les b sont en nombre égal.

Supposons par exemple que l'on dispose d'une marque X que l'on puisse écrire sur un ruban «spécial», à raison d'une occurrence par case, chaque fois qu'on trouve un a. Alors lorsque l'automate aura lu la première partie du mot (la suite de a), il aura «stocké» autant de X qu'il y a de a dans le mot. Ensuite, lorsqu'il lira la deuxième partie (les b du mot), il suffira qu'il «consomme» à chaque lecture un X, jusqu'à ce qu'il n'y en ait plus aucun. Si on arrive en effet à un stade où il n'y a plus de X sur le ruban «spécial», c'est qu'on aura bien compté les b et vérifié qu'il y en avait le même nombre que de a.

En fait, ce ruban «spécial» sera appelé une *pile*. La lecture d'un a sera associée au fait d'*empiler* un X et celle d'un b au fait de *dépiler*.

Un mot sera accepté (donc appartiendra au langage qui nous intéresse) si et seulement si on se trouve dans un état final avec une pile vide au moment où il ne reste plus de symboles à consommer sur le mot d'entrée.

Cette pile, qui joue le rôle d'une «mémoire», peut être implicite, c'est le cas dans les *automates récursifs*, ou explicite, c'est le cas pour les *automates à pile*. Ce genre d'automates acceptent exactement les langages hors contexte.

4.2.2 Automates récursifs

On appelle automates récursifs un ensemble de réseaux de transitions dont les transitions spécifient soit des éléments terminaux, soit des éléments non terminaux. Une transition spécifiant un élément non terminal peut être franchie de la même manière que dans un automate non-déterministe classique. Une transition spécifiant un élément non terminal correspond à un appel au réseau de transitions associé à cet élément non terminal. Une transition de ce type peut être satisfaite si le réseau appelé peut être parcouru avec succès.

En d'autre termes, une transition est satisfaite si l'élément spécifié par son étiquette – que ce soit un élément non terminal ou non – a été reconnu.

Comme un automate récursif peut invoquer d'autres automates, y compris lui-même, il est facile de montrer que ce mécanisme a la propriété d'auto-enchâssement, et qu'il a par conséquent la puissance nécessaire pour reconnaître des langages de type 2.

La figure 4.1 montre un automate récursif qui reconnaît le langage $a^n b^n$.

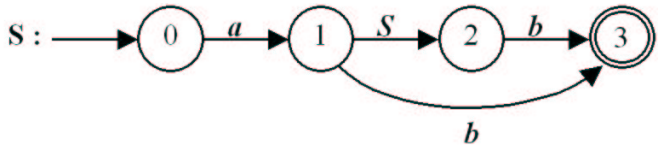


FIG. 4.1 – *Automate récursif pour le langage $a^n b^n$.*

La procédure de reconnaissance pour les automates récursifs fait intervenir une pile. Cette pile est nécessaire pour mémoriser l'état dans lequel l'automate devra se trouver suite à l'appel d'un sous réseau. Cette pile est utilisée de la manière suivante : lors d'une transition spécifiant un élément non terminal, on ajoute au sommet de la pile l'état auquel aboutit cette transition. L'état initial du réseau désigné par l'élément non terminal devient alors le nouvel état courant. Lorsque le système atteint un état final d'un sous-réseau, l'état placé au sommet de la pile est retiré de la pile et devient le nouvel état courant.

4.2.3 Automates à pile

Définition 4.3 -automate à pile- *Un* automate à pile *(*pushdown automaton*) est défini par :*

- *Un* ***ensemble fini E d'états*** *;*
- *Un état $e_0 \in E$ distingué comme étant l'****état initial****.*
- *un ensemble fini $F \subset E$ d'états distingués comme* ***états finaux*** *(ou* ***états terminaux****).*
- *Un* ***alphabet d'entrée*** *Σ des symboles qui constituent le mot à reconnaître.*
- *Un alphabet de pile Γ des symboles qui peuvent être mis dans la pile.*
- *Un* ***symbole initial de pile*** *$\$ \in \Gamma$;*
- *Une* ***relation de transition*** *Δ qui à tout triplet formé d'un état de E, d'un ensemble de symboles de Σ et d'un ensemble de symboles de Γ fait correspondre un ensemble (éventuellement vide) de couple formé d'un état de E et d'un ensemble de symboles de Γ : $\Delta \subset ((E \times \Sigma^* \times \Gamma^*) \times (Q \times \Gamma^*))$.*

Σ et Γ ne sont pas nécessairement distincts. $ est le contenu initial de la pile. Un élément de Δ est donc un couple $((e_i,u,\beta),(e_j,\gamma))$ qui signifie : si le mot d'entrée commence par le préfixe u et si la chaîne β se trouve en haut de la pile, alors l'automate peut passer de l'état e_i à l'état e_j. Une fois cela fait, il replace le haut de la pile β par γ et passe au symbole suivant le symbole u dans le mot à reconnaître.

Un automate à pile ainsi défini correspond à la notion d'automate non déterministe (les transitions sont données sous la forme d'une relation et non d'une fonction).

Empiler un symbole de pile (ou un mot formé de plusieurs symboles de pile) β se représente donc par la transition

$$(p,u,\varepsilon) \rightarrow (q,\beta)$$

Dépiler β se représente par :

$$(p,u,\beta) \rightarrow (q,\varepsilon)$$

Théorème 4.1 – *un langage est hors-contexte si et seulement s'il est accepté par un automate à pile.*

Il est maintenant intéressant de se demander si nous ne pouvons pas restreindre la définition 4.3 de manière à définir des automates à pile déterministes. C'est bien-entendu possible. Cependant on ne peut pas toujours trouver un automate à pile déterministe reconnaissant le même langage qu'un automate à pile non-déterministe. En fait, les automates à pile déterministes reconnaissent une classe de langages plus réduite que les langages hors-contexte dans leur totalité.

4.2.4 Représentation par un diagramme d'états

On peu représenter un automate à états finis par un réseau de transition en dessinant des états et des arc étiquetés par des symboles (ou des mots). On peut représenter un automate à pile par un réseau du même genre mais où les arcs sont étiquetés à la fois par un symbole (ou un mot) et par une action $\alpha \rightarrow \beta$ sur la pile. l'action $\alpha \rightarrow \beta$ signifie que l'on dépile α et que l'on empile β. Empiler un symbole β sera donc noté $\varepsilon \rightarrow \beta$ tandis que dépiler β sera noté $\beta \rightarrow \varepsilon$.

Voici un exemple d'automate à pile représenté par un réseau de transition :

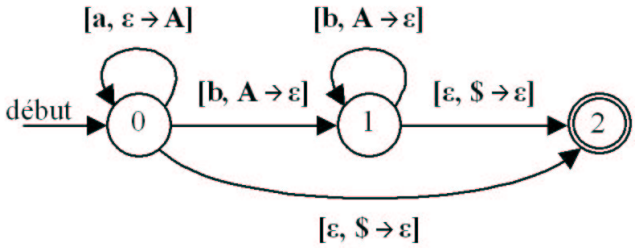


FIG. 4.2 – *Exemple de réseau de transition pour un automate à pile.*

4.2.5 TD : automates à pile

4.2.5.1 Reconnaissance

Dites si l'automate à pile de la figure 4.2 reconnaît les mots suivants :

1. ε ;
2. aa ;
3. ba ;

4. ab ;
5. aba ;
6. aab ;
7. $aabb$;
8. $aaaabb$;
9. $aaabbb$.

Quel langage décrit en fait cet automate ?

