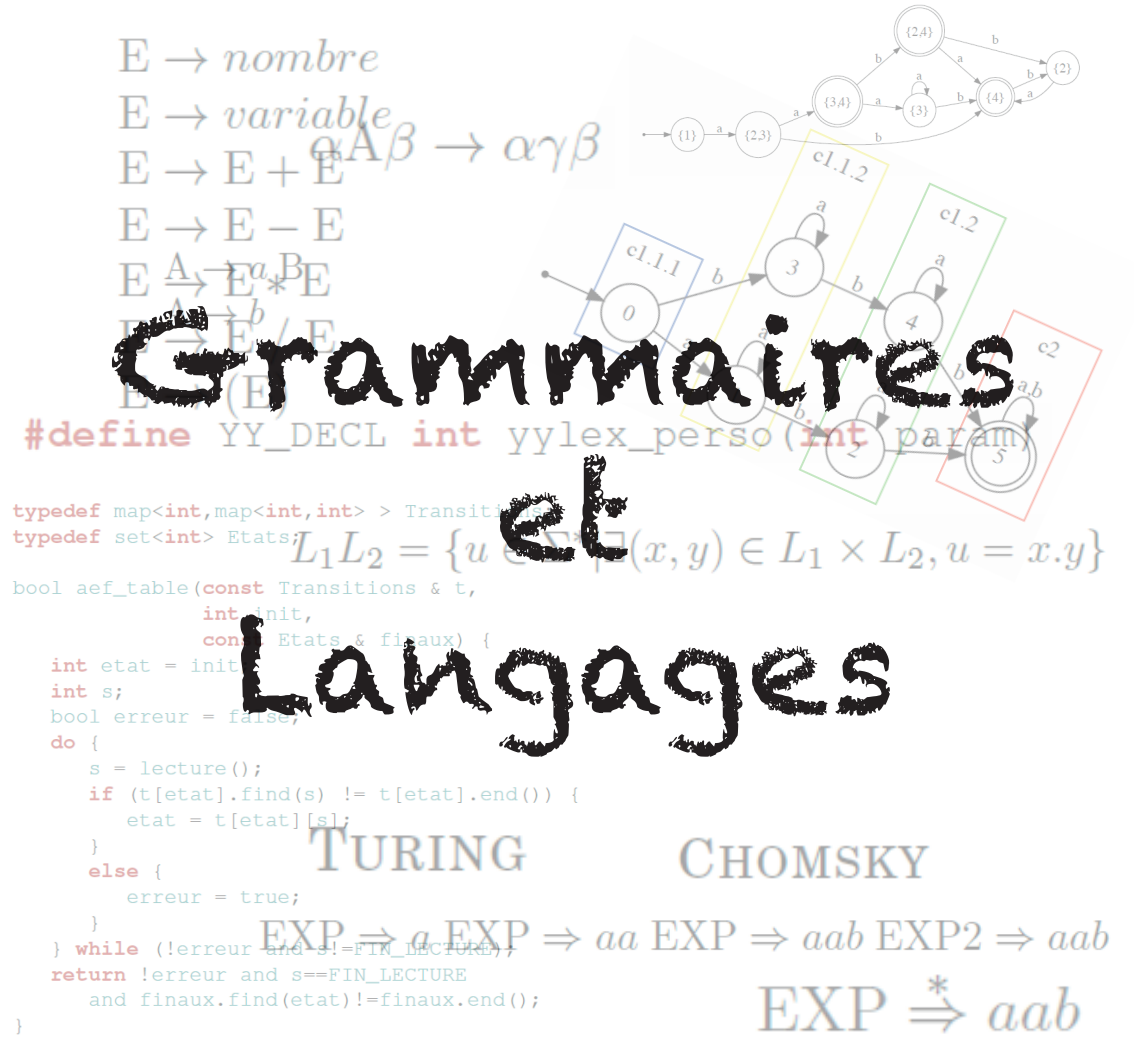
INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Année scolaire 2016-2017



Grammaires et Langages

Éric Guérin

23 janvier 2017

Année scolaire
2016-2017

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

23 janvier 2017

Éric Guérin

REMERCIEMENTS

Un grand merci aux relecteurs de ce polycopié, Nabila Benharkat et Jean-François Boulicaut.
Mes remerciements vont aussi vers tous les auteurs de cours sur le thème des grammaires et langages, ils ont été une source d'inspiration certaine (voir la bibliographie page 101 pour une liste complète).

TABLE DES MATIÈRES

CHAPITRE 1
INTRODUCTION

1.1 Introduction

Grammaires et langages sont des éléments primordiaux de la culture d'un informaticien. Les aspects théoriques font partie de ceux qui font de l'informatique une vraie discipline scientifique au même titre que la calculabilité par exemple. Les aspects pratiques vous permettront de mieux comprendre comment un langage de programmation est conçu et quelles sont les techniques qui permettent de le compiler. Mais les applications de ce cours ne se limitent pas à la compilation, et vous pourrez constater que les grammaires et les langages sont utilisés dans beaucoup de contextes.

▶ Contexte applicatif

Les notions de grammaires et langages sont utilisées dans toute une panoplie de contextes applicatifs :

- Compilation de langages de programmation
- Analyse automatique de documents
- Analyse et synthèse de la langue naturelle
- Transformation de modèles
- Synthèse ou analyse d'images
- Apprentissage automatique
- ...

Plus généralement, on peut distinguer trois domaines principaux d'application :

- Analyse
  Étant donnée une grammaire, on cherche à savoir si une phrase correspond à cette grammaire et éventuellement faire des traitements avec le résultat de l'analyse.
- Synthèse
  Étant donnée une grammaire, on cherche à produire des phrases de cette grammaire (donc appartenant au langage généré par la grammaire).
- Inférence grammaticale
  On possède un échantillonnage des éléments du langage. Le but est de construire la grammaire (avec des contraintes évidemment) qui permet de générer ce langage.

Dans tous les cas, la conception de grammaire est au cœur du dispositif. Une grammaire ne tombe jamais du ciel, il faudra à un moment ou à un autre y réfléchir et la concevoir. Ce cours vous y aidera.

▶ Contexte théorique

Il existe des résultats formels puissants qui relient grammaires, langages et expressions. Historiquement, ce sont les domaines applicatifs qui ont poussé les développements théoriques (création de compilateurs).
Ces résultats ont permis à de nouvelles générations de langages de programmation de naître, pour lesquels l'analyse ne nécessite pas une lecture trop en avance (grammaires dites hors-contexte qui ont des propriétés LL(1) ou LR(1) en général).

▶ Liens avec d'autres cours

Au fur et à mesure du cours vous pourrez constater les liens qu'il peut y avoir, de manière plus ou moins visible, entre ce cours et d'autres :
- Algorithmique
- Programmation
- Programmation logique
- Théorie des graphes
- Mathématiques discrètes
- Théorie des automates
- Synthèse et analyse d'images
- Modélisation (au sens conception)
- ...

▶ À l'INSA

Le module Grammaire et Langages est composé de :
- Ce cours (4 séances d'amphi, avec polycopié de cours)
- Les amphis liés au PLD-Comp (2 séances d'amphi, avec le même polycopié de cours)
- Un TD sur l'analyse LL et LR
- Un TP d'implémentation d'un automate LR
- Un PLD en hexanomes de 8 séances (programmation d'un compilateur du langage C en C++/Bison/flex)
- Un DS qui portera sur tout le module (cours, TD, projet)

Lorsque le cours est en rapport avec le PLD, l'icône PLD apparaîtra. Ce module fait partie de l'UE *développement logiciel* et proposera en plus des aspects théoriques systématiquement des implémentations en C++ ou prolog des algorithmes étudiés.
Ce support de cours reprend quasiment *verbatim* le contenu des transparents vus en amphi. Seuls quelques exemples ont été omis pour plus de concision.

▶ Compétences

Au sein du département et même de manière plus large, une réflexion sur les compétence a été faite[1]. Voici les compétences que vous allez acquérir avec ce module ou auxquelles ce module et le PLD vont contribuer (Cours/TD/projet) :
- Mettre en œuvre une chaîne complète pour analyser / transformer un langage
  - Concevoir, transformer et interpréter une grammaire formelle
  - Mettre en œuvre un analyseur lexical
  - Mettre en œuvre un analyseur syntaxique descendant
  - Mettre en œuvre un analyseur syntaxique ascendant
- Utiliser des diagrammes UML pour modéliser un objet d'étude
  - Concevoir un diagramme UML modélisant un objet d'étude
  - Vérifier la cohérence de différents diagrammes modélisant un même objet d'étude
- Concevoir l'architecture d'un logiciel orienté objet

1. Le référentiel des compétences complet se trouve sur Moodle

  - Structurer un logiciel en paquetages et classes faiblement couplés et fortement cohésifs
  - Utiliser des Design Patterns
- Mettre en œuvre un processus de contrôle qualité logicielle (V&V : Vérification et Validation)
  - Mettre en œuvre une démarche de test globale
  - Être capable d'écrire des programmes testables

1.2 Grammaires et langages

Grammaires formelles et langages sont étroitement liés. Les grammaires permettent de décrire grâce à des règles une syntaxe d'écriture. Les langages sont des ensembles de phrases d'un alphabet donné. Pour les grammaires, on ne parle pas d'alphabet mais de symboles terminaux mais il s'agit de la même chose.
Exemples de langages :

1. $\{a^*b^*\}$ une suite (éventuellement vide) de a suivie d'une suite (éventuellement vide) de b (langage régulier *cf.* p. 12)
2. $\{a^ib^i/i > 0\}$ une suite de i occurrences de a suivie d'une suite de i occurrences de b (langage non régulier dit hors-contexte *cf.* p. 11)
3. $\{a^ib^ic^i/i > 0\}$ une suite de i occurrences de a suivie d'une suite de i occurrences de b suivie d'une suite de i occurrences de c (langage contextuel *cf.* p. 11)

Dans ces exemples, l'alphabet est constitué des lettres a, b, et c.
On peut décrire de manière formelle à l'aide d'une grammaire les règles qui permettent de produire ces langages. Prenons l'exemple du langage $\{a^*b^*\}$, il peut être décrit par :

$$\begin{aligned}&\text{EXP} \rightarrow a\ \text{EXP} \mid b\ \text{EXP2} \mid \epsilon\\ &\text{EXP2} \rightarrow b\ \text{EXP2} \mid \epsilon\end{aligned}$$

où la flèche $\rightarrow$ désigne le symbole de *production*, la barre verticale $\mid$ désigne les alternatives et ϵ désigne une production vide.

▶ Notations

Les symboles d'un alphabet (symboles terminaux) sont notés par des lettres minuscules (exemples : a ou b). Les mots d'un alphabet sont en général notés eux aussi par des lettres minuscules (exemples : u ou v). Les symboles non terminaux sont notés en majuscule (exemple : EXP).

▶ Définitions

Proto-mot Un mot qui contient en même temps des symboles terminaux et non terminaux (par exemple aa EXP) est appelé *proto-mot*. Il est en général noté par une lettre grecque.

Dérivation La réécriture d'un proto-mot par l'application d'une règle de la grammaire est appelée dérivation. La dérivation simple se note avec la flèche double $\Rightarrow$. Une suite de dérivations se note $\overset{*}{\Rightarrow}$. Par exemple, avec la grammaire précédente, on a :

$$\text{EXP} \Rightarrow a\ \text{EXP} \Rightarrow aa\ \text{EXP} \Rightarrow aab\ \text{EXP2} \Rightarrow aab$$

et donc :

$$\text{EXP} \overset{*}{\Rightarrow} aab$$

Lorsque $\alpha \Rightarrow \beta$, on dit que β dérive de α.

Phrase générée par une grammaire Une phrase peut être générée par une grammaire si on peut la déduire grâce à une suite de dérivations de l'axiome de cette grammaire.

Langage associé à une grammaire Le langage associé à une grammaire est l'ensemble des phrases que la grammaire permet de générer par le processus de dérivation.

1.3 Motivations

Pourquoi un informaticien doit-il s'intéresser aux grammaires et aux langages ? C'est ce que nous allons essayer d'introduire de manière intuitive dans cette section.