4.2.5.2 Forme de la définition

Ecrivez l'automate à pile de la figure 4.2 sous la forme de la définition 4.3.

4.2.5.3 Langage miroir

Trouvez un automate à pile qui accepte le langage $\{w\overline{w} : w \in \{a,b\}^*\}$ où $\overline{w}$ désigne le « mot en miroir » de w. Par exemple, les mots $aaabbaaa$, $baab$, $ababbaba$ sont reconnus.

Dessinez son réseau de transition dans l'application JFLAP et testez votre réseau.

4.3 Normalisation des grammaires de type 2

4.3.1 Introduction

Nous avons vu que, selon la définition des grammaires de type 2, la partie droite des règles peut consister en n'importe quel mot non nul constitué de symboles terminaux ou non terminaux. Il est cependant possible de contraindre la forme des règles, de manière à simplifier tout travail sur la grammaire, sans diminuer la capacité générative de la grammaire. Pour cela, on peut dans un premier temps, éliminer les règles de la forme $A \longrightarrow \varepsilon$ $(A \neq S)$, les règles unaires et les règles inutiles, de manière à obtenir une nouvelle grammaire équivalente « propre ». On peut ensuite normaliser cette grammaire pour obtenir une grammaire en forme normale.

Les formes normales les plus connues pour les grammaires hors contextes sont les formes normales de *Chomsky* et de *Greibach*.

Tout langage de type 2 peut être engendré par une grammaire de type 2 propre et en forme normale de Chomsky ou de Greibach. Autrement dit, les restrictions imposées par les formes propres et normales n'ont aucun effet sur la puissance descriptive des grammaires de type 2.

Les formes propres et normales conservent également l'ambiguïté. Une phrase ambiguë engendrée par une grammaire de type 2 sera également ambiguë dans le langage engendré par une grammaire propre équivalente en forme normale de Chomsky ou de Greibach.

Par contre, les formes propres et normales ne conservent pas les structures arborescentes. Ainsi une grammaire quelconque de type 2 sera seulement faiblement équivalente à la grammaire propre en forme normale de Chomsky ou de Greibach associée.

4.3.2 Obtenir une grammaire propre : élimination des règles $A \longrightarrow \varepsilon$, des règles unaires et inutiles

4.3.2.1 Définitions

Soit $G =< V_N, V_T, S, R >$ une grammaire de type étendu 2 d'ensemble de symboles non-terminaux V_N, d'ensemble de symboles terminaux V_T, d'axiome S et d'ensemble de règles de réécriture R. Si $A \in V_N$, notons $L_G(A)$ l'ensemble des mots terminaux $s \subset V_T^*$ qui dérivent de A : $L_G(A) = \{s : A \vdash_G^* s, s \subset V_T^*\}$.

Définition 4.4 **-symbole improductif-** *Un symbole A non-terminal $A \in V_N$ est dit* improductif *si et seulement si on ne peut rien dériver de A : $L_G(A) = \varnothing$.*

Définition 4.5 **-symbole inaccessible-** *Un symbole $X \in V_N \cup V_T$ est dit* inaccessible *si et seulement si il n'existe aucun mot $\varphi \in (V_N \cup V_T)^*$ contenant X et dérivant de l'axiome S ($\forall \varphi \in (V_N \cup V_T)^*, \nexists X : X \subset \varphi, S \vdash_G^* \varphi$).*

Définition 4.6 **-symbole inutile-** *Un symbole est dit* inutile *s'il est improductif ou inaccessible.*

Définition 4.7 **-règle unaire-** *Une règle est dire* unaire *si elle est de la forme $A \longrightarrow B$ avec $A, B \in V_N$.*

Définition 4.8 **-grammaire avec cycle-** *Une grammaire G contient* un cycle *si et seulement si $\exists A \in V_N : A \vdash_G^* A$.*

Définition 4.9 **-grammaire propre-** *Une grammaire G est dite* propre *si elle est :*

- *sans cycles ;*
- *sans symbole inutile ;*
- *sans règle du genre $A \longrightarrow \varepsilon$, sauf si $A = S$ (mais alors S ne figure dans aucune partie droite).*

Théorème 4.2 *– Tout langage hors-contexte admet une grammaire propre.*

Autrement dit : pour toute grammaire G hors contexte, il existe une grammaire G' hors contexte et propre qui engendre le même langage. On dit alors que G et G' sont faiblement équivalentes.

4.3.2.2 Elimination des règles : $A \longrightarrow \varepsilon$

Pour toute règle $A \longrightarrow \varepsilon$, on regarde toutes les règles qui contiennent A en partie droite et pour toutes ces règles, de la forme $B \longrightarrow uAv$, on ajoute la règle $B \longrightarrow uv$ (et si, ce faisant, on fait apparaître une nouvelle règle $B \longrightarrow \varepsilon$, on réitère le processus). On élimine ensuite toutes les règles du genre $A \longrightarrow \varepsilon$. Si, parmi les règles éliminées, figure $S \longrightarrow \varepsilon$, alors on introduit un nouvel axiome S' et on ajoute la règle $S' \longrightarrow \varepsilon$ ainsi qu'une règle $S' \longrightarrow \varphi$ pour chaque règle $S \longrightarrow \varphi$.

4.3.2.3 Elimination des règles unaires

S'il existe une règle du genre $A \longrightarrow B$ avec A,$B \in V_N$, alors pour chaque règle $B \longrightarrow \varphi$ on ajoute à la grammaire la règle $A \longrightarrow \varphi$, et on peut alors supprimer la règle $A \longrightarrow B$. On réitère la procédure tant qu'il reste des règles du genre $A \longrightarrow B$ avec A,$B \in V_N$.

4.3.2.4 Elimination des règles inutiles

Recherche des symboles productifs – On notera Ep l'ensemble des symboles non-terminaux productifs. Un symbole non-terminal est productif si et seulement si on peut en dériver un mot terminal. Cet ensemble Ep peut être construit à partir de l'algorithme suivant :

$E[0] \leftarrow \varnothing$
$i \leftarrow 0$
Répéter :
  $i \leftarrow i+1$
  $E[i] \leftarrow E[i-1] \cup \{A : (A \longrightarrow \varphi) \in R \text{ et } \varphi \in (E[i-1] \cup V_T)^*\}$
jusqu'à $(E[i] = E[i-1])$
$Ep = E[i]$

Recherche des symboles accessibles – On notera Ea l'ensemble des symboles non-terminaux accessibles. Un symbole non-terminal est accessible si et seulement si il existe un mot le contenant dérivant de l'axiome S. Cet ensemble Ea peut être construit à partir de l'algorithme suivant :

$E[0] \leftarrow \{S\}$
$i \leftarrow 0$
Répéter :
  $i \leftarrow i+1$
  $E[i] \leftarrow E[i-1] \cup \{X \in V_N \cup V_T : \exists (A \longrightarrow \varphi X \psi) \in R \text{ et } A \in E[i-1]\}$
jusqu'à $(E[i] = E[i-1])$
$Ea = E[i]$

Elimination des règles inutiles – Toutes les règles qui contiennent un symbole inutile (i.e. un symbole improductif ou inaccessible) peuvent être supprimée. On supprimera donc toutes les règles qui contiennent des symboles non-terminaux qui ne sont pas dans Ep (symboles improductifs) et toutes les règles qui contiennent des symboles qui ne sont pas dans Ea (symboles inaccessibles).

4.3.3 Forme normale de Greibach

Définition 4.10 -forme normale de Greibach- *Une grammaire est dite en* forme normale de Greibach*, si et seulement si les règles sont de la forme : $A \longrightarrow a\beta$ avec $\beta \in V_N^*$ et $a \in V_T$.*

La partie droite d'une règle de réécriture, d'une grammaire en forme normale de Greibach, comprend obligatoirement un élément terminal unique, suivi d'une séquence (éventuellement nulle) d'éléments non terminaux.

Voici un exemple de grammaire en forme normale de Greibach pour le langage $a^n b^n$ avec $n \geq 1$:

- $V_N = \{S,A\}$;
- $V_T = \{a,b\}$;
- axiome : S ;
- règles de réécritures :

$$S \longrightarrow aSB$$
$$S \longrightarrow aB$$
$$B \longrightarrow b$$

4.3.4 Forme normale de Chomsky

4.3.4.1 Définition

Définition 4.11 -forme normale de Chomsky- *Une grammaire est dite en* forme normale de Chomsky*, si et seulement si les règles sont de la forme : $A \longrightarrow BC$ (schéma 1) ou $A \longrightarrow a$ (schéma 2) avec $A,B,C \in V_N$ et $a \in V_T$.*

La forme normale de Chomsky impose donc une distinction stricte entre dérivation d'éléments terminaux et dérivation d'éléments non terminaux. La dérivation d'un élément terminal ne peut se faire que par l'intermédiaire d'une règle conforme au schéma 2 où un élément non terminal se réécrit sous la forme d'un élément terminal unique. D'un autre côté, la dérivation d'éléments non terminaux se fait au moyen de règles conformes au schéma 1, qui réécrivent un élément non terminal sous la forme d'exactement deux éléments non terminaux. Il s'ensuit que les structures associées aux phrases engendrées par une grammaire en forme normale de Chomsky sont strictement binaire dans leur partie non terminale.

L'un des intérêts d'une grammaire en forme normale de Chomsky est de permettre l'utilisation d'un algorithme relativement efficace pour reconnaître l'appartenance d'un mot à cette grammaire (cf. section 5.2).

4.3.4.2 Mise en forme normale de Chomsky

Soit une règle $A \longrightarrow X_1 X_2 \ldots X_n$ où $n \geq 2$.

Si X_i est un terminal a, alors on introduit un nouveau non terminal C_a, conjointement avec la règle $C_a \longrightarrow a$ et on remplace X_i par C_a. On peut ainsi se ramener à une règle de la forme $A \longrightarrow B_1 B_2 \ldots B_n$ où tous les B_i sont des non-terminaux.

Pour chaque $A \longrightarrow B_1 B_2 \ldots B_m$ où tous les B_i sont des non-terminaux et avec $m \geq 3$, on introduit des non-terminaux $D_1,D_2,\ldots,D_{m-2}$ et les règles :

$$A \longrightarrow B_1 D_1$$
$$D_1 \longrightarrow B_2 D_2$$
$$D_2 \longrightarrow B_3 D_3$$
$$\ldots$$

$D_{m-3} \longrightarrow B_{m-2}D_{m-2}$
$D_{m-2} \longrightarrow B_{m-1}B_m$

La grammaire ainsi obtenue est faiblement équivalente à la grammaire de départ et est en forme normale de Chomsky.

4.3.4.3 Exemple de mise en forme normale de Chomsky

Soit la grammaire, pour le langage a^nb^n avec $n \geq 1$, suivante :

- $V_N = \{S\}$;
- $V_T = \{a,b\}$;
- axiome : S ;
- règles de réécritures :

  $S \longrightarrow aSb$
  $S \longrightarrow ab$

On se propose de trouver une grammaire faiblement équivalente en forme normale de Chomsky.

La règle $S \longrightarrow aSb$ peut se réécrire :

$S \longrightarrow ASB$
$A \longrightarrow a$
$B \longrightarrow b$

Il faut ensuite décomposer $S \longrightarrow ASB$, on obtient alors :

$S \longrightarrow AC$
$C \longrightarrow SB$
$A \longrightarrow a$
$B \longrightarrow b$

Il ne reste plus que la règle $S \longrightarrow ab$ que l'on peut réécrire $S \longrightarrow AB$ ce qui permet de réutiliser les règles déjà produites $A \longrightarrow a$ et $B \longrightarrow b$.

La grammaire en forme normale de Chomsky finalement obtenue est :

- $V_N = \{S,A,B,C\}$;
- $V_T = \{a,b\}$;
- axiome : S ;
- règles de réécritures :

  $S \longrightarrow AB$
  $S \longrightarrow AC$
  $C \longrightarrow SB$
  $A \longrightarrow a$
  $B \longrightarrow b$

4.3.5 TD : normalisation des grammaires

4.3.5.1 Grand nettoyage

Trouvez une grammaire propre équivalente à la grammaire suivante :

- $V_N = \{S,A,B,D,E\}$;

- $V_T = \{a,b\}$;
- axiome : S ;
- règles de réécritures :

$$S \longrightarrow ASB$$
$$S \longrightarrow \varepsilon$$
$$A \longrightarrow D$$
$$A \longrightarrow a$$
$$E \longrightarrow a$$
$$B \longrightarrow b$$

4.3.5.2 Langage miroir

1. Trouvez une grammaire qui accepte le langage $\{w\overline{w} : w \in \{a,b\}^*\}$ où $\overline{w}$ désigne le « mot en miroir » de w. Par exemple, les mots *aaabbaaa*, *baab*, *ababbaba* font parti de ce langage, mais pas ε.
2. Mettez en forme normal de Chomsky la grammaire établie ci-dessus.

Cours 5

Analyseurs (*parser*) syntaxiques

5.1 Analyse lexicale et syntaxique : présentation

Informellement, on peut considérer que le rôle de l'analyse lexical consiste à découper une chaîne d'entrée (un mot) en une séquence d'unités lexicales, ou symboles (*token* en anglais). Ces derniers constituent les éléments terminaux de l'analyse syntaxique [1].

Le rôle de l'analyseur syntaxique (*parser* en anglais) consiste à vérifier que la syntaxe d'un langage a bien été respectée. Il analyse une chaîne de caractères (ou plutôt une chaîne de *token* fournie par l'analyseur lexical), détermine si celle-ci fait partie du langage et construit l'arbre syntaxique (ou arbre de dérivation) correspondant. Pour faire partie du langage, il faut que l'on puisse, à partir de la grammaire et de cette chaîne, écrire une dérivation : la suite des règles de production à utiliser pour obtenir la chaîne de départ.

Classiquement, avant de construire un analyseur syntaxique, on rajoute une variable S' qui devient le nouvel axiome de la grammaire, un terminal \$ ou EOF représentant la fin de la chaîne, et une règle $S' \longrightarrow S\$$.

Certain langage permettent la construction d'analyseurs efficaces capables de construire l'arbre de dérivation de tout les mots du langage. Avec de tels analyseurs, si l'analyse d'un mot donné se révèle impossible, cela signifie que le mot n'appartient pas au langage.

Il y a au moins deux grandes classes de techniques différentes pour construire un arbre syntaxique et vérifier si un mot appartient au langage, donc deux grandes classes d'analyseurs syntaxiques. Une première façon de faire consiste à partir du symbole initial de la grammaire (l'axiome) et à construire une représentation de haut en bas, c'est-à-dire de la racine de l'arbre vers les extrémités des branches. Alternativement, on peut partir des mots de la phrase, c'est-à-dire des éléments terminaux de la grammaire et tenter de réduire ces éléments en constituants de niveau supérieur jusqu'à ce que l'on parvienne à la racine de la structure. Dans le premier cas on parle de procédure *descendante* et dans le second cas de procédure *ascendante*. Comme nous le verrons, une procédure descendante simule une dérivation la plus à gauche, alors qu'une procédure ascendante simule une dérivation la plus à droite.