▶ Exemple classique

Un exemple souvent donné est celui de l'analyse des expressions arithmétiques. Si on demande à un informaticien d'écrire une grammaire pour les expressions arithmétiques littérales, il va assez rapidement arriver à quelque chose dans ce goût là :

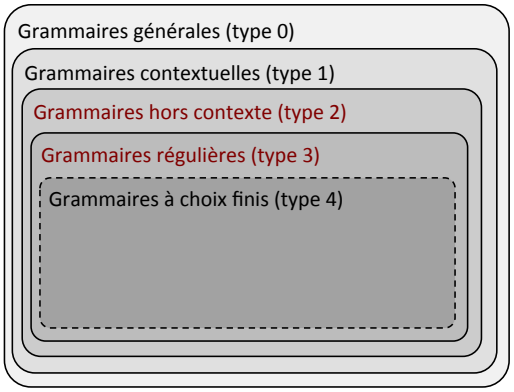
$$\begin{aligned}
&\mathrm{E} \rightarrow \mathit{nombre}\\
&\mathrm{E} \rightarrow \mathit{variable}\\
&\mathrm{E} \rightarrow \mathrm{E} + \mathrm{E}\\
&\mathrm{E} \rightarrow \mathrm{E} - \mathrm{E}\\
&\mathrm{E} \rightarrow \mathrm{E} * \mathrm{E}\\
&\mathrm{E} \rightarrow \mathrm{E} \,/\, \mathrm{E}\\
&\mathrm{E} \rightarrow (\mathrm{E})
\end{aligned}$$

C'est maintenant que les soucis commencent à apparaître...
En effet :

- Qu'est-ce qu'un nombre, ou une variable ? On n'a pas envie à ce niveau là de description de les définir de manière précise, car cela alourdirait fortement la grammaire. D'une manière intuitive, on sent bien que leur définition formelle est plus simple et ne nécessite pas une grammaire, mais quel outil ? La réponse est dans le chapitre 2.
- Si je considère l'expression très simple `3+a*5`, je peux construire deux dérivations avec la même grammaire. Laquelle est la bonne, comment lever automatiquement l'ambigüité ? La réponse est dans les chapitres 3, 4 et 5.
- Comment programmer un analyseur capable de dire si une expression est bien conforme à cette grammaire ? Une implémentation de type descente récursive se révèle catastrophique car peut engendrer une boucle infinie avec cette grammaire. Réponse aux chapitres 3, 4 et 5.
- Comment transformer cette grammaire pour qu'un analyseur performant puisse être généré ? Réponse dans les chapitres 3, 4 et 5.

1.4 Classification

Suivant les propriétés que les grammaires satisfont, on peut les classifier. En 1956, CHOMSKY a proposé une classification en 5 catégories de grammaires (et donc de langages). Chaque niveau hiérarchique est un sous-ensemble du niveau précédent :



Pour décrire cette hiérarchie, il convient d'utiliser les notations suivantes :

- T est l'ensemble des symboles terminaux de la grammaire ou alphabet (parfois noté Σ dans ce cas)
- N est l'ensemble des symboles non terminaux de la grammaire
- R est l'ensemble des règles de la grammaire
- S est l'axiome de la grammaire (le symbole de départ), $S \in N$
- V est l'ensemble des symboles terminaux et non terminaux : $V = T \cup N$

▶ Grammaires générales

Ce sont les grammaires les plus expressives de la hiérarchie, toutes les autres sont donc des cas particuliers de grammaires générales. Les règles ont la forme suivante :

$$\alpha \rightarrow \beta$$

avec $\alpha \in V^*NV^*$ et $\beta \in V^*$. D'une manière générale, ces grammaires sont extrêmement difficiles à analyser. En effet, il est possible grâce à une machine de TURING de savoir en un temps fini qu'une phrase appartient au langage généré par la grammaire. Par contre, si cette phrase n'appartient pas à ce langage, la machine va boucler sans donner de réponse. Le problème est donc indécidable.

▶ Grammaires contextuelles

Intuitivement, ces grammaires permettent de transformer un symbole dans un contexte, c'est-à-dire lorsque ce symbole est entouré d'autres symboles. Les règles sont de cette forme :

$$\alpha \mathrm{A} \beta \rightarrow \alpha \gamma \beta$$

avec $\mathrm{A} \in N$, $\alpha, \beta, \gamma \in V^*$ et $\gamma \neq \epsilon$.
Les langages générés par ces grammaires sont dits contextuels. Ils sont reconnus par une machine de TURING linéairement bornée (espace de stockage proportionnel à la taille des données d'entrée) non déterministe.
Il existe des résultats théoriques importants qui montrent que le complémentaire d'un langage contextuel est lui-même aussi contextuel. Les grammaires contextuelles sont aussi appelées croissantes car on peut voir que les dérivations ne peuvent pas diminuer la taille de la phrase (car on a $\gamma \neq \epsilon$).

▶ Grammaires hors-contexte

Il s'agit de grammaires contextuelles dépourvues de contexte (*sic* !) d'où leur nom. Dans la définition précédente, il suffit de supprimer α et β ce qui donne :

$$\mathrm{A} \rightarrow \gamma$$

avec $\mathrm{A} \in N$ et $\gamma \in V^*$.

La partie gauche contient un et un seul symbole non terminal. Une telle grammaire produit un langage hors-contexte, reconnaissable par un automate à pile.
Les grammaires hors-contexte sont très utilisées pour la définition de langages de programmation car leur analyse est aisée et surtout possible. La partie analyse syntaxique de ce cours montrera les différentes techniques qui permettent de générer un analyseur à partir de la grammaire.
Il existe plusieurs sous-catégories dans ces grammaires suivant la nature de ce que l'on trouve dans la partie droite des règles. Lorsque l'on trouve un seul symbole non terminal dans la partie droite, on parle de grammaire linéaire.

▶ Grammaires régulières ou rationnelles

Les grammaires régulières sont rarement explicitées sous forme de règles. On utilise plus généralement les expressions régulières voire les automates finis. Toutefois, il existe un contexte formel pour définir les règles d'une grammaire régulière :

$$\begin{aligned} \mathrm{A} &\rightarrow a\ \mathrm{B} \\ \mathrm{A} &\rightarrow b \end{aligned}$$

avec $\mathrm{A}, \mathrm{B} \in N$ et $a, b \in T$.

Les grammaires régulières sont très utilisées pour l'analyse lexicale, elles permettent de reconnaître dans une suite de lettres des mots ou des expressions simples qui rendent la phase d'analyse syntaxique plus aisée.
Il existe des résultats théoriques très forts permettant de les relier aux expressions régulières et aux automates finis. Ce sera l'objet du chapitre sur l'analyse lexicale.

▶ Grammaires à choix finis

Ces grammaires sont très simples, elles ne génèrent que des langages finis. Inversement, tous les langages finis peuvent être générés par une grammaire de ce type par une construction triviale. Les règles sont du type :

$$\mathrm{A} \rightarrow a$$

avec $\mathrm{A} \in N$ et $a \in T^+$.

1.5 Notation BNF

▶ Historique

Introduite dans la fin des années 50 par John Backus Peter Naur pour décrire le langage de programmation Algol 60, cette notation est encore très utilisée de nos jours. Elle permet de décrire des grammaires et signifie Backus Naur Form.

▶ Symboles

- `::=` est le symbole de définition d'une règle
- `|` est le symbole du ou
- `< >` permet d'indiquer qu'il s'agit d'un symbole non-terminal
- `[ ]` indique une partie facultative
- `{ }` permet de grouper des éléments entre eux
- `" "` les terminaux d'un seul caractère doivent être entre guillemets

▶ EBNF

Une version dite étendue de la BNF consiste à ajouter les symboles suivants :
- `?` en notation post-fixée permet d'indiquer que ce qui précède est facultatif
- `*` indique que ce qui précède peut être répété de 0 à n fois
- `+` indique que ce qui précède peut être répété de 1 à n fois

Attention, certaines grammaires exprimées en BNF ou EBNF font des entorses aux règles décrites précédemment car les grammaires ne sont pas destinées à être analysées de manière automatique. Cas le plus fréquent : on ne met pas les `< >` autour d'un non-terminal.

CHAPITRE 2
ANALYSE LEXICALE

2.1 Langages et expressions rationnels

On peut définir de manière inductive [1] les langages réguliers ou rationnels [2] ainsi que les expressions régulières ou rationnelles.

▶ Langages réguliers

Un langage régulier ou rationnel est défini par :
- $\{\epsilon\}$ et $\varnothing$ sont des langages rationnels
- $\forall a \in \Sigma$, $\{a\}$ est un langage rationnel
- si L_1 et L_2 sont des langages rationnels, alors $L_1 \cup L_2$, $L_1 L_2$ et L_1^* sont des langages rationnels

où * est l'étoile de KLEENE, un opérateur qui signifie un nombre quelconque (y compris 0) d'occurrences (concaténations) de son opérande.

⚠ Attention, la concaténation (ou le produit) de deux langages (notée par la juxtaposition de deux langages) est l'ensemble des concaténations possibles de chacun des éléments de chaque langage :

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

où le $x.y$ désigne la concaténation de x et y.

Par définition, l'ensemble des langages rationnels est clos par les opérations rationnelles d'union, concaténation et étoile de KLEENE. On peut aussi démontrer que c'est le plus petit ensemble de langages qui possède cette propriété.

▶ Expressions régulières

La définition des expressions régulières ou rationnelles est quasiment la même que celle des langages :
- ϵ est une expression rationnelle représentant le mot de longueur nulle
- $\varnothing$ est une expression rationnelle représentant l'absence de mot (à bien différencier de son précédent)
- $\forall a \in \Sigma$, a est une expression rationnelle
- si e_1 et e_2 sont des expressions rationnelles, alors $(e_1 + e_2)$, $(e_1 e_2)$ et (e_1^*) sont des expressions rationnelles où le $+$ signifie "ou" et où la concaténation est formalisée par la juxtaposition (c'est un produit)

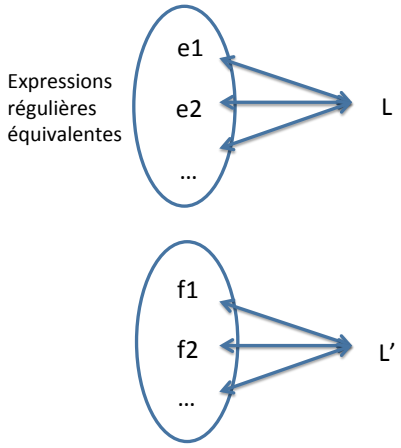
1. récursive
2. les deux termes sont utilisés

Quelques propriétés/règles simples mais utiles :
- $\epsilon e = e\epsilon = e$
- $\varnothing e = e\varnothing = \varnothing$
- Le $+$ est commutatif (pas la concaténation !)
- La concaténation et le $+$ sont associatifs
- Le $+$ se distribue sur la concaténation : $e_1(e_2 + e_3) = e_1e_2 + e_1e_3$
- Pour utiliser moins de parenthèses, on utilise la priorité d'opérateurs suivante *, concaténation, et $+$
- $\varnothing + e = e + \varnothing = e$
- $e + e = e$
- $e^*e^* = e^*$
- $(e^*)^* = e^*$
- *etc.*

▶ Équivalence

Langages et expressions régulières sont définis de manière équivalente. N'importe quelle expression régulière décrit un langage régulier et n'importe quel langage régulier peut être décrit par une expression régulière.

⚠ Attention cependant, cette correspondance n'est pas unique. Un langage régulier peut être décrit par des expressions régulières différentes. On dit d'ailleurs que deux expressions régulières sont équivalentes si elles décrivent le même langage. Par contre, une expression régulière engendre un et un seul langage.



L'exemple le plus simple pour illustrer cette propriété est le langage constitué du mot vide $\{\epsilon\}$ qui peut être décrit par les expressions régulières suivantes :
- ϵ
- $\epsilon + \epsilon$
- ϵ^*
- $\epsilon\epsilon$
- toute autre combinaison rationnelle des expressions précédentes

Un exemple un peu moins particulier, le langage correspondant à une suite (éventuellement de taille nulle) de a :
- a^*
- a^*a^*

— $a^* + a$
— $a^* + \epsilon$
— *etc.*

Cette notion de multiplicité de description d'un langage par une grammaire (ou ici une expression) se généralise à d'autres types de langages.

2.2 Automates finis

Nous allons voir dans cette section comment les automates finis (déterministes ou non) sont un moyen facile et efficace d'analyser et reconnaître des langages et expressions réguliers.

▶ Définition

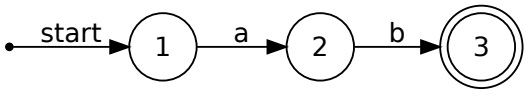
Un automate fini [3] déterministe est défini par :
— Σ un ensemble fini de symboles (appelé alphabet)
— Q un ensemble fini d'états
— $q_0 \in Q$ l'état initial
— $F \subset Q$ l'ensemble des états finaux
— δ une fonction totale [4] de $Q \times \Sigma$ dans Q, appelée fonction de transition.

Le qualificatif *déterministe* correspond au fait que chaque couple $(q, a) \in Q \times \Sigma$ ne possède qu'une valeur dans la fonction de transition.

⚠ Dans la pratique, on ne va pas exprimer la totalité des transitions possibles et simplement considérer que lorsqu'une transition n'est pas explicitée, elle renvoie vers un état d'erreur

▶ Représentation graphique

Un automate fini déterministe peut être représenté sous la forme d'un graphe orienté pour lequel les nœuds sont les états de l'automate, et les arcs représentent les transitions.
Si $\delta(p, a) = q$ alors on fera un arc du nœud p vers le nœud q avec l'étiquette a. L'état initial ainsi que les états finaux sont indiqués par des nœuds visuellement différents.
Lorsqu'une transition non explicite renvoie vers un état d'erreur on ne la fera pas apparaître sur le graphe pour ne pas le surcharger.
Exemple d'automate fini déterministe très simple :
— $\Sigma = \{a, b\}$
— $Q = \{1, 2, 3\}$
— $q_0 = 1$
— $F = \{3\}$
— $\delta(1, a) = 2$ et $\delta(2, b) = 3$.