1. Quand on analyse une langue naturel, les symboles terminaux sont des mots de la langue. Il est alors courant de considérer les catégories lexicales des mots de la langue plutôt que les mots eux-même comme éléments terminaux de la grammaire. Ces grammaires n'engendrent donc pas des phrases au sens usuel du terme, mais des séquences de catégories lexicales. Dans ce cadre, le rôle de l'analyse lexical ne se borne pas seulement à découper la chaîne d'entrée en mots de la langue, il doit également retourner la catégorie lexicale de ces mots.

5.2 Algorithme de Cocke, Younger et Kasami

L'algorithme de Cocke, Younger et Kasami est un algorithme tabulaire limité aux grammaires en forme normale de Chomsky. La reconnaissance est effectuée de façon ascendante, la génération des structures de façon descendante. Aussi dirons-nous qu'il s'agit d'un algorithme mixte. Nous ne présenterons ici que l'algorithme de reconnaissance.

5.2.1 Algorithme de reconnaissance

Soit G une grammaire en forme normale de Chomsky. L'algorithme de Cocke, Younger et Kasami permet de reconnaître si un mot w appartient au langage engendré par G.

Supposons que w soit de longueur n. L'idée est la suivante : rechercher pour chaque sous-mot $w[i,j]$ de longueur j commençant à la position i tous les non-terminaux X tels que $X \vdash_G^* w[i,j]$, en commençant évidemment par les valeurs de j les plus petites, jusqu'à atteindre $j = n$. Alors quand $j = n$, il suffit de regarder si le symbole axiome S fait partie des non-terminaux obtenus.

Pratiquement, on construit un tableau triangulaire. Les colonnes sont numérotées par $i = 1,2,\ldots,n$ (les positions de début du sous-mot dans le mot w) et les lignes par $j = 1,2,\ldots,n$ (les longueurs possibles des sous-mots). Evidemment, la ligne 1 a n colonnes remplies, la ligne 2 en a $n-1$ et ainsi de suite, la ligne k en a $n-k+1$ et la ligne n n'en a qu'une.

On commence par remplir les cases de la ligne 1, puis celles de la ligne 2 et ainsi de suite. Le contenu $c_{i,j}$ de la case i,j, intersection de la colonne i et de la ligne j, correspond a l'ensemble des non-terminaux qui peuvent engendrer le sous-mot $w[i,j]$. Ce sous-mot peut se décomposer en deux sous-mots : $w[i,k]$ et $w[i+k,j-k]$. Il faut examiner toutes les coupures du mot $w[i,j]$ en faisant varier k entre 1 et j $(1 \leq k < j)$. Les ensembles de non-terminaux ($c_{i,k}$ et $c_{i+k,j-k}$) des cases correspondants à ces sous-mots étant déjà générés, on regarde toutes les expressions YZ avec $Y \in c_{i,k}$ et $Z \in c_{i+k,j-k}$ en recherchant s'il existe un non-terminal X dans la grammaire tel que $X \longrightarrow YZ$. S'il existe un tel X, on le met dans l'ensemble $c_{i,j}$ de la case i,j, cela pour tous les X possibles et tous les k compris entre 1 et j.

Etant donnés une grammaire G et un mot w de longueur n, on peut écrire cet algorithme de la manière suivante :

1. $c_{i,1} = \{X : X \longrightarrow \alpha_i \in G\}$ où α_i est le i^{e} symbole du mot w.
2. $c_{i,j\neq 1} = \{X : \forall k\ ,\ 1 \leq k < j\ ,\ X \longrightarrow YZ \in G \text{ avec } Y \in c_{i,k} \text{ et } Z \in c_{i+k,j-k}\}$

5.2.2 Exemple

Prenons la grammaire G en forme normale de Chomsky suivante :

- $V_N = \{S,A\}$;
- $V_T = \{a,b\}$;
- axiome : S ;
- règles de réécritures :

$$S \longrightarrow AA$$
$$S \longrightarrow AS$$

$S \longrightarrow b$
$A \longrightarrow SA$
$A \longrightarrow AS$
$A \longrightarrow a$

	1	2	3	4	5	*i (position de début du sous-mot)*
1	$\{A\}$	$\boldsymbol{\{S\}}$	$\{A\}$	$\{A\}$	$\{S\}$	
2	$\{A,S\}$	$\{A\}$	$\{S\}$	$\{A,S\}$		
3	$\boldsymbol{\{A,S\}}$	$\{S\}$	$\{A,S\}$			
4	$\{A,S\}$	$\{A,S\}$				
5	$\{A,S\}$					
j (longueur du sous-mot)						

FIG. 5.1 – *Table de dérivation pour le mot abaab*

On se propose, en utilisant l'algorithme de Cocke, Younger et Kasami, de déterminer si le mot *abaab* appartient au langage engendré par la grammaire G.

On commence par remplir la première ligne du tableau : $c_{i,1} = \{X : X \longrightarrow \alpha_i \in G\}$. Par exemple, pour $c_{2,1}$ on a $\alpha_2 = b$ (2^{e} symbole du mot $w : a\boxed{b}aab$). Dans notre grammaire G, il n'y a qu'une règle qui permet de produire ce symbole terminal b, c'est $S \longrightarrow b$. Donc $c_{2,1} = \{S\}$.

On rempli ensuite la 2^{e} ligne, puis la 3^{e}, puis la 4^{e}, puis la 5^{e} de la manière suivante : $c_{i,j} = \{X : \forall k\ ,\ 1 \leq k < j\ ,\ X \longrightarrow YZ \in G \text{ avec } Y \in c_{i,k} \text{ et } Z \in c_{i+k,j-k}\}$. Prenons, par exemple, $c_{1,3} : c_{1,3} = \{X : \forall k\ ,\ 1 \leq k < 3\ ,\ X \longrightarrow YZ \in G \text{ avec } Y \in c_{1,k} \text{ et } Z \in c_{1+k,3-k}\}$. Ici $1 \leq k < 3$ donc $k = 1$ ou $k = 2$.

- Pour $k = 1$ on veut $Y \in c_{1,1}$ et $Z \in c_{2,2}$ soit $Y \in \{A\}$ et $Z \in \{A\}$, ce qui donne $X \longrightarrow AA$ donc $X = S$ (puisque seule la règle $S \longrightarrow AA$ produit AA dans G).
- Pour $k = 2$ on veut $Y \in c_{1,2}$ et $Z \in c_{3,1}$ soit $Y \in \{A,S\}$ et $Z \in \{A\}$, ce qui donne $X \longrightarrow AA$ ou $X \longrightarrow SA$ donc $X = S$ ou $X = A$ (puisque seule la règle $S \longrightarrow AA$ produit AA et seule la règle $A \longrightarrow SA$ produit SA dans G).

D'où $c_{1,3} = \{A,S\}$

5.2.3 TD : algorithme de Cocke, Younger et Kasami

Soit la grammaire G suivante :

- $V_N = \{S,A,B,C,D\}$;
- $V_T = \{a,b\}$;
- axiome : S ;
- règles de réécritures :

  $S \longrightarrow AC$
  $S \longrightarrow BD$

$S \longrightarrow AA$
$S \longrightarrow BB$
$C \longrightarrow SA$
$D \longrightarrow SB$
$A \longrightarrow a$
$B \longrightarrow b$

En utilisant l'algorithme de Cocke, Younger et Kasami, dites si les mots suivants sont reconnus par cette grammaire :

1. *aa* ;
2. *ababa* ;
3. *bbaabb* .

5.3 Analyse déterministe descendante LL(k)

5.3.1 Introduction

On appelle *analyses descendantes* les procédures qui essaient de reconstruire la dérivation d'un mot en partant du symbole initial de la grammaire et en effectuant une dérivation la plus à gauche. La dérivation est obtenue en remplaçant itérativement les symboles non terminaux de la séquence par une expansion de ces symboles jusqu'à ce que la séquence de symboles dérivée ne comporte plus que des symboles terminaux. Le mot est dit reconnu s'il correspond à cette séquence de symboles, la séquence de pas de la dérivation détermine une analyse du mot.

Les analyseurs qui utilisent une stratégie descendante sont également appelés *analyseurs prédictifs*, puisqu'ils procèdent par une succession de prédictions ou d'hypothèses.

Nous présenterons ici, sans trop entrer dans les détails, ce que sont les *analyseurs LL(k)*, dont les lettres signifient respectivement :

- **L** pour *Left scanning*, c'est à dire analyse des symboles du mot de gauche à droite ;
- **L** pour *Left parsing* pour dérivation la plus à gauche ;
- **(k)** pour k symboles (ou jetons ou *token* en anglais) de prévision (*lookahead* en anglais).

Les analyseurs LL(k) réalisent l'analyse syntaxique du mot sans retour en arrière (ie. de manière déterministe) pour les grammaires dites LL(k). Les grammaires LL(k) constituent un sous-ensemble des grammaires hors-contexte.

Le succès des analyseurs LL(k) vient du fait qu'il sont facile à mettre en œuvre à la main. Parmi ces analyseurs, les analyseurs LL(1), bien que très limités, sont très populaires puisqu'ils ont seulement besoin de connaître le symbole suivant pour prendre leur décision d'analyse.

5.3.2 Analyse de type LL(0)

Considérons la grammaire suivante :

- $V_N = \{S,A,B,C\}$;

- $V_T = \{a,b,c\}$;
- axiome : S ;
- règles de réécritures :

$$S \longrightarrow ACA$$
$$A \longrightarrow a$$
$$C \longrightarrow c$$

Supposons que nous ayons à reconnaître le mot *aca*. On suppose que ce mot peut être une dérivation de S. On effectue une dérivation la plus à gauche de ce symbole non terminal :

S (règle $S \longrightarrow ACA$)
ACA (règle $A \longrightarrow a$)
aCA (règle $C \longrightarrow c$)
acA (règle $A \longrightarrow a$)
aca

Le mot *aca* est bien reconnu. Ce type d'analyse est en fait une *analyse du type LL(0).*

Dans cette grammaire, il n'y a qu'une seule règles de production possible pour un même symbole. En fait, cette grammaire décrit un langage ne contenant qu'un seul mot, le mot *aca*.

Cependant, de manière général, les grammaires de type 2 ont plusieurs règles de production possibles pour un même symbole.

5.3.3 Analyse de type LL(1)

Considérons la grammaire suivante :

- $V_N = \{S\}$;
- $V_T = \{a,b,c\}$;
- axiome : S ;
- règles de réécritures :

$$S \longrightarrow aSa$$
$$S \longrightarrow bSb$$
$$S \longrightarrow c$$

Si nous voulons analyser le mot *abacaba* par une démarche descendante gauchedroite, nous aurons à chaque étape le choix à faire entre ces trois règles, ce qui rend cette analyse impossible à faire de manière déterministe. En effet, il faudrait se donner la possibilité de faire des retours en arrière : on essaye la 1re règle, si on aboutit à un échec, alors on retourne en arrière et on essaye la deuxième règle, etc. Mais cette démarche correspond à une analyse non déterministe.

Pourtant, on voit bien intuitivement qu'il s'en faut de peu : il suffirait qu'on soit autorisé à jeter un coup d'oeil sur le premier symbole du mot qui reste à analyser pour que le choix de la bonne règle puisse se faire automatiquement. C'est ce qu'on appelle une analyse prédictive.

Un type d'analyse prédictive est associé à un nombre de symboles qu'on se donne le droit de regarder pour choisir une règle, c'est ce qu'on appelle le regard en avant (*lookahead*

en anglais). Ainsi une *analyse de type LL(1)* est une analyse selon la stratégie descendante, gauchedroite, avec un regard en avant de 1 symbole.

Reprenons le mot *abacaba* et la grammaire ci-dessus. Tentons une analyse du type *LL(1)*. Lors d'un pas de dérivation, pour savoir quelle règle utiliser, il faut regarder le dernier symbole non reconnu du mot.

On suppose que ce mot peut être une dérivation de S.
Le mot commençant par a, le seul pas de dérivation possible est : aSa (en utilisant la règle $S \longrightarrow aSa$).
Le second symbole étant b, le seul pas de dérivation possible est : $abSba$ (en utilisant la règle $S \longrightarrow bSb$).
Le troisième symbole étant a, le seul pas de dérivation possible est : $abaSaba$ (en utilisant la règle $S \longrightarrow aSa$).
Le quatrième symbole étant c, le seul pas de dérivation possible est : $abacaba$ (en utilisant la règle $S \longrightarrow c$).
La séquence de symboles dérivée ne comporte plus que des symboles terminaux. Le mot est *abacaba* est bien reconnu puisqu'il correspond au mot dérivé *abacaba*.

5.3.4 Principes de construction des analyseurs LL(k)

Pour pouvoir automatiser la démarche que nous venons de voir, il faut construire une table d'analyse qui permette de désigner la règle à appliquer en fonction du symbole à dériver et des k symboles qui suivent le symbole où l'on se trouve dans le mot à analyser.

La construction d'un analyseur prédictif du type LL(k) repose donc sur celle de la table d'analyse. La construction de la table d'analyse repose elle-même sur le calcul des fonctions $First_k()$ et $Follow_k()$

Pour un symbole A, on définit $First_k(A)$ comme l'ensemble des k premiers symboles des mots que l'on peut produire à partir de A.

On définit $Follow_k(A)$ comme l'ensemble des k premiers symboles qui peuvent suivre la production de A.

Il existe des algorithmes assez simples pour calculer ces fonctions (se référer à la littérature spécialisée).

Une grammaire sera dite *LL(k)* si une prévision de k symboles permet de déterminer les productions à utiliser, c'est à dire qu'on ne rencontre pas de conflit $First/First$ ou $First/Follow$ (cf section 5.4.3).

5.3.5 Table d'analyse pour les grammaires LL(1)

Nous présentons ici un algorithme qui permet de construire une table d'analyse M pour une grammaire G. La table d'analyse M est une matrice $M[A,x]$ où A est un symbole non-terminal et x un symbole terminal ou bien le symbole de fin de mot \$ [2].

Pour chaque règle $A \longrightarrow \psi$ **faire :**

Pour chaque terminal a dans $First_1(\psi)$ **ajouter** $A \longrightarrow \psi$ à $M[A,a]$.

2. Les symbole \$ est simplement ajouté à la fin de la liste des symboles du mot, il sert à signaler la fin du mot.

Si ε est dans $First_1(\psi)$, **alors**

Pour chaque terminal b dans $Follow_1(A)$ **ajouter** $A \longrightarrow \psi$ à $M[A,b]$.

Si ε est dans $First_1(\psi)$ et \$ dans $Follow_1(A)$, **alors** ajouter $A \longrightarrow \psi$ à $M[A,\$]$.