▶ Langage reconnu

Un automate déterministe peut être utilisé pour reconnaître les mots d'un langage d'un alphabet Σ.
Prenons un mot $\alpha = e_1 \dots e_n \in \Sigma^*$. On dit que α est reconnu par l'automate $A = (\Sigma, Q, q_0, F, \delta)$ si $\delta^*(q_0, \alpha) \in F$ avec :

$$\begin{cases} \delta^*(q, u) = \delta(q, u) \text{ si } |u| = 1 \\ \delta^*(q, au) = \delta^*(\delta(q, u), a) \text{ sinon} \end{cases}$$

3. aussi appelé automate à états finis
4. domaine de définition = ensemble de départ

δ^* correspond donc à l'application successive de la fonction de transition δ aux symboles successifs d'un mot.
Inversement, si $\delta^*(q_0, \beta) \notin F$, on dit que β n'est pas reconnu par l'automate.
L'ensemble des mots reconnus par un automate fini A est appelé langage reconnu par cet automate $L(A)$:

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

▶ Table de transitions

On peut symboliser (et même stocker) les transitions d'un automate fini sous la forme d'un tableau à deux dimensions. Les lignes représentent les états de départs de transitions, les colonnes indiquent les symboles et les valeurs les états de destination de la transition.
Avec l'automate précédent, on obtient le tableau suivant :

δ	a	b
1	2	
2		3
3		

Les transitions non explicites sont donc des cases vides dans cette représentation (et mènent vers un état d'erreur).

▶ Méthode des dérivées

Il existe une méthode directe pour trouver un automate fini déterministe à partir de son expression régulière. Cette méthode consiste à dériver l'expression par rapport à un symbole. La définition de la dérivée est très simple et intuitive :

$$D_a(e) = \{u | au \in e\}$$

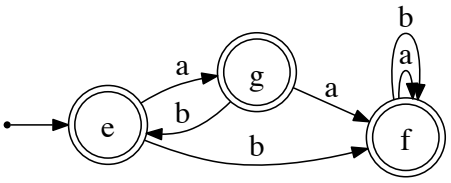
Autrement dit, la dérivée de l'expression e par rapport au symbole a est l'expression de ce qui peut suivre le symbole a dans e.
Cela implique les propriétés suivantes :

- $D_a(a) = \epsilon$
- $D_a(b) = \varnothing$
- $D_a(e_1 + e_2) = D_a(e_1) + D_a(e_2)$
- $D_a(e_1 e_2) = D_a(e_1)e_2 + \Delta(e_1)D_a(e_2)$
- $D_a(e^*) = D_a(e)e^*$

où $\Delta(e) = \epsilon$ si e peut être de longueur nulle et $\Delta(e) = \varnothing$ sinon.
Les états de l'automate sont les dérivées successives de l'expression, et il y a une transition de e_i vers e_j sur le symbole a si $D_a(e_i) = e_j$. Un état sera final si le langage qu'il génère contient le mot vide.
Exemple avec l'expression régulière $e = (ab)^* + (a + b)^*$.

- $D_a(e) = b(ab)^* + (a + b)^* = g$
- $D_b(e) = (a + b)^* = f$
- $D_a(g) = (a + b)^* = f$
- $D_b(g) = (ab)^* + (a + b)^* = e$
- $D_a(f) = (a + b)^* = f$
- $D_b(f) = (a + b)^* = f$

Nous arrivons à un automate à seulement trois états e, f, et g :



Exercice : simplifier l'automate ainsi que l'expression.

▶ Non-déterministe

La méthode des dérivées est un peu complexe à mettre en œuvre. Nous allons voir ici comment générer de manière très simple un automate fini à partir de son expression régulière. Cet automate sera non-déterministe.
La différence se situe dans la fonction de transition.
Un automate fini non-déterministe est défini par :

- Σ un ensemble fini de symboles (appelé alphabet)
- Q un ensemble fini d'états
- $q_0 \in Q$ l'état initial [5]
- $F \subset Q$ l'ensemble des états finaux
- δ une fonction $Q \times \Sigma$ dans 2^Q, appelée fonction de transition.

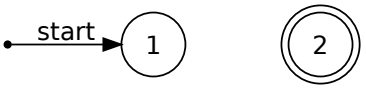
Contrairement à la version déterministe, la transition associée à un couple état-symbole mène vers un ensemble d'états (sous-ensemble de Q). Les automates finis non-déterministes sont donc une généralisation des automates finis déterministes. Même si cela paraît étrange, un automate déterministe est donc non-déterministe (la réciproque est fausse évidemment).
On peut utiliser une version plus pratique des automates finis non-déterministes qui consiste à ajouter des transitions spontanées c'est-à-dire dont le symbole associé est le mot vide ϵ. Cette définition n'ajoute pas d'expressivité mais elle est très importante pour la suite du cours.
Un automate fini non-déterministe avec ϵ-transitions est défini par :

- Σ un ensemble fini de symboles (appelé alphabet)
- Q un ensemble fini d'états
- $q_0 \in Q$ l'état initial
- $F \subset Q$ l'ensemble des états finaux
- δ une fonction $Q \times (\Sigma \cup \{\epsilon\})$ dans 2^Q, appelée fonction de transition.

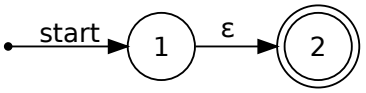
▶ Construction de Thompson

Nous allons voir comment il est possible de construire un automate fini non-déterministe qui reconnait une expression régulière. Cette méthode permet de produire des briques d'automates ayant un seul état initial mais aussi un seul état final.
Les expressions régulières sont définies de manière inductive (concaténation, union et étoile de KLEENE).
Commençons par voir comment on peut construire les automates des briques de base :

- Ensemble vide $\varnothing$

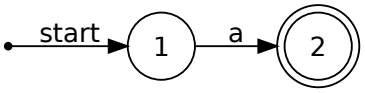


- Mot vide $\{\epsilon\}$



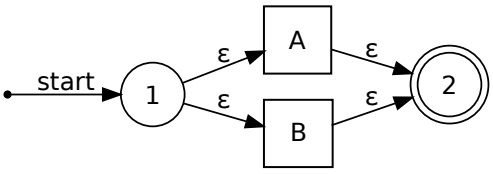
- Singleton $\{a\}$

5. parfois on considère un ensemble d'états initiaux mais cela n'apporte rien

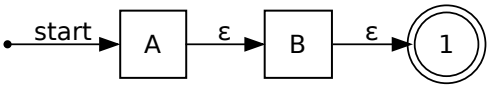


Nous pouvons maintenant construire les automates qui vont reconnaître des expressions plus complexes :

— Union $e_1 + e_2$ (A étant l'automate de e_1 et B celui de e_2)



— Concaténation e_1e_2 (A étant l'automate de e_1 et B celui de e_2)



— Opérateur de KLEENE : laissé en exercice

▶ Théorème de KLEENE

Il est assez aisé de voir que le langage reconnu par l'automate de Thompson est le même que celui engendré par l'expression régulière qui a servi à le construire.
Des résultats théoriques montrent qu'il y a une équivalence entre les automates finis et les expressions régulières. Le théorème de KLEENE transpose ce résultat aux langages :

Un langage est reconnaissable par un automate fini si et seulement s'il est rationnel (ou régulier).

2.3 Implémentation

Nous allons voir dans cette section comment il est possible d'implémenter un automate fini.

▶ Déterministe

Un automate fini déterministe peut être implémenté de manière totalement procédurale, sans mémoire. Pour cela il suffit de programmer une fonction par état et suivant le symbole lu, appeler la fonction correspondant à l'état cible de la transition.
Version purement algorithmique

```
bool etat1(void) {
  switch (lecture()) {
    case SYMBOLE1:
      return etat2();
    case SYMBOLE2:
      return etat5();
      ...
    case FIN_LECTURE:
      return true; // ou false si l'état 1
                   // n'est pas final
    case defaut:
      return false; // transition non prévue
  }
}
```

⚠ L'implémentation ci-dessus est mauvaise car elle oblige à faire un appel récursif pour chaque symbole lu. Une version plus heureuse serait d'imbriquer deux `switch`, l'un pour les états le deuxième pour les symboles lus et de n'avoir ainsi qu'une seule fonction.

Une autre façon de voir les choses est d'utiliser une table de transitions. Imaginons que l'état soit codé sous forme d'un entier, que les symboles lus sont des entiers, on peut coder cette table de la manière suivante :

```
typedef map<int,map<int,int> > Transitions;
```

Maintenant, une seule procédure va permettre d'implémenter l'automate fini, mais elle va utiliser de la mémoire (pour la table et pour l'état courant au minimum).
Version avec table

```
typedef map<int,map<int,int> > Transitions;
typedef set<int> Etats;

bool aef_table(const Transitions & t,
               int init,
               const Etats & finaux) {
   int etat = init;
   int s;
   bool erreur = false;
   do {
      s = lecture();
      if (t[etat].find(s) != t[etat].end()) {
         etat = t[etat][s];
      }
      else {
         erreur = true;
      }
   } while (!erreur and s!=FIN_LECTURE);
   return !erreur and s==FIN_LECTURE
      and finaux.find(etat)!=finaux.end();
}
```

L'avantage de cet algorithme est qu'il est complètement générique, il sera le même quel que soit l'automate.
Ces algorithmes sont donc très faciles à implémenter. Le seul souci, c'est que lorsqu'on dispose d'une expression régulière, il est difficile de construire de manière automatique un automate fini **déterministe**.

▶ Non-déterministe

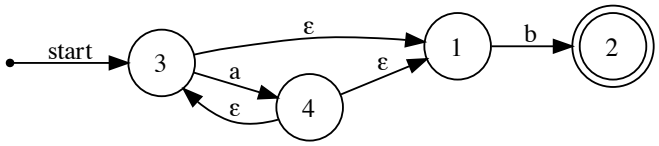
L'implémentation d'un automate fini non-déterministe est bien moins aisée et intuitive. En effet, la lecture d'un symbole ne permet pas de décider de l'état suivant. La structure de données permettant de stocker ces transitions devient :

```
typedef set<int> Etats;
typedef map<int,map<int,Etats> > NDTransitions;
```

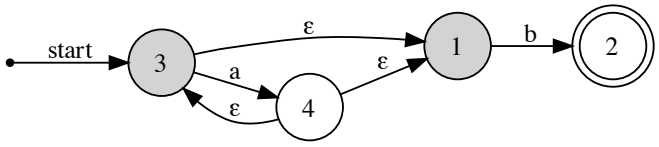
(variante possible avec une `multimap`)
Il y a deux manières d'implémenter cet automate :

- Exploratoire : on commence par explorer une des transitions possibles, si cette transition mène à une impasse (un état non final), on essaie la suivante. On remarquera que jusqu'à la fin de la lecture, on ne sait pas si le chemin est le bon ou non (sauf dans le cas d'un symbole n'ayant pas de transition associée). Si aucune des explorations ne mène à un état final, alors le mot n'est pas reconnu par l'automate. Si un chemin mène à un état final, alors le mot est reconnu.
- De front : la méthode consiste à stocker un ensemble d'états courants, et à les mettre à jour à chaque symbole lu. Si à la fin de la lecture, un des états courants est final, alors le mot est considéré comme reconnu par l'automate.

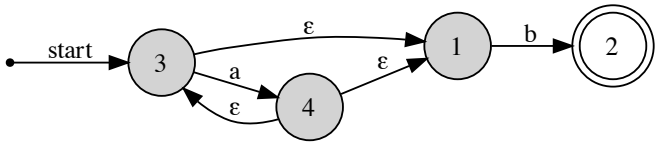
On voit clairement que la complexité d'exécution d'un automate fini déterministe est bien moindre que dans la version non-déterministe. Nous ne donnerons ici qu'une illustration graphique de la deuxième méthode. Illustration avec l'automate fini associé à l'expression régulière a^*b. L'automate de THOMPSON associé (ici généré automatiquement donc non optimisé) est le suivant :



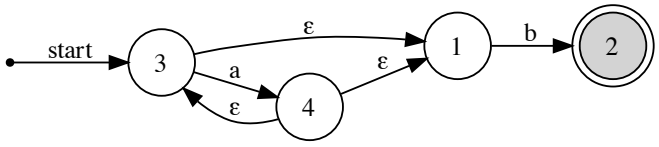
À l'initialisation, voici les états courants :



Après lecture d'un ou plusieurs a, voici les états courants :



Après lecture d'un b :



En cas d'erreur (aucune transition trouvée), il y a deux stratégies :
- Mettre fin à l'analyse et reporter l'erreur
- Garder en mémoire l'erreur et ne pas mettre à jour les états (une sorte de récupération sur erreur)
- Peut-être que l'on se trouve dans un cadre plus large d'analyse et que cette absence de transition indique simplement qu'il faut s'arrêter d'analyser et renvoyer un jeton lexical

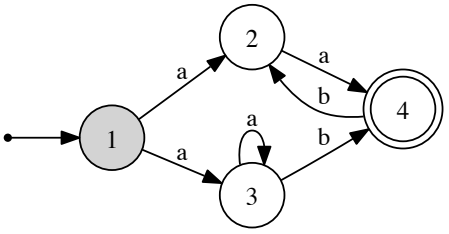
▶ Déterminisation

Afin de réduire la complexité des algorithmes d'analyse d'expression régulières, il convient de transformer les automates finis non-déterministes en automates finis déterministes. C'est ce que l'on appelle l'opération de déterminisation d'un automate.

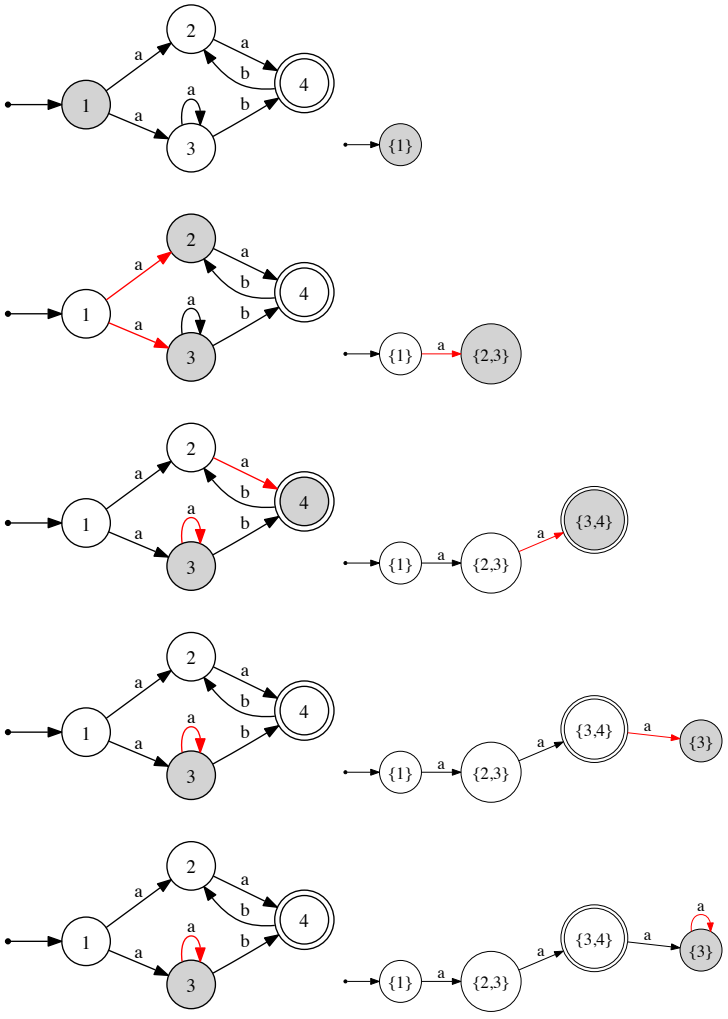
Le principe est lié à la deuxième version de l'implémentation d'un automate fini non-déterministe : chaque état du nouvel automate déterministe est un sous-ensemble d'états de l'automate non-déterministe. L'algorithme consiste à partir de l'état initial et à appliquer toutes les transitions possibles. Le résultat de chaque transition est un nouvel ensemble d'états. Il faut appliquer l'algorithme du point fixe à cet algorithme et donc s'arrêter lorsque l'ensemble des nouveaux états et transitions ne change plus.

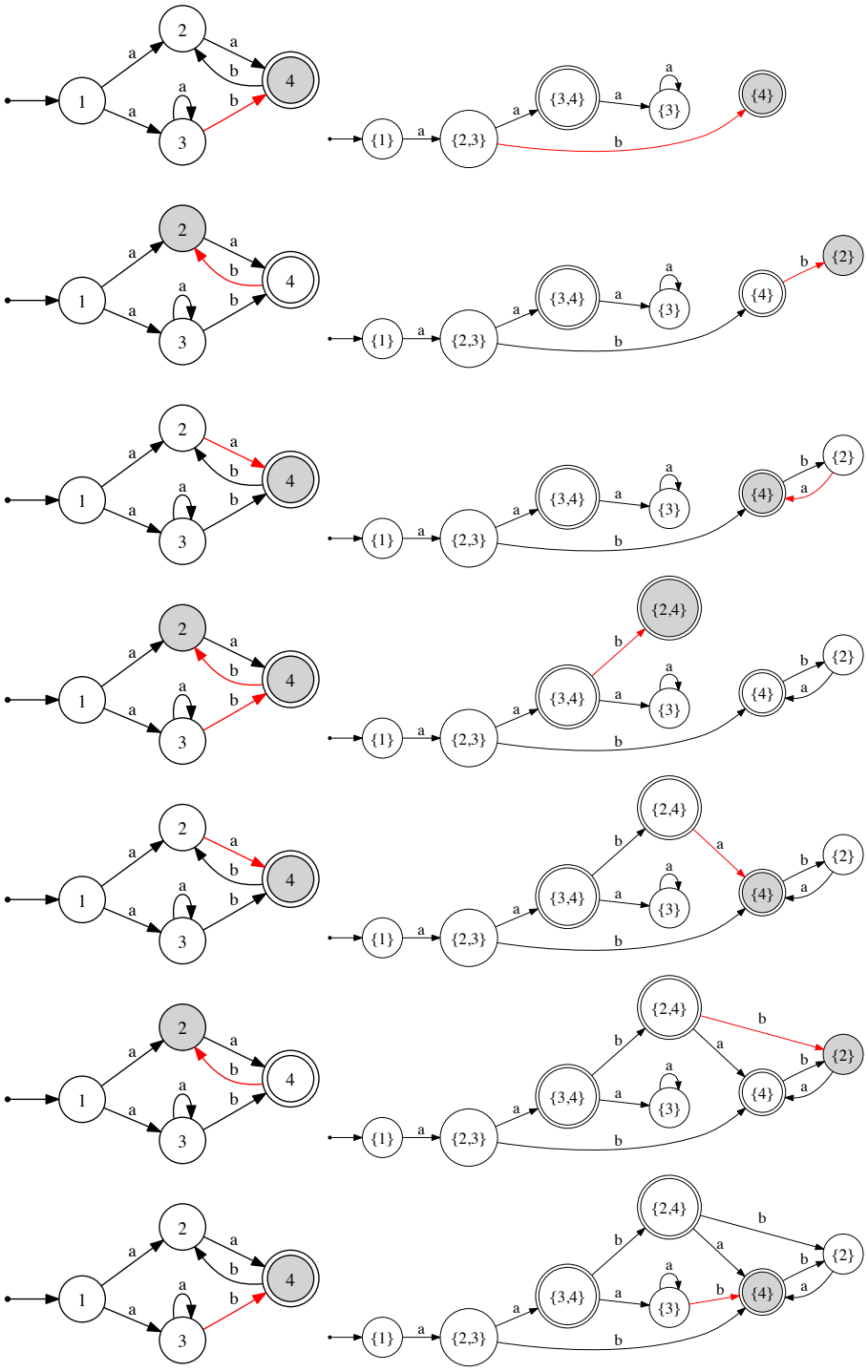
⚠ Le nombre d'états du nouvel automate peut être vite grand car $Q_d = 2^{Q_{\text{nd}}}$ donc $|Q_d| = 2^{|Q_{\text{nd}}|}$. Si l'automate non-déterministe a seulement 4 états, la version déterministe peut en avoir jusqu'à 16 !

Afin d'expliquer cet algorithme, rien de tel qu'un exemple !
Voici l'automate non-déterministe sur lequel sera basé l'exemple :

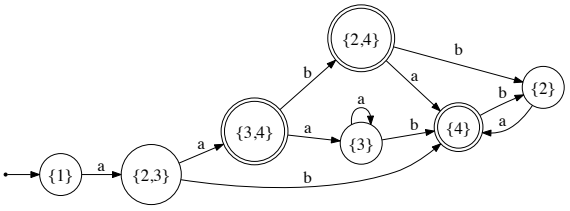


L'automate déterministe commence avec un état initial uniquement constitué du singleton $\{1\}$. Tous les nouveaux états qui contiennent l'état 4 sont des états finaux.



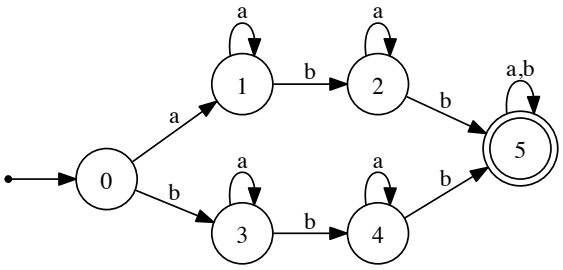


Automate final à 7 états

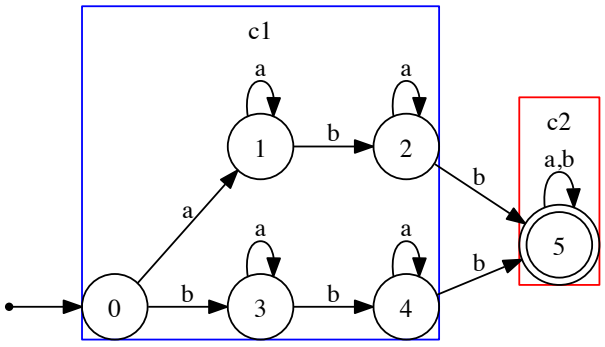


Minimisation

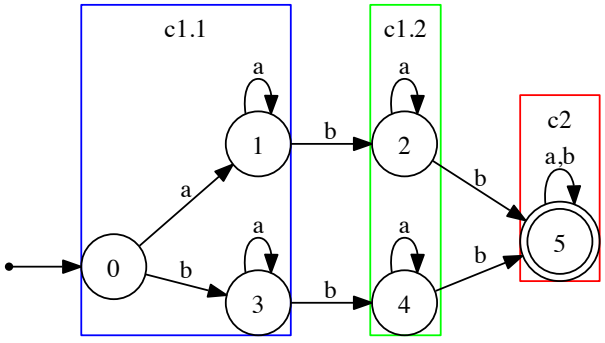
On vient de voir que la phase de déterminisation d'un automate peut générer un grand nombre d'états dont certains sont peut-être inutiles. La minimisation d'un automate fini déterministe consiste à regrouper certains états entre eux. Deux états peuvent être regroupés si toutes leurs transitions mènent vers le même groupe d'état.
L'algorithme va procéder inversement, c'est-à-dire qu'il va commencer avec une partition initiale simple (finaux/non-finaux) et séparer les ensembles lorsque la condition n'est pas satisfaite.



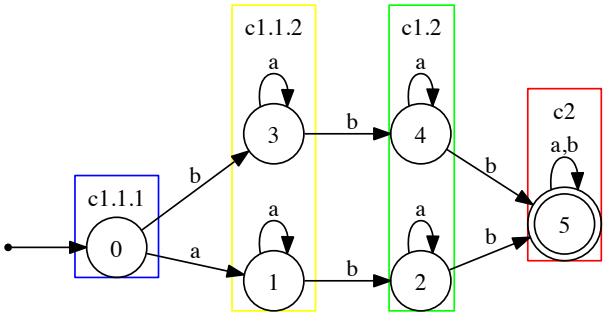
Initialisation avec l'état final 5 et les autres dans deux groupes :



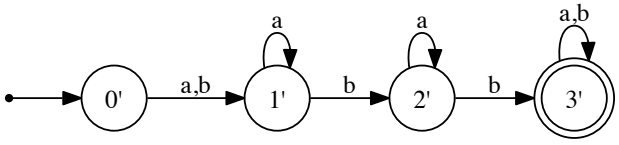
Sur la transition b, les états 2 et 4 sortent de c1 alors que les états 0, 1 et 3 y restent, il faut donc séparer en deux groupes :



Sur la transition b, les états 1 et 3 sortent de c1.1 alors que l'état 0 y reste, il faut donc séparer en deux groupes :



Tous les états satisfont maintenant les propriétés. Après fusion des états, on obtient :



▶ prolog

Il est très facile d'implémenter un automate fini en prolog. Les symboles sont représentés dans une liste. Au fur et à mesure que l'on fait des transitions, on enlève les éléments de la liste. Version déterministe (accepteur de $(a+b)cd$)[6] :

```
automate([X|Y],E,Y,F) :-
   transition(E,X,F).
accepte([],F) :-
   member(F,[4]).
accepte(X,E) :-
   automate(X,E,Y,F),
   accepte(Y,F).
transition(1,a,2).
transition(1,b,2).
transition(2,c,3).
transition(3,d,4).
```

Le gros avantage est que l'on peut aussi bien tester une liste pour voir si elle est reconnue par l'automate que de questionner prolog pour avoir des listes reconnues par l'automate :

```
?- accepte(X,1).
X = [a,c,d] ? ;
X = [b,c,d] ? ;
no
?- accepte([a,c],1).
no
```

Pour en faire un automate non-déterministe, il suffit d'ajouter les ϵ-transitions et les transitions multiples (ici l'accepteur de $(a+b)^*cd$)[7] :

```
automate([X|Y],E,Y,F) :-
   transition(E,X,F).
automate(X,E,X,F) :-
   etransition(E,F).
accepte([],F) :-
```

6. Exercice : faire l'automate de cette expression régulière
7. Exercice : faire l'automate de cette expression régulière

```
   member(F,[4]).
accepte(X,E) :-
   automate(X,E,Y,F),
   accepte(Y,F).
transition(1,a,2).
transition(1,b,2).
transition(2,c,3).
transition(3,d,4).
etransition(1,2).
etransition(2,1).
```

De la même manière que précédemment, on peut utiliser prolog pour tester ou demander les listes reconnues :

```
?- accepte(X,1).
X = [a,c,d] ? ;
X = [a,a,c,d] ? ;
X = [a,a,a,c,d] ? ;
X = [a,a,a,a,c,d] ?
yes
?- accepte([a,b,c,d],1).
true ?
yes
```

Le seul souci, c'est que prolog ne renvoie de manière évidente que les listes commençant par des a mais il est toutefois capable de reconnaître les autres si on lui fournit. Pour changer ce comportement dans prolog, il faut écrire un méta-interpréteur mais on sort du cadre de ce cours.