Faire de chaque entrée non définie de M une **erreur**.

Une grammaire LL(1) sera, par définition, une grammaire telle que chaque case de cette table M contienne au plus une règle. Sinon c'est qu'il y a des conflits et la grammaire n'est pas LL(1).

5.3.6 Analyseur LL(1) : exemple complet

Considérons la grammaire G suivante :

- $V_N = \{E,A,T,M,O\}$;
- $V_T = \{nb, +, *, (,)\}$;
- axiome : E ;
- règles de réécritures :

$E \longrightarrow TA$
$A \longrightarrow +TA$
$A \longrightarrow \varepsilon$
$T \longrightarrow OM$
$M \longrightarrow *OM$
$M \longrightarrow \varepsilon$
$O \longrightarrow (E)$
$O \longrightarrow nb$

5.3.6.1 Construction de la table

On se propose de construire la table d'analyse M. Il s'agit donc d'un tableau à deux dimensions. Chaque ligne correspondra à un symbole non-terminal et chaque colonne à un symbole terminal ou bien au symbole de fin de mot \$.

	nb	$+$	$*$	$($	$)$	\$
E	$E \longrightarrow TA$			$E \longrightarrow TA$		
A		$A \longrightarrow +TA$			$A \longrightarrow \varepsilon$	$A \longrightarrow \varepsilon$
T	$T \longrightarrow OM$			$T \longrightarrow OM$		
M		$M \longrightarrow \varepsilon$	$M \longrightarrow *OM$		$M \longrightarrow \varepsilon$	$M \longrightarrow \varepsilon$
O	$O \longrightarrow nb$			$O \longrightarrow (E)$		

FIG. 5.2 – *Table d'analyse de la grammaire G*

Construction de la table :

Règle $E \longrightarrow TA$ **:** $First_1(TA) = \{nb,(\}$. On écrit donc la règle $E \longrightarrow TA$ dans $M[E,nb]$ et dans $M[E,(]$.

Règle $A \longrightarrow +TA$ **:** $First_1(+TA) = \{+\}$. On écrit donc la règle $A \longrightarrow +TA$ dans $M[A,+]$.

Règle $A \longrightarrow \varepsilon$ **:** $First_1(\varepsilon) = \{\varepsilon\}$. $Follow_1(A) = \{),\$\}$. On écrit donc la règle $A \longrightarrow \varepsilon$ dans $M[A,)]$ et dans $M[A,\$]$.

Règle $T \longrightarrow OM$ **:** $First_1(OM) = \{(,nb\}$. On écrit donc la règle $T \longrightarrow OM$ dans $M[T,(]$ et dans $M[T,nb]$.

Règle $M \longrightarrow *OM$ **:** $First_1(*OM) = \{*\}$. On écrit donc la règle $M \longrightarrow *OM$ dans $M[M,*]$.

Règle $M \longrightarrow \varepsilon$ **:** $First_1(\varepsilon) = \{\varepsilon\}$. $Follow_1(M) = \{+,),\$\}$. On écrit donc la règle $M \longrightarrow \varepsilon$ dans $M[M,+]$, dans $M[M,)]$ et dans $M[A,\$]$.

Règle $O \longrightarrow (E)$ **:** $First_1((E)) = \{(\}$. On écrit donc la règle $O \longrightarrow (E)$ dans $M[O,(]$.

Règle $O \longrightarrow nb$ **:** $First_1(nb) = \{nb\}$. On écrit donc la règle $O \longrightarrow nb$ dans $M[O,nb]$.

La table d'analyse, représentée figure 5.2 est maintenant terminée.

5.3.6.2 Exemple d'analyse

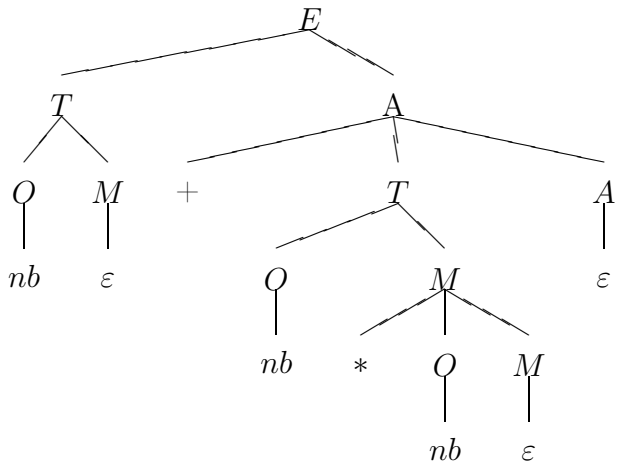
$ est utilisé dans le mot d'entré pour signaler la fin du mot, et dans la pile pour signaler le «fond» de la pile. Au début, la pile contient simplement l'axiome de la grammaire, juste au-dessus de $. La table d'analyse permet de sélectionner la règle à appliquer à chaque étape en fonction du sommet de la pile, P, et du symbole courant s.

Voici l'algorithme d'analyse :

1. Si $P = \$$ et $s = \$$ alors le mot a été reconnu : **SUCCES**.
2. Si $P = s$ et $s \neq \$$ alors le symbole s est accepté, l'analyseur enlève s de la pile et avance d'une case sur le ruban d'entrée.
3. Si P est un symbole non terminal, alors on consulte la table pour savoir quoi faire, en fonction de P et de s. Si la case $M[P,s]$ est vide, alors le mot n'est pas reconnu : **ECHEC**. Si $M[P,s] = P \longrightarrow \psi$ alors l'analyseur dépile P et empile la liste des symboles de ψ.

Le tableau de la figure 5.3 montre la suite des étapes de l'analyse du mot $nb + nb * nb$.

Le mot a donc bien été reconnu. La dernière colonne du tableau de la figure 5.3 permet de retracer la dérivation et donc de construire l'arbre de dérivation associé à ce mot :



Symboles acceptés du mot d'entrée	Etat de la pile (haut à gauche)	Symboles à traiter du mot d'entrée	Règle qui est choisie
	$E\$$	$nb+nb*nb\$$	$E \longrightarrow TA$
	$TA\$$	$nb+nb*nb\$$	$T \longrightarrow OM$
	$OMA\$$	$nb+nb*nb\$$	$O \longrightarrow nb$
	$nbMA\$$	$nb+nb*nb\$$	
nb	$MA\$$	$+nb*nb\$$	$M \longrightarrow \varepsilon$
nb	$A\$$	$+nb*nb\$$	$A \longrightarrow +TA$
nb	$+TA\$$	$+nb*nb\$$	
$nb+$	$TA\$$	$nb*nb\$$	$T \longrightarrow OM$
$nb+$	$OMA\$$	$nb*nb\$$	$O \longrightarrow nb$
$nb+$	$nbMA\$$	$nb*nb\$$	
$nb+nb$	$MA\$$	$*nb\$$	$M \longrightarrow *OM$
$nb+nb$	$*OMA\$$	$*nb\$$	$M \longrightarrow *OM$
$nb+nb*$	$OMA\$$	$nb\$$	$M \longrightarrow *OM$
$nb+nb*$	$OMA\$$	$nb\$$	$O \longrightarrow nb$
$nb+nb*$	$nbMA\$$	$nb\$$	
$nb+nb*nb$	$MA\$$	$\$$	$M \longrightarrow \varepsilon$
$nb+nb*nb$	$A\$$	$\$$	$A \longrightarrow \varepsilon$
$nb+nb*nb$	$\$$	$\$$	**SUCCES**

FIG. 5.3 – *Etapes de l'analyse du mot* $nb+nb*nb$

5.3.7 Conflits *First/First* et *First/Follow*

On se place dans le cadre d'analyseurs LL(1).