▶ Conclusion

La construction d'un automate fini non-déterministe à partir d'une expression régulière est simple et directe. Un automate fini non-déterministe possède une représentation plutôt concise (avec peu d'états) mais souffre d'une lenteur d'exécution à cause du non-déterminisme. On peut transformer un automate fini non-déterministe en automate fini déterministe par l'opération de déterminisation. Cette opération fait potentiellement augmenter le nombre d'états de manière exponentielle. La complexité d'analyse est plus faible mais l'espace nécessaire au stockage est plus élevé. L'opération de déterminisation s'apparente à l'exécution (exhaustive) de l'automate fini non-déterministe, on peut donc dire avec certitude que si une seule analyse doit être effectuée, la déterminisation ne sert à rien. Il existe des méthodes pour construire directement un automate fini déterministe à partir d'une expression régulière mais il s'agit d'une construction basée sur du calcul formel (dérivation d'expression) donc difficile à implémenter. Afin d'optimiser le nombre d'états d'un automate fini déterministe, on peut recourir à une opération de minimisation, qui va permettre de diminuer le nombre d'états de l'automate.

	Non-dét.	Dét.
Facilité de construction	✔	✘
Rapidité d'exécution	✘	✔
Espace mémoire	✔	✘

2.4 Outils

Il existe une multitude d'outils qui utilisent les expressions régulières pour décrire, transformer, rechercher des textes ou portions de textes.

▶ Extensions

La plupart des outils proposent un certain nombre d'extensions aux opérations rationnelles standard de concaténation, union et étoile de Kleene. Ces extensions (appelées *extended regular expressions* ou EREs) n'ajoutent rien au pouvoir d'expression mais sont uniquement là pour simplifier l'écriture :

- Classes de caractères
  - L'opérateur `[...]` permet de donner une liste de caractères autorisés ou non autorisés si la liste commence par `^`
  - Certaines classes de caractères sont pré-déclarées (extension POSIX), comme `[:alpha:]` ou `[:digit:]`
  - Le `.` indique n'importe quel caractère
- Cardinalités
  - `?` indique une cardinalité de 0 ou 1 donc un élément optionnel
  - `+` indique une cardinalité de 1 à n
  - `{n1,n2}` indique une cardinalité de $n1$ à $n2$ (attention, cette extension peut générer de très gros automates)

Attention, certaines extensions peuvent conduire à la perte de la propriété d'expression régulière. Plusieurs implémentations récentes des expressions régulières (en Python, PHP, .Net notamment) permettent de faire référence dans une expression régulière à une portion d'une expression déjà *matchée* précédemment. Ces expressions d'une manière théorique sont normalement extrêmement difficiles à analyser car elles mettent en jeu des langages qui sont généraux !

▶ POSIX

Le monde unix possède toute une série d'outils en ligne de commande permettant de filtrer, rechercher, remplacer des chaînes de caractères.

- `grep` permet de filtrer les lignes qui correspondent à une expression régulière donnée
- `ed` un (vieil) éditeur de texte qui permet en mode commande de faire des modifications à l'aide d'expressions régulières
- `sed` un éditeur de flux (*Stream EDitor*) qui permet de lire un fichier texte en entrée et d'associer des commandes lorsqu'une expression régulière correspond
- `awk` un éditeur/modificateur de flux de texte semblable à `sed` mais plus puissant encore
- `vi` un éditeur de texte dans lequel certaines commandes d'édition ou de recherche sont paramétrées par des expressions régulières

Une bibliothèque standard POSIX permet d'utiliser les expressions régulières en deux temps :

- `regcomp()` : compilation de l'expression régulière (fabrication de l'automate)
- `regexec()` : permet de tester si une chaîne satisfait l'expression régulière (exécution de l'automate)

Exemple d'utilisation :

```
regex_t preg;
string chaine;

regcomp(&preg,"(a|b)*ab",REG_EXTENDED);

cout<<"Saisissez une chaine"<<endl;
cin>>chaine;
if (!regexec(&preg,chaine.c_str(),0,0,0)) {
   cout<<"Conforme"<<endl;
} else {
   cout<<"Non conforme"<<endl;
}

regfree(&preg);
```

▶ C++11

Le standard C++ connu sous le nom de C++11 (11 pour 2011) inclut une bibliothèque standard pour la gestion des expressions régulières. Il est entièrement intégré à la STL avec la manipulation des `string`. Il inclut la possibilité de faire des recherches et remplacements. Récent, il n'est supporté que par les dernières versions de compilateurs, il est important de noter que son implémentation dans g++/stdlibc++ est incomplète et buguée. Préférer Clang/libc++ ou Visual Studio à partir de la version 2012.

2.5 Flex PLD

Voici quelques éléments qui permettent de comprendre le fonctionnement de Flex.

▶ Historique

lex/Flex permet de faire de l'analyse lexicale
- Origine de lex, années 70, UNIX
- Flex signifie fast lex, version GNU de lex
- Analyse lexicale : langages réguliers, permet d'identifier les mots d'un langage

yacc/Bison permet de faire de l'analyse syntaxique
- Origine de yacc, années 70, UNIX (Yet Another Compiler Compiler)
- Bison est la version GNU de yacc (jeu de mot)
- Analyse syntaxique : interprétation de grammaires non contextuelles, permet de reconnaître les phrases d'un langage

En pratique, rien n'oblige à utiliser les deux ensemble, mais cela simplifie pas mal de choses

▶ Fichier en entrée

Voici la structure d'un fichier Flex :

```
définitions
%%
règles
%%
code
```

▶ Définitions

Les définitions sont des lignes contenant deux parties :
- Un nom
- Une définition associée

Sur chaque ligne, on associe une définition sous la forme d'expression régulière à un nom (une sorte d'identifiant).
Exemples :

```
chiffre [0-9]
ident   [a-z][a-z0-9]*
```

Une définition peut ensuite être utilisée dans une autre grâce aux accolades :

```
entier  {chiffre}+
```

Cette même notation sera utilisée ensuite dans la partie règles.

▶ Règles

Les règles sont utilisées pour indiquer des actions (du code) à exécuter lorsque des expressions régulières sont rencontrées. Chaque ligne contient tout d'abord une expression régulière suivie d'une action sous forme de code C sur une ligne ou plusieurs lignes (avec des accolades).
Exemples :

```
{chiffre}  puts("Un chiffre a été lu");
"une chaine" puts("Une chaine lue");
```

N'importe quel code peut être associé à l'expression régulière, y compris un `return` qui aura pour effet de renvoyer le contrôle au contexte appelant `yylex()`. C'est le comportement utilisé quand on le couple avec `bison`. L'appel suivant à `yylex()` renverra le jeton suivant.

▶ Jetons

Lorsque l'action associée est un `return`, on dit que l'analyseur renvoie un jeton. Un jeton (*token* en anglais), c'est :
- Un identifiant unique (valeur entière) : la valeur du `return`
- Une valeur associée

Voici quelques exemples de jeton
- Un mot clé d'un langage (pas de valeur associée)
- Un opérateur (pas de valeur associée)
- Une valeur entière (valeur associée = entier)
- Un identificateur (valeur associée = chaîne de caractères)

`flex` tout seul ne prévoit pas de mécanisme pour communiquer la valeur associée, il faut recourir à une variable globale (processus automatisé par `bison`). La variable globale `yytext` contient un pointeur vers le texte qui a été *matché* avec l'expression régulière.

▶ Code généré

Le code généré est mis dans un fichier `lex.yy.c` (nom par défaut) contenant l'analyseur lexical :
- `yylex` fonction qui exécute le scanner sur l'entrée `yyin` (par défaut l'entrée standard `stdin`). Si aucune action n'a de clause `return`, la fonction `yylex` prend fin lorsque le caractère de fin de fichier est lu.
- Si on veut changer le prototype de la fonction scanner, on doit utiliser la macro `YY_DECL`, par exemple :

  ```
  #define YY_DECL int yylex_perso(int param)
  ```

- On peut aussi influer sur le nom de la fonction grâce à l'option en ligne de commande `-P` qui change le préfixe `yy` en un autre
- Le code généré est très instructif, vous pouvez aller y jeter un coup d'œil mais attention, c'est un concentré de tout ce que l'on vous interdit de faire : variables globales, `goto`, code illisible,... Vous pouvez retenir que l'automate est géré par des tables.

2.6 Exercices

▶ Dérivées

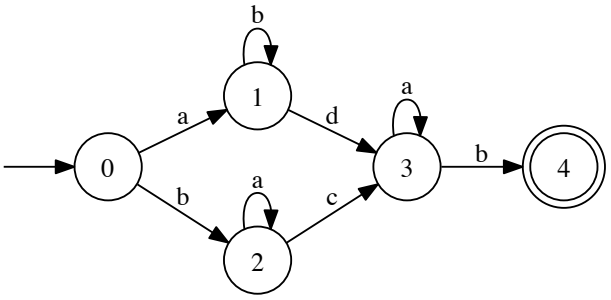
Dans tous les exercices ci-dessous, on utilisera la notation | pour indiquer un *ou* dans une expression régulière.
Exercice 1 Soit l'expression régulière $E = abc(a*|b)a$. Dériver successivement l'expression.
Exercice 2 En déduire l'automate déterministe de manière directe.
Exercice 3 Soit l'expression régulière suivante : $E = a(b|(cd)*)e*$. Dériver successivement l'expression.
Exercice 4 En déduire l'automate déterministe de manière directe.

▶ Construction d'automate

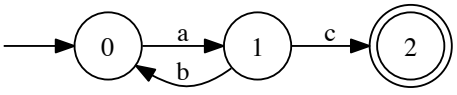
Exercice 5 Donner l'automate fini non-déterministe associé à $E = abc(a*|b)a$ (construction de Thompson).
Exercice 6 Donner la version déterministe de cet automate (chaque état de cet automate donnera l'ensemble des états de l'automate précédent)
Exercice 7 Donner l'automate fini non-déterministe associé à $E = a(b|(cd)*)e*$ (construction de Thompson).
Exercice 8 Donner la version déterministe de cet automate (chaque état de cet automate donnera l'ensemble des états de l'automate précédent)
Exercice 9 Donner l'automate fini non-déterministe associé à $E = (ab|c*|b+)ac$ (construction de Thompson).
Exercice 10 Donner la version déterministe de cet automate (chaque état de cet automate donnera l'ensemble des états de l'automate précédent)

▶ Expressions régulières

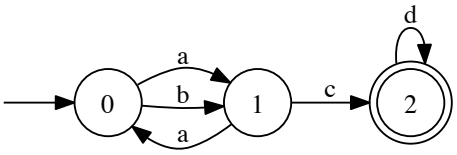
Donner les expressions régulières associées aux automates suivants :
Exercice 11



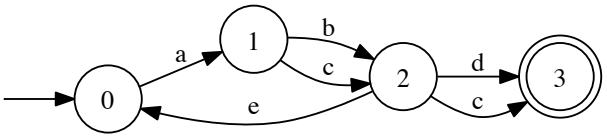
Exercice 12



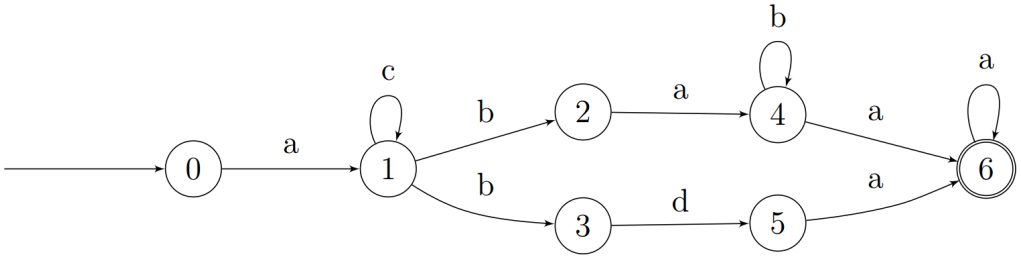
Exercice 13



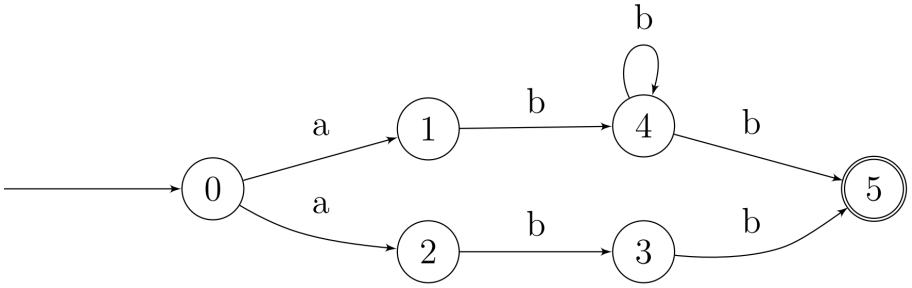
Exercice 14



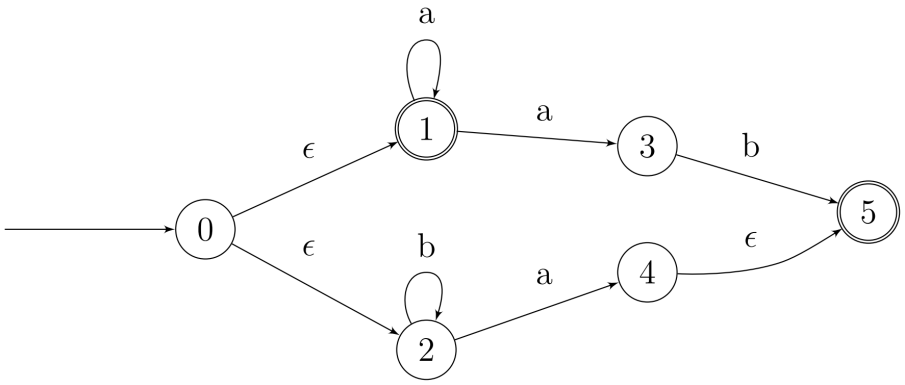
Exercice 15



Exercice 16



Exercice 17



CHAPITRE 3
ANALYSE SYNTAXIQUE

3.1 Introduction

Comme nous l'avons vu dans l'introduction, une grammaire hors-contexte est une grammaire dont les règles ont cette forme :

$$A \rightarrow \gamma$$

avec $A \in N$ et $\gamma \in V^*$.