5.3.7.1 Conflits *First/First*

Si une grammaire contient les règles suivantes :

$V \longrightarrow n$
$V \longrightarrow T$
$T \longrightarrow n[e]$

Lorsque le sommet de la pile est V et le symbole courant n, on ne sait pas s'il faut utiliser la règle $V \longrightarrow n$, se qui revient à accepter le symbole n, ou s'il faut utiliser la règle $V \longrightarrow T$ se qui revient à continuer en attendant [.

En fait une telle grammaire n'est pas LL(1). Pour analyser une telle grammaire, il faudrait peut être un analyseur LL(2) qui aurait la possibilité de regarder 2 symboles en avant.

5.3.7.2 Conflits *First/Follow*

Si une grammaire contient les règles suivantes :

$S \longrightarrow Aab$
$A \longrightarrow a$
$A \longrightarrow \varepsilon$

Lorsque le sommet de la pile est A et le symbole courant a, on ne sait pas s'il faut utiliser la règle $A \longrightarrow a$ ou s'il faut utiliser la règle $A \longrightarrow \varepsilon$.

5.3.8 TD : analyse LL

5.3.8.1 Analyses

En utilisant la table d'analyse de la figure 5.2, effectuez une analyse des mots suivant :

- $nb * nb$;
- $nb + *nb$.

5.3.8.2 Construction de table

Construisez la table d'analyse de la grammaire suivante :

- $V_N = \{S\}$;
- $V_T = \{a,b\}$;
- axiome : S ;
- règles de réécritures :

$$S \longrightarrow aSa$$
$$S \longrightarrow bSb$$
$$S \longrightarrow \varepsilon$$

5.3.8.3 Encore une ?

Construisez la table d'analyse de la grammaire suivante :

- $V_N = \{S,A\}$;
- $V_T = \{a,b\}$;
- axiome : S ;
- règles de réécritures :

$$S \longrightarrow aBa$$
$$B \longrightarrow Bb$$
$$B \longrightarrow b$$

Rencontrez-vous un problème ?

Que peut-on changer dans cette grammaire pour résoudre le problème sans changer le langage qu'elle décrit ?

5.4 Analyse déterministe ascendante LR

5.4.1 Introduction

Les *procédures ascendantes*, qui constituent l'autre grande classe de procédures d'analyse ou de reconnaissance, fonctionnent de façon inverse aux procédures descendantes. Alors qu'un analyseur descendant va de la racine vers les feuilles de l'arbre, un analyseur

ascendant part des symboles du mot et tente de reconstruire l'arbre dérivationnel jusqu'à sa racine. Si les procédures descendantes simulent une dérivation la plus à gauche, les procédures ascendantes simulent une dérivation la plus à droite.

Nous présenterons ici, sans trop entrer dans les détails, ce que sont les *analyseurs LR(k)*, dont les lettres signifient respectivement :

- **L** pour *Left scanning*, c'est à dire analyse des symboles du mot de gauche à droite ;
- **R** pour *Right parsing* pour dérivation la plus à droite ;
- **(k)** pour k symboles (ou jetons ou *token* en anglais) de prévision (*lookahead* en anglais).

Les *analyseurs LR* sont des analyseur de langage hors-contexte très utilisés par les compilateurs de langage de programmation.

Une grammaire est dite LR(k) s'il existe un analyseur LR(k) capable de l'analyser. Les grammaires LR(k) sont un sous-ensemble des grammaires hors contexte. Cependant, tout langage hors-contexte non ambiguë peut être engendré par une grammaire LR(1). Les grammaires LR ont donc une capacité descriptive supérieure au grammaires LL.

Les grammaires LR possèdent de nombreux avantages :

- tout langage de programmation peut être analysé par un analyseur LR ;
- les analyseurs LR peuvent être implémentés de manière efficace ;
- de tous les analyseurs qui lisent les symboles du mot à analyser de gauche à droite, les analyseurs LR sont ceux qui détectent les erreurs de syntaxe le plus rapidement.

De plus, dans le cas d'applications où un traitement de mots syntaxiquement mal formées est souhaité (c'est souvent le cas pour les phrases en langage naturel), il est préférable d'utiliser des procédures ascendantes. En effet, les procédures descendantes fournissent souvent peu d'informations lorsque le mot à analyser est mal formé. Au contraire, une procédure ascendante livre toujours des analyses partielles lorsque le mot est mal formé : des constituants partiels auront été constitués même s'ils ne peuvent être combinés en une structure complète issue de l'axiome de la grammaire.

L'analyseur LR(0) est très simple et ses tables sont compactes, mais cette méthode n'est pas suffisante pour la plupart des grammaires. Un analyseur syntaxique LR(1) est efficace, mais cependant trop coûteux en terme de mémoire. Il existe toutefois d'autre méthodes voisines et plus économiques que LR(1) tout en gardant un pouvoir descriptif supérieur à LR(0). Il y a par exemple les méthodes Simple LR(k) ou SLR(k) et LookAhead LR(k) ou LALR(k).

Les analyseurs LR sont difficiles à réaliser à la main et sont habituellement construits par des générateurs d'analyseur. Aussi nous attacherons-nous essentiellement à comprendre ces analyseurs et non pas à les construire.

5.4.2 Analyse à décalage réduction

L'analyse ascendante consiste à partir de la phrase en entrée et à la parcourir, en réduisant chaque groupe de symboles correspondant à une partie droite de règle au symbole qui figure en partie gauche de la même règle. Lorsqu'on ne peut pas réduire, on avance sur la phrase d'entrée jusqu'à ce qu'une règle nous permette de faire une réduction. On a ainsi une alternance de mouvements de décalage (*shift* en anglais) et de réduction (*reduce* en anglais). D'où le nom d'analyseur « à décalage et réduction » (ou en anglais, *shiftreduce*).

Voici un exemple à portant sur une langue naturelle. Soit la grammaire G :

- $V_N = \{P,SN,SV,Det,N,V\}$;
- $V_T = \{aime,chat,chien,le\}$;
- axiome : P ;
- règles de réécritures :
  - (1) $P \longrightarrow SN\ SV$
  - (2) $SN \longrightarrow Det\ N$
  - (3) $SV \longrightarrow V\ SN$
  - (4) $Det \longrightarrow le$
  - (5) $N \longrightarrow chat$
  - (6) $N \longrightarrow chien$
  - (7) $V \longrightarrow aime$

Soit la phrase : *le chat aime le chien*.

L'analyse à décalage et réduction de cette phrase selon cette grammaire se fera suivant les étapes suivantes :

Pile	Reste	Action
\$	*le chat aime le chien* \$	Shift
le	*chat aime le chien* \$	Reduce 4
Det	*chat aime le chien* \$	Shift
Det chat	*aime le chien* \$	Reduce 5
Det N	*aime le chien* \$	Reduce 2
SN	*aime le chien* \$	Shift
SN aime	*le chien* \$	Reduce 7
SN V	*le chien* \$	Shift
SN V le	*chien* \$	Reduce 4
SN V Det	*chien* \$	Shift
SN V Det chien	\$	Reduce 6
SN V Det N	\$	Reduce 2
SN V SN	\$	Reduce 3
SN SV	\$	Reduce 1
P	\$	SUCCES

Ceci représente le cas le plus simple qu'on puisse avoir : l'état de la pile déterminent de manière unique l'action à effectuer. Pourtant, comme dans le cas de l'analyse descendante, on va rencontrer des situations où plusieurs choix possibles apparaissent à un moment donné.