La règle a donc uniquement un non-terminal à gauche et à droite un proto-mot qui peut combiner terminaux et non-terminaux. Les grammaires hors-contexte sont moins générales que les grammaires contextuelles, mais elles ont suffisamment de pouvoir d'expression pour être utilisées dans une large panoplie d'applications.
Il existe fondamentalement deux manières d'envisager l'analyse de grammaires hors-contexte :

- Descendante (top-down) : c'est la manière intuitive, on part de l'axiome, puis on essaye les dérivations successives de chaque non-terminal jusqu'à arriver aux symboles terminaux de la phrase à analyser
- Ascendante (bottom-up) : c'est l'inverse ! On prend la phrase à analyser (constituée au départ uniquement de symboles terminaux) et on identifie des parties droites de règles pour transformer la phrase en proto-mots de plus en plus constitués de non-terminaux jusqu'à arriver à l'axiome

3.2 Analyse descendante

Une analyse descendante simplifiée consisterait en une fonction qui prend deux arguments :

- Le proto-mot courant α : constitué de terminaux et de non-terminaux
- La phrase à reconnaître u : constituée de terminaux uniquement

La fonction va chercher toutes les règles associées au premier non-terminal de α et procéder récursivement au remplacement du non-terminal par la partie droite de la règle. La fonction prend fin lorsque le proto-mot est la phrase à reconnaître.

▶ Algorithme naïf

descendant1(α,u)
if $\alpha = u$ **then**
  return true
end if

$\alpha = wA\gamma$ {A est le premier non-terminal de α}
while $\exists A \to \delta$ **do**
  if descendant1($w\delta\gamma, u$) **then**
    return true
  end if
end while
return false

Que dire de cet algorithme ?

- ✘ Si une des règles est récursive à gauche, l'algorithme va partir en boucle infinie
- — Des cas de récursivité plus complexes qui mettent en jeu plusieurs règles peuvent arriver aussi
- ✘ L'algorithme fait une recherche exhaustive de tous les proto-mots qu'il est possible de construire sans même tenir compte de la phrase à reconnaître
- ✘ Conclusion : cet algorithme trop simpliste ne marche pas ou marche uniquement dans des cas dégénérés qui ne sont pas des grammaires hors-contextes (grammaires à choix finis)

▶ Algorithme moins naïf

descendant2(α,u)
if $\alpha = u$ **then**
  return true
end if
$\alpha = u_1 \dots u_k A\gamma$
while $\exists A \to \beta$ **do**
  $\beta = u_{k+1} \dots u_{k+l}\delta$
  if descendant2($u_1 \dots u_k u_{k+1} \dots u_{k+l}\delta\gamma, u$) **then**
    return true
  end if
end while
return false

Que dire de cette nouvelle version ?

- ✘ L'algorithme ne gère que des règles dont la partie droite commence par un non-terminal
- ✘ On a restreint l'expressivité de la grammaire aux grammaires linéaires (catégorie intermédiaire entre régulière et hors-contexte)

▶ Exemple

Exemple d'exécution avec la grammaire suivante :

$$S \to aSb$$
$$S \to ab$$

$descendant(S, aabb)$
$descendant(aSb, aabb)$
$descendant(aaSbb, aabb) \to$ false
$descendant(aabb, aabb) \to$ true

3.3 Analyse ascendante

L'analyse ascendante procède inversement par rapport à l'analyse descendante. Elle essaye d'inverser le processus de production de règles.

▶ Algorithme ascendant

L'analyse ascendante ne prend qu'un seul argument en paramètre :
— Le proto-mot courant α : constitué de terminaux et de non-terminaux
Contrairement à l'analyse descendante, le premier appel se fait avec la phrase à reconnaître. L'algorithme cherche dans le proto-mot un pattern de la partie droite d'une règle.

```
ascendant(α)
if α = S then
  return true
end if
for i = 1 to |α| do
  for j = i to |α| do
    if ∃A → α_i ... α_j then
      if ascendant(α_1 ... α_{i-1} A α_{j+1} ... α_n) then
        return true
      end if
    end if
  end for
end for
return false
```

Que dire de cet algorithme ?
✔ Il marche sans restriction sur la nature des règles de la grammaire !
✘ Il est encore plus combinatoire que son homologue descendant (chaque appel à la fonction est en $O(|\alpha|^2)$)
✘ Il nécessite une connaissance complète de la phrase à analyser à tout moment

▶ Exemple

Exemple d'exécution avec la grammaire suivante :

$$S \to aSb$$
$$S \to ab$$

$$ascendant(aabb)$$
$$ascendant(aSb)$$
$$ascendant(S) \to \text{true}$$

3.4 Conclusion

Il y a deux philosophies pour analyser des phrases d'une grammaire hors-contexte, mais les algorithmes naïfs présentés précédemment souffrent tous d'au moins un des défauts suivants :
— Piètres performances
— Restrictions d'utilisation sur la grammaire
— Restrictions d'utilisation sur la lecture du flux d'entrée
En revanche, les algorithmes présentés précédemment ont un gros avantage : ils sont très faciles à implémenter et ne nécessitent pas de transformer la grammaire.
Au prix de pré-traitements sur la grammaire, il est possible de construire des analyseurs performants. Voici les propriétés que l'on cherche à obtenir :
— Lecture dans le flux en avant limitée : dans le meilleur des cas, il suffira de lire de symbole courant et le suivant pour déterminer l'action à effectuer
— Performance : dans le meilleur des cas on construira un analyseur dit linéaire c'est-à-dire dont la complexité est linéaire en fonction de la taille de phrase à analyser (automate à pile)

CHAPITRE 4
ANALYSEURS DESCENDANTS

4.1 Introduction

La deuxième version de l'analyseur descendant est performante car elle est capable de déterminer quel sous-ensemble de règles on va pouvoir appliquer en comparant le début de la règle aux symboles non lus dans la phrase à analyser. Cela se fait au prix d'une restriction sur les règles qui doivent commencer par un symbole terminal.

Comment peut-on essayer de construire un analyseur sur le même principe mais qui lèverait cette restriction ?

L'idée est assez simple : si la règle commence par un symbole non terminal, il suffit d'aller chercher toutes les règles associées à ce non-terminal et cela récursivement jusqu'à l'obtention d'un terminal. C'est l'un des principes des analyseurs dits LL(1) et LL(k).

4.2 Analyse prédictive

Prenons l'exemple de la grammaire suivante :

$$
\begin{aligned}
E &\rightarrow E + F \mid F \\
F &\rightarrow F * T \mid T \\
T &\rightarrow (E) \mid N \\
N &\rightarrow a \mid b \mid c
\end{aligned}
$$

E ne produit que des proto-mots commençant par E ou F. Le seul moyen pour ces proto-mots de commencer par un terminal est de dériver avec la deuxième règle qui produit un F. On peut donc maintenant s'intéresser aux proto-mots commençants par F. Le mécanisme est le même que précédemment, ils ne commenceront que par des F ou des T, et au final par des T. Pour T cela devient différent, car une règle a un terminal en première position dans sa partie droite. T peut donc produire une phrase commençant par (ou par un N qui lui-même produit les terminaux a, b ou c. Tous les non-terminaux E, F et T peuvent produire finalement des phrases qui commencent par (, a, b, ou c. La lecture du premier caractère nous permettra donc de savoir quelle règle a été utilisée pour la production T. Mais comment savoir quelle règle a été utilisée pour E et pour F ? Il suffit de regarder ce qui se trouve derrière le symbole courant, cela peut-être un $+$ ou un $*$.

Le procédé d'analyse que nous venons d'évoquer peut être généralisé et automatisé.

▶ Premier $\mathcal{P}(A)$

L'ensemble des premiers d'un non-terminal A correspond à tous les symboles terminaux qui peuvent être en tête d'une dérivation gauche[1] de A. On le note $\mathcal{P}(A)$.
D'une manière formelle, soit $G = (N, \Sigma, S, R)$ une grammaire hors-contexte et $A \in N$, l'ensemble $\mathcal{P}(A)$ est un sous-ensemble de Σ défini par :

$$\mathcal{P}(A) = \{a \in \Sigma, \exists \alpha \in V^*, A \overset{*}{\Rightarrow} a\alpha\}$$

Le calcul automatique de $\mathcal{P}(A)$ se heurte à deux problèmes :
- Comment traiter les règles récursives à gauche ? Ce problème n'a pas de solution, il faudra écarter les grammaires qui contiennent ces règles ou les transformer.
- Comment traiter les règles qui peuvent produire ϵ ? Il faut identifier tous les non-terminaux susceptibles de réduire ϵ, l'ensemble $\mathcal{N}$.

▶ Nul $\mathcal{N}$

Soit $G = (N, \Sigma, S, R)$ une grammaire hors-contexte, l'ensemble $\mathcal{N}$ est un sous-ensemble de N défini par :

$$\mathcal{N} = \{A \in N, A \overset{*}{\Rightarrow} \epsilon\}$$

L'algorithme qui permet le calcul de $\mathcal{N}$ est simple, il consiste à inclure initialement tous les non-terminaux qui ont des ϵ-productions. Ensuite, la propagation se fait de manière itérative, en ajoutant tous les non-terminaux qui peuvent produire ceux déjà présents dans $\mathcal{N}$.

▶ Calcul de $\mathcal{P}(A)$

L'algorithme (simplifié) de calcul de $\mathcal{P}$ procède ainsi :

$\forall A \in N, \mathcal{P}^0(A) = \varnothing$
$k = 0$
Répéter
  $k = k + 1$
  $\forall A \in N, \mathcal{P}^k(A) = \mathcal{P}^{k-1}(A)$
  Pour toutes les règles $A \to X_1 \dots X_n$
    $i = 0$
    Répéter
      $i = i + 1$
      Ajouter $\mathcal{P}^k(X_i)$ à $\mathcal{P}^k(A)$
    Tant que $X_i \in \mathcal{N}$
  FinPour
Tant que $\{\mathcal{P}^k(A)\}$ différent de $\{\mathcal{P}^{k-1}(A)\}$

L'explication de l'algorithme est simple, sur une règle de type $A \to X_1 \dots X_n$, on sait que les premiers de X_1 seront forcément premiers de A. Mais si ϵ peut dériver de X_1, autrement dit si $X_1 \in \mathcal{N}$, alors il faut aussi considérer X_2, et ainsi de suite.

▶ Suivant $\mathcal{S}(A)$

Intéressons-nous maintenant aux derniers ensembles, ceux qui répertorient les symboles terminaux qui peuvent suivre un symbole non-terminal, appelés suivant et notés $\mathcal{S}(A)$.
Soit $G = (N, \Sigma, S, R)$ une grammaire hors-contexte et $A \in N$, l'ensemble $\mathcal{S}(A)$ est un sous-ensemble de Σ défini par :

$$\mathcal{S}(A) = \{a \in \Sigma, \exists u \in \Sigma^*, \alpha, \beta \in V^*, \\ S \overset{*}{\Rightarrow} uA\alpha \Rightarrow uAa\beta\}$$

1. réécriture du non-terminal le plus à gauche

Pour indiquer que le symbole suivant peut être la fin du mot, on utilise le symbole spécial \$. De manière évidente, ce symbole fait partie d'emblée des suivants de l'axiome de la grammaire. La propagation de ce symbole spécial va permettre de connaître la liste des non-terminaux qui peuvent être en fin de mot.
Pour calculer les $\mathcal{S}(A)$ on procède en deux passes :

— Initialisation basée sur les premiers : $\mathcal{S}(X) = \mathcal{S}(X) \cup \mathcal{P}(Y)$ s'il existe une règle dont la partie droite contient XY ou $X\alpha Y$ avec $\alpha \in \mathcal{N}^*$.
— Propagation : on ajoute $\mathcal{S}(A)$ à $\mathcal{S}(X)$ pour toutes les règles du type $A \rightarrow \alpha X \beta$ où $\beta \in \mathcal{N}^*$. La modification de $\mathcal{S}(X)$ peut alors engendrer la propagation pour un autre non-terminal, il faut donc appliquer un algorithme de point fixe (itération jusqu'à stabilité).

▶ Exemple complet

La grammaire :

$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid -\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid /\ F\ T' \mid \epsilon \\ F & \rightarrow & (\ E\ ) \mid nb \mid id \end{array}$$

On commence par calculer $\mathcal{N}$, ici c'est simple :

$$\mathcal{N} = \{E', T'\}$$

On calcule ensuite les premiers :

$$\begin{array}{l} \mathcal{P}(E) = \mathcal{P}(T) = \mathcal{P}(F) = \{(, nb, id\} \\ \mathcal{P}(E') = \{+, -, \epsilon\} \\ \mathcal{P}(T') = \{*, /, \epsilon\} \end{array}$$

Pas besoin d'itérations pour le calcul des premiers car aucun non-terminal de début de règle n'appartient à $\mathcal{N}$. Voici maintenant l'initialisation des suivants :

$$\begin{array}{l} \mathcal{S}(E) = \{\$,)\} \\ \mathcal{S}(E') = \{\} \\ \mathcal{S}(T) = \mathcal{P}(E') = \{+, -\} \\ \mathcal{S}(T') = \{\} \\ \mathcal{S}(F) = \mathcal{P}(T') = \{*, /\} \end{array}$$

La règle $E \rightarrow T\ E'$ permet de conclure que les suivants de E sont aussi suivants de E', et comme $E' \in \mathcal{N}$ de T aussi :

$$\begin{array}{l} \mathcal{S}(E') = \{\$,)\} \\ \mathcal{S}(T) = \{+, -, \$,)\} \end{array}$$

La règle $T \rightarrow F\ T'$ permet de propager les suivants de T vers ceux de T' et F :

$$\begin{array}{l} \mathcal{S}(T') = \{+, -, \$,)\} \\ \mathcal{S}(F) = \{*, /, +, -, \$,)\} \end{array}$$

La quatrième ligne de règles permet de propager les suivants de T' vers ceux de F, mais ceux-ci sont déjà présents. L'application d'une deuxième itération de propagation ne change rien au résultat, donc on s'arrête. Voici le résultat final :

$$\begin{array}{l} \mathcal{S}(E) = \{\$,)\} \\ \mathcal{S}(E') = \{\$,)\} \\ \mathcal{S}(T) = \{+, -, \$,)\} \\ \mathcal{S}(T') = \{+, -, \$,)\} \\ \mathcal{S}(F) = \{*, /, +, -, \$,)\} \end{array}$$

▶ Récapitulatif

Afin de ne rien oublier (c'est vite fait), il faut procéder dans cet ordre :

1. Commencer par calculer les symboles qui font partie de $\mathcal{N}$ et les marquer au surligneur par exemple. De cette manière, on voit tout de suite qu'un symbole peut être nul par la suite.
2. Calculer les ensembles des premiers en deux temps :
   — Les premiers qui sont directs (symboles terminaux en début de règle)
   — Propagation en regardant la partie de droite de chaque règle, en commençant par la gauche et tant que les symboles non-terminaux peuvent être nuls
3. Calculer les ensembles des suivants en deux temps :
   — Diffusion des premiers vers les suivants lorsque dans une règle deux symboles sont successifs (ou alors séparés par un ou plusieurs symboles potentiellement nuls)
   — Propagation des suivants de la partie gauche de chaque règle vers la partie droite en commençant par le symbole le plus à droite puis en décalant à gauche tant que le non-terminal peut être nul

4.3 Analyseurs LL

Maintenant que nous avons à disposition les informations de prédiction relatives à une grammaire, nous allons pouvoir construire un analyseur LL qui signifie *Left to right Leftmost derivation*.

▶ À la main

La première solution consiste à écrire à la main une fonction par symbole non-terminal. Cette fonction renvoie true en cas de succès et false en cas d'échec. Elle a le droit d'appeler deux fonctions :

— `next()` qui indique le prochain symbole terminal à lire sans passer au suivant
— `shift()` qui passe au symbole terminal suivant dans le flux

L'écriture de la fonction est faite grâce aux règles ainsi qu'aux éléments de prédiction. Ces éléments devront permettre de sélectionner la bonne règle. Voici le fonctionnement général (supposons que l'on veut analyser A) :

— Si le prochain symbole n'appartient pas à $\mathcal{P}(A)$ on retourne false (sauf si $A \in \mathcal{N}$, voir dernier point)
— Si le prochain symbole est en tête d'une règle, on le lit et on sélectionne cette règle (avec un appel aux fonctions associées aux non-terminaux)
— Si le prochain symbole appartient à $\mathcal{P}(B)$ où B est un non-terminal d'une règle, alors on ne lit pas et on sélectionne cette règle
— Si $A \in \mathcal{N}$ et que le prochain symbole appartient à $\mathcal{S}(A)$ on renvoie true (on a validé la règle vide)

Prenons l'exemple du symbole E dans la grammaire dont nous avons étudié les informations de prédiction. E n'a qu'une seule règle associée, $E \rightarrow T\ E'$. L'implémentation de la fonction associée sera simple, elle consistera uniquement à regarder si le symbole à lire fait bien partie de $\mathcal{P}(E)$ et appeler les fonctions associées à T et E'.

```
analyseE()
if next() ∉ {(, nb, id} then
    return false
end if
if not analyseT() then
    return false
end if
if not analyseE'() then
    return false
end if
```

return true

Passons maintenant à un cas plus difficile, celui du symbole E' :

$$E' \rightarrow + \ T \ E' \mid - \ T \ E' \mid \epsilon$$

Si le prochain symbole à lire est $+$ ou $-$ alors on sait qu'il faut choisir les règles correspondantes, sinon il faut recourir à $\mathcal{S}(E')$ afin de savoir si l'ϵ-production doit être sélectionnée.

```
analyseE'()
if next() ∈ {+, −} then
  shift()
  return analyseT() and analyseE'()
end if
return next() ∈ {$, )}
```

Remarque : le code a été volontairement simplifié.

▶ Avec une table

L'écriture de toutes les fonctions associées à chaque non-terminal peut s'avérer fastidieuse. Comme pour les automates finis déterministes, on peut calculer une table des transitions qui en fonction d'un symbole non-terminal et d'un symbole terminal va associer la réécriture du non-terminal. Cette table a donc deux entrées et les cases vides indiqueront une erreur.
Reprenons notre grammaire exemple :

$$\begin{array}{lcl} E & \rightarrow & T \ E' \\ E' & \rightarrow & + \ T \ E' \mid - \ T \ E' \mid \epsilon \\ T & \rightarrow & F \ T' \\ T' & \rightarrow & * \ F \ T' \mid / \ F \ T' \mid \epsilon \\ F & \rightarrow & (\ E \ ) \mid nb \mid id \end{array}$$

Et commençons par remplir le tableau avec toutes les réécritures triviales :

	$+$	$-$	$/$	$*$	$($	$)$	nb	id	$\$$
E									
E'	$+TE'$	$-TE'$							
T									
T'			$/FT'$	$*FT'$					
F					(E)		nb	id	

Viennent ensuite les transitions qui sont dues aux premiers :

	$+$	$-$	$/$	$*$	$($	$)$	nb	id	$\$$
E					TE'		TE'	TE'	
E'	$+TE'$	$-TE'$							
T					FT'		FT'	FT'	
T'			$/FT'$	$*FT'$					
F					(E)		nb	id	

Il ne manque plus que les ϵ-transitions :

	$+$	$-$	$/$	$*$	$($	$)$	nb	id	$\$$
E					TE'		TE'	TE'	
E'	$+TE'$	$-TE'$				ϵ			ϵ
T					FT'		FT'	FT'	
T'	ϵ	ϵ	$/FT'$	$*FT'$		ϵ			ϵ
F					(E)		nb	id	

L'algorithme associé à l'utilisation de cette table s'apparente à un automate, mais pour lequel il faut ajouter le stockage d'une information : une pile. Dans cette pile, on va stocker des proto-mots (mélange de terminaux et non-terminaux).

- L'automate est initialisé avec l'axiome de la grammaire.
- Lorsque l'entrée dans la table commence par un non-terminal, il n'y aura pas de lecture d'un symbole mais juste la transformation dans la pile de la partie gauche en partie droite.
- Lorsque l'entrée dans la table commence par un terminal, ce terminal sera lu et le reste de la partie droite remplace la partie gauche.
- Lorsque l'entrée dans la table est ϵ, on se contente de lire le symbole et dépiler le non-terminal.
- L'automate se termine quand la pile est vide.

Exemple : la lecture de $3 * x + 5$.

E	$nb(3)\ *\ id(x)\ +\ nb(5)\ \$$	Initialisation
$T\ E'$	$nb(3)\ *\ id(x)\ +\ nb(5)\ \$$	$E \to T\ E'$
$F\ T'\ E'$	$nb(3)\ *\ id(x)\ +\ nb(5)\ \$$	$T \to F\ T'$
$T'\ E'$	$*\ id(x)\ +\ nb(5)\ \$$	$F \to nb$
$F\ T'\ E'$	$id(x)\ +\ nb(5)\ \$$	$T' \to *\ F\ T'$
$T'\ E'$	$+\ nb(5)\ \$$	$F \to id$
E'	$+\ nb(5)\ \$$	$T' \to \epsilon$
$T\ E'$	$nb(5)\ \$$	$E' \to +\ T\ E'$
$F\ T'\ E'$	$nb(5)\ \$$	$T \to F\ T'$
$T'\ E'$	$\$$	$F \to nb$
E'	$\$$	$T' \to \epsilon$
	$\$$	$E' \to \epsilon$
	$\$$	Fin

Remarque : la pile que l'on manipule ici est le reflet exact des appels des fonctions d'analyse créées dans la section précédente.

▶ En C++

Les symboles sont simplement représentés par des entiers :

```
typedef enum {E_, Ep_, T_, Tp_,
              ..., close_, end_} Token;
```

Le template `deque` permet de stocker efficacement une pile :

```
typedef deque<Token> Pile;
```

Un état est décrit par sa pile ainsi que les symboles restant à lire :

```
struct Etat {
   Pile pile;
   Pile alire;
};
```

Enfin, les transitions sont gérées par une double `map` :

```
typedef map<Token,map<Token,Pile> > Transitions;
```

Afin de savoir si une transition existe ou non, on fait une petite fonction qui cherche dans les deux `map` :

```
bool Existe(Transitions & t, Token i, Token f) {
   return (t.find(i)!=t.end()) and
          (t[i].find(j)!=t[i].end());
}
```

On peut maintenant écrire notre automate :

```
bool LL1AAP(const Pile & mot,
            Transitions & transitions,
            Token axiome)
{
   Etat etat;
   etat.pile.push_front(axiome);
   etat.alire = mot;
   Token a,b;
   if (etat.alire.back() != end_) {
      etat.alire.push_back(end_);
   }

   while (!etat.pile.empty()) {
      a = etat.pile.front();
      b = etat.alire.front();
      if (a==b) {
         etat.pile.pop_front();
         etat.alire.pop_front();
      } else if (!Existe(transitions,a,b)) {
         return false;
      } else {
         etat.pile.pop_front();
         Pile::reverse_iterator i;
         for (i=transitions[a][b].rbegin();
              i!=transitions[a][b].rend();i++) {
            etat.pile.push_front(*i);
         }
      }
   }
   return true;
}
```

Vous trouverez en annexe un exemple complet de ce code.

▶ Grammaires et langages LL(1)

Lors de la construction de la table précédente, nous avons rempli au plus une valeur par case. Cela signifie que la lecture d'un seul symbole permet de décider de la règle à appliquer (d'où le (1)).

Grammaire LL(1) Une grammaire hors-contexte est dite LL(1) si sa table d'analyse prédictive ne comporte qu'une seule séquence associée à chaque couple $(A, a) \in (V, \Sigma)$.

Dans ce cas-là une analyse dite LL(1) est possible par l'exploitation de la table construite.
Parfois, une grammaire n'est pas LL(1) alors qu'on pourrait en construire une.

Langage LL(1) Un langage est dit LL(1) s'il existe une grammaire LL(1) qui le génère.

La plupart des langages de programmation (tout du moins récents) sont LL(1). Ils peuvent donc être analysés par un analyseur prédictif de type LL(1).

▶ Transformations

De simples transformations permettent de modifier une grammaire pour qu'elle se rapproche d'une grammaire LL(1).
Une des premières limitations qui empêchent une grammaire d'être LL(1) est qu'elle comporte des récursions gauches. Lorsque cela est possible, on pourra les transformer en récursions droites moyennant l'ajout de nouveaux non-terminaux et de productions vides. Exemple :

$$E \rightarrow E \ + \ T \mid T$$

peut être transformé en :

$$E \rightarrow T \ E'$$
$$E' \rightarrow + \ T \ E' \mid \epsilon$$

Deuxième souci possible, la table des prédictions comporte plus d'un élément par case. Cela arrive lorsque des règles commencent de la même manière notamment. Il suffit alors de factoriser à gauche la partie qui pose problème. Exemple classique du $if/else$ en compilation :

$$S \rightarrow if\ T\ then\ S$$
$$S \rightarrow if\ T\ then\ S\ else\ S$$

on peut factoriser la partie commune et faire du $else$ une partie optionnelle :

$$S \rightarrow if\ T\ then\ S\ S'$$
$$S' \rightarrow else\ S \mid \epsilon$$

▶ LL(k)

Lorsqu'il est impossible de transformer la grammaire en grammaire LL(1), il se peut que la grammaire soit LL(k). Dans ce cas-là, l'analyse descendante nécessitera la lecture de k symboles dans le flux d'entrée au maximum pour décider de l'action à effectuer.
Par exemple, la grammaire suivante n'est pas LL(1) :

$$A \rightarrow a\ b\ B \mid \epsilon$$
$$B \rightarrow A\ a\ a \mid b$$

En effet, le symbole a ne permettra pas à lui tout seul de savoir si l'on doit utiliser la première ou la deuxième règle de A.
Exercice : quel est le langage généré par cette grammaire ?
On peut généraliser la notion de prédiction à plusieurs symboles (y compris les ensembles premiers et suivants) et construire une table d'analyse LL(2) :

	aa	ab	b	\$
A	ϵ	abB		ϵ
B	Aaa	Aaa	b	

Parfois il est même impossible de construire une grammaire LL(k). Exemple de cas pathologique :

$$S \rightarrow A \mid B$$
$$A \rightarrow a\ A\ b \mid \epsilon$$
$$B \rightarrow a\ B\ b\ b \mid \epsilon$$

Le langage généré est $\{a^i b^i, i \geq 0\} \cup \{a^i b^{2i}, i \geq 0\}$. Avant de savoir s'il s'agissait de A ou B il faut attendre la lecture de tous les a puis de la même quantité de b, soit potentiellement une infinité de symboles !
Autre exemple de cas dégénéré, les palindromes pairs, qui peuvent être construits avec une grammaire hors-contexte ultra-simple, non ambigüe, non récursive à gauche, et factorisée à gauche :

$$A \rightarrow a\ A\ a \mid b\ A\ b \mid \epsilon$$

Cette grammaire n'est ni LL(1) ni LL(k).