Nous pouvons donc, comme dans le cas de l'analyse descendante, introduire une distinction entre les grammaires qui autorisent une analyse ascendante déterministe et celles qui ne l'autorisent pas. Nous appellerons, grammaires LR(0) les grammaires qui permettent une analyse ascendante déterministe sans regard en avant.

Comme dans le cas de l'analyse descendante, on pourra parfois obtenir une analyse déterministe en s'autorisant un regard en avant. Nous appellerons, grammaires LR(1) les grammaires qui permettent une analyse ascendante déterministe avec un regard en avant.

Nous pouvons maintenant nous demander s'il ne serait pas possible de fabriquer à partir de la grammaire une sorte d'automate qui nous permette la même analyse, mais sans avoir besoin de reconsulter les règles de grammaires à chaque étape : ces règles auraient été consultées une bonne fois pour toutes au moment de construire l'automate. C'est à cet objectif que répond la construction d'analyseurs LR.

5.4.3 Conflits *Shift/Reduce* et *Reduce/Reduce*

On se place dans le cadre d'analyseurs LR(0).

5.4.3.1 Conflits *Shift/Reduce*

Voici un petit exemple de grammaire qui n'est pas LR(0) :

(1) $S \longrightarrow aS$

(2) $S \longrightarrow a$

On est ici en présence d'un conflit *Shift/Reduce* :

Pile	Reste	Action
a	$a\$$	Shift ou Reduce 2?

5.4.3.2 Conflits *Reduce/Reduce*

Voici un petit exemple de grammaire qui n'est pas LR(0) :

(1) $A \longrightarrow c$

(2) $B \longrightarrow c$

On est ici en présence d'un conflit *Reduce/Reduce* :

Pile	Reste	Action
c	\$	Reduce 1 ou Reduce 2?

5.4.4 Analyseurs LALR(1)

Comme nous l'avons dit, nous ne nous intéresserons pas à la construction des table d'analyse LR et encore moins à la construction des table d'analyse LALR. Un logiciel, comme *GOLD Parser* par exemple, peu se charger, à notre place, de la construction de telles tables. Par contre, il reste important de comprendre ces tables.

Un analyseur LALR(1) utilise une pile. Au début, la pile contient l'état initial : 0. Pour connaître l'action à effectuer, il faut se référer à la table d'analyse LALR(1). Il y a quatre actions possibles en fonction du premier symbole du mot à analyser ou, le cas échéant, du symbole retourné par une action *Reduce*.

a **Shift** x **:** on retire le symbole a du mot à analyser et on le met dans la pile ; on empile également l'état x ; le nouvel état courant est l'état x.

a **Reduce** x **:** x désigne une règle de la forme $N \longrightarrow \psi$; on retire de la pile tout ce qui concerne ψ (les symboles de ψ et les états qui les suivent immédiatement) ; soit y l'état qui se trouve en haut de la pile à ce moment ; on empile N ; le nouvel état courant est l'état y.

A **Goto** x **:** quand une action Reduce vient d'être effectuée et que le symbole A est en haut de la pile, on empile x et le nouvel état courant est l'état x.

Accept : le mot est complètement analysé avec succès.

Ici, la pile est mixte, elle contient des états, des symboles terminaux et non-terminaux.

Prenons l'exemple de la grammaire suivante :

- $V_N = \{S\}$;
- $V_T = \{a,b\}$;
- axiome : S ;
- règles de réécritures :

  (0) $S \longrightarrow aSb$
  (1) $S \longrightarrow ab$

La table d'analyse LALR(1) correspondant à cette grammaire est la suivante :

Etat	**Action**
0	a Shift 1 S Goto 5
1	a Shift 1 b Shift 2 S Goto 3
2	\$ Reduce 1 b Reduce 1
3	b Shift 4
4	\$ Reduce 0 b Reduce 0
5	\$ Accept

L'analyse LALR(1) du mot *aabb* selon la grammaire que nous venons de voir et sa table associée se fera suivant les étapes suivantes :

Pile	**Reste**	**Action**
0	$aabb$\$	Shift 1
0a1	abb\$	Shift 1
0a1a1	bb\$	Shift 2
0a1a1b2	b\$	Reduce 1
0a1S	b\$	Goto 3
0a1S3	b\$	Shift 4
0a1S3b4	\$	Reduce 0
0S	\$	Goto 5
0S5	\$	**Accept**

5.4.5 TD : analyse LALR(1)

Soit la grammaire suivante :

- $V_N = \{S\}$;
- $V_T = \{a,b,c\}$;
- axiome : S ;

– règles de réécritures :

$S \longrightarrow aSa$
$S \longrightarrow bSb$
$S \longrightarrow c$

Un générateur d'analyseur à produit pour nous la table d'analyse suivante :

Etat	**Action**
0	a Shift 1 b Shift 2 c Shift 3 F Goto 8
1	a Shift 1 b Shift 2 c Shift 3 F Goto 4
2	a Shift 1 b Shift 2 c Shift 3 F Goto 6
3	\$ Reduce 2 a Reduce 2 b Reduce 2
4	a Shift 5
5	\$ Reduce 0 a Reduce 0 b Reduce 0
6	b Shift 7
7	\$ Reduce 1 a Reduce 1 b Reduce 1
8	\$ Accept

A l'aide de cette table, faite une analyse des mots suivant :

1. *aca* ;
2. *abcab* ;
3. *babcbab*

5.4.6 TP : Expressions arithmétiques

Soit la grammaire suivante :

– $V_N = \{E\}$;
– $V_T = \{nb\}$;
– axiome : E ;

– règles de réécritures :

$$E \longrightarrow E + E$$
$$E \longrightarrow E * E$$
$$E \longrightarrow nb$$

1. Quel est le langage décrit par cette grammaire ?
2. On désire que ce langage supporte le parenthèsage. Quelle règle faut-il rajouter pour cela ?
3. Le mot $nb + nb * nb$ est ambigu. Donnez les deux structures arborescentes associées à ses deux dérivations.
4. L'une de ces deux structures doit être rejetée. Pourquoi ?
5. Modifiez la grammaire pour que le mot $nb + nb * nb$ ne soit plus ambigu et que la structure arborescente associé à la dérivation de ce mot tienne compte de la priorité de l'opérateur de multiplication sur l'opérateur d'addition.
6. Ecrivez votre grammaire dans le logiciel *Gold Parser*. Observez les automates à état finis du *tokeniser*. Observez la table d'analyse. Effectuez des tests de la grammaire. Observez les structures arborescentes associées aux mots testés.

Bibliographie

Aho, A., & Ullman, J. (1992). *Concepts fondamentaux de l'informatique.* Paris: Dunod.

Alliot, J.-M., & Schiex, T. (1994). *Intelligence artificielle & informatique théorique.* Cépaduès éditions.

Gire, S. (2000). *Compilation théorique des langages.* http://fastnet.univ-brest.fr. (Deuxième année IUP Ingénierie Informatique Université de Bretagne Occidentale)

Lecomte, A. (2000). *Langages et communication.* http://brassens.upmf-grenoble.fr/-~alecomte/alaincours.html. (Cours pour le DESS Double Compétence "Informatique et Sciences Sociales")

Véronis, J. (2001). *Cours d'Informatique et linguistique 1 (INFZ18).* (Université de Provence, Centre Informatique pour les Lettres et Sciences Humaines)

Wehrli, E. (1997). *L'analyse syntaxique des langues naturelles problèmes et méthodes.* Paris Milan Barcelone: Masson.