4.4 prolog

▶ Concaténation

L'implémentation d'analyseurs en prolog est assez aisée. Le flux à lire est représenté par une liste. Cette liste est découpée grâce au prédicat de concaténation `append/3`. Chaque règle donne lieu

à l'écriture d'un prédicat où l'on commence par appeler la concaténation, puis on appelle les prédicats de la suite de la règle. Exemple avec la grammaire suivante :

$$
\begin{aligned}
&E \rightarrow T\ E' \\
&E' \rightarrow +\ T\ E' \mid \epsilon \\
&T \rightarrow F\ T' \\
&T' \rightarrow *\ F\ T' \mid \epsilon \\
&F \rightarrow nb
\end{aligned}
$$

```
expression(X) :- % E -> T E'
   append(A,B,X),
   terme(A),
   expression_suite(B).
expression_suite(['+'|X]) :- % E' -> + T E'
   append(A,B,X),
   terme(A),
   expression_suite(B).
expression_suite([]). % E' -> e
terme(X) :- % T -> F T'
   append(A,B,X),
   facteur(A),
   terme_suite(B).
terme_suite(['*'|X]) :- % T' -> * F T'
   append(A,B,X),
   facteur(A),
   terme_suite(B).
terme_suite([]). % T' -> e
facteur([X]) :- % F -> nb
   number(X).
```

Le gros inconvénient de cette méthode est que les possibilités énumérées par le concaténateur sont combinatoires et la plupart d'entre elles sont inexploitables. Cela nous amène à utiliser une autre représentation du flux d'entrée.

▶ Listes différentielles

L'idée des listes différentielles (DL) est de représenter les données avec deux listes, la deuxième étant le suffixe de la première indiquant quelle partie de la liste il ne faut pas considérer. On peut donc représenter facilement :

- Une liste vide avec `[X,X]`
- Une décomposition en sous-listes (en évitant la concaténation). Exemple : la DL `[A,B]` se décompose en `[A,X1]`, `[X1,X2]`, `[X2,X3]`, ..., `[Xn,B]`.
- La comparaison avec un symbole : la DL `[X,S]` représente le mot `A` si `mot(X,A,S)` avec `mot([Y|S],Y,S)`.

```
expression(X) :- % wrapper
   expressiondl([X,[]]).
expressiondl([X,S]) :-
   terme([X,S1]),
   expression_suite([S1,S]).
expression_suite([X,S]) :-
   mot(X,'+',S1),
   terme([S1,S2]),
   expression_suite([S2,S]).
expression_suite([X,X]).
terme([X,S]) :-
   facteur([X,S1]),
   terme_suite([S1,S]).
```

```
terme_suite([X,S]) :-
   mot(X,'*',S1),
   facteur([S1,S2]),
   terme_suite([S2,S]).
terme_suite([X,X]).
mot([X|S],X,S).
facteur([[X|Y],Y]) :-
   number(X).
facteur([X,_]) :-
   number(X).
```

4.5 Exercices

▶ Grammaire de grammaire

Voici la grammaire qui permet de décrire une grammaire formelle hors-contexte :

$$\begin{array}{lcl} G & \rightarrow & R\ p\ G \\ G & \rightarrow & \epsilon \\ R & \rightarrow & n\ f\ L \\ L & \rightarrow & n\ L \\ L & \rightarrow & t\ L \\ L & \rightarrow & \epsilon \end{array}$$

Les symboles terminaux sont les suivants : n, t, p et f. G symbolise la grammaire, R une règle, et L une liste de symboles terminaux (t) ou non-terminaux (n). f est une flèche et p un séparateur de type point-virgule. Le symbole ϵ indique le mot vide (epsilon production).
Exercice 18 Indiquer l'ensemble des nuls $\mathcal{N}$ de cette grammaire ainsi que pour chaque symbole non terminal l'ensemble des premiers $\mathcal{P}$ et des suivants $\mathcal{S}$ (on représentera la fin d'un mot par le symbole \$).
Exercice 19 Donner sa table d'analyse LL(1).

▶ Pseudo XML

Étant donné la grammaire :

$$\begin{array}{lcl} E & \rightarrow & \text{open}\ C\ \text{close} \\ C & \rightarrow & I\ C \\ C & \rightarrow & \epsilon \\ I & \rightarrow & E \\ I & \rightarrow & \text{data} \end{array}$$

Les symboles terminaux sont open, close et data.
Exercice 20 Indiquer l'ensemble des nuls $\mathcal{N}$ de cette grammaire ainsi que pour chaque symbole non terminal l'ensemble des premiers $\mathcal{P}$ et des suivants $\mathcal{S}$ (on représentera la fin d'un mot par le symbole \$).
Exercice 21 Donner sa table d'analyse LL(1).

CHAPITRE 5
ANALYSEURS ASCENDANTS

5.1 Automates à pile

▶ Introduction

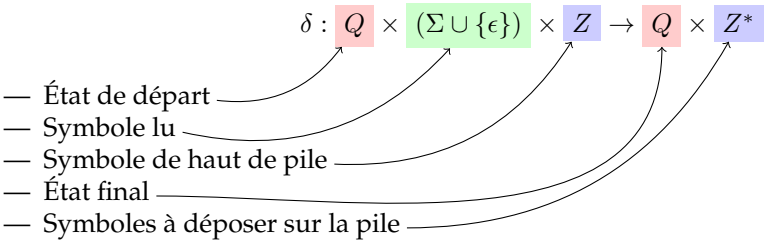
Nous avons vu dans le chapitre précédent que pour reconnaître un langage LL(1), on peut avoir recours à un analyseur sous la forme d'un automate à pile. Nous allons dans cette section définir de manière formelle les automates à pile ainsi que les automates d'items qui sont la base des analyseurs ascendants.

▶ Définition

D'une manière formelle, un automate à pile est un automate à états fini que l'on dote d'une pile (non bornée). Les éléments que l'on dépose dans la pile font partie d'un alphabet de pile Z.
Un automate à pile déterministe est défini par :

- Σ un ensemble fini de symboles (appelé alphabet)
- Q un ensemble fini d'états
- $q_0 \in Q$ l'état initial
- $F \subset Q$ l'ensemble des états finaux
- Z un alphabet de pile
- z_0 le symbole de fond de pile
- δ une fonction de $Q \times (\Sigma \cup \{\epsilon\}) \times Z$ dans $Q \times Z^*$, appelée fonction de transition.

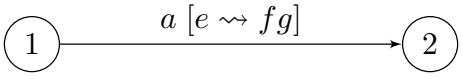
Détail de la fonction de transition



⚠ Dans la définition précédente, on a autorisé les ϵ-transitions, ce qui peut paraître contradictoire avec la notion de déterminisme. En effet, il faudra s'assurer que lorsqu'une ϵ-transition est définie pour un couple état/sommet de pile, alors aucune autre transition ne sera définie pour cette même configuration et un autre symbole. Une autre solution consiste à ne pas autoriser d'ϵ-transition.

▶ Représentation graphique

Comme pour les automates finis, on peut faire une représentation graphique d'un automate à pile. Pour cela on va ajouter l'information concernant la pile sur l'arc joignant deux états lors d'une transition.
Si on a $\delta(1, a, e) = (2, fg)$, alors on va relier les états 1 et 2 par un arc sur lequel on indiquera toutes les informations associées, par exemple de la forme $a\ [e \rightsquigarrow fg]$:



▶ Langage reconnu (état final)

Comme pour les automates finis, on dit qu'un mot est reconnu par l'automate à pile si les transitions successives de chacun de ses symboles mènent à un état final.
Le langage reconnu par un automate à pile est l'ensemble des mots que cet automate reconnait.

▶ Langage reconnu (pile vide)

Il existe une autre manière de définir un langage reconnu par un automate à pile qui consiste à dire qu'un mot est reconnu lorsque les transitions successives de chacun de ses symboles mènent à une pile vide. Cette définition est plus utilisée (et permet au passage de s'affranchir de la définition des états finaux).

▶ Automates à piles non-déterministes

Les définitions que nous avons données permettent d'exprimer des automates déterministes, c'est-à-dire dont chaque transition mène à un seul état. Afin d'obtenir une version non-déterministe, l'ensemble d'arrivée de la fonction δ doit être $2^{Q \times Z^*}$:

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times Z \to 2^{Q \times Z^*}$$

Un mot est reconnu s'il existe une série de transitions menant à l'état final ou à la pile vide (selon le type de reconnaissance que l'on souhaite utiliser).

▶ Pouvoir d'expression

Les automates à pile non-déterministes ont un pouvoir d'expression plus fort que leur variante déterministe. Ils permettent donc de décrire plus de langages. En conséquence, il n'y aura pas de mécanisme permettant de construire un automate à pile déterministe à partir d'un automate à pile non-déterministe. Le challenge va donc être de construire directement un automate à pile déterministe à partir d'une grammaire hors contexte.

▶ Simplification

Dans la plupart des utilisations que nous allons faire des automates à piles, l'ensemble des états Q et l'alphabet de la pile Z seront confondus. Cela signifie que nous allons empiler des états. C'est le cas des automates des items que nous allons aborder maintenant.

5.2 Automate des items

▶ Introduction

L'automate des items d'une grammaire hors-contexte permet de construire un automate à pile **non-déterministe** qui reconnaît le langage associé à cette grammaire. Il est non-déterministe donc

en pratique inutilisable. Il sert de cadre théorique pour la plupart des algorithmes d'analyse ascendante.

▶ Item

Un item d'une grammaire G a la forme suivante :

$$[X \to \alpha \bullet \beta]$$

où $X \to \alpha\ \beta$ est une production de la grammaire G et $\alpha, \beta \in V^*$.

Que l'on peut exprimer par : "En cherchant à dériver de X un mot $w = uv \in T^*$, on a déjà dérivé un mot u de α et il reste à dériver v de β."

Le point peut se trouver en première ou dernière position, indiquant ainsi des cas spéciaux :

— $[X \to \bullet\gamma]$ signifie que l'on cherche à reconnaître un mot de la forme γ pour le non-terminal X
— $[X \to \gamma\bullet]$ signifie que l'on a reconnu un mot pour X

Une production vide ne possède qu'un seul item $[X \to \bullet]$.

Il est courant et pratique d'étendre la grammaire en ajoutant un nouvel axiome qui n'interviendra que dans une seule règle en partie gauche et dans aucune en partie droite. Lorsque ce nouvel axiome est reconnu, on sait que l'on a terminé.

▶ Exemple

Voici une grammaire du langage $\{a^n b^n\}$:

$$A \to a\ A\ b \mid \epsilon$$

On commence par étendre cette grammaire :

$$\begin{array}{lcl} S & \to & A \\ A & \to & a\ A\ b \mid \epsilon \end{array}$$

La liste des items associés à cette grammaire est :
$[S \to \bullet A]$ $[S \to A\bullet]$
$[A \to \bullet\ a\ A\ b]$ $[A \to a \bullet A\ b]$ $[A \to a\ A \bullet b]$ $[A \to a\ A\ b\ \bullet]$
$[A \to \bullet]$

▶ Construction de l'automate

Les items que l'on vient de construire vont en même temps servir d'alphabet de pile et d'états. Soit Z_G l'ensemble des items de la grammaire G. Nous allons modifier la fonction de transition de l'automate à pile pour pouvoir regarder le sommet ainsi que le sous-sommet de la pile avant de faire une transition :

$$\delta : (Z_G \cup (Z_G \times Z_G)) \times (\Sigma \cup \{\epsilon\}) \rightarrow {Z_G}^*$$

— Item (un ou deux) présents sur le haut de la pile
— Symbole lu
— Items à déposer sur la pile

▶ Transition de lecture

Cette transition est très intuitive. Elle revient à exprimer que lorsqu'on a l'item $[X \rightarrow \alpha \bullet a\ \beta]$ sur le sommet de la pile et que le prochain symbole à lire est a, alors on lit le symbole et on remplace le sommet de la pile par $[X \rightarrow \alpha\ a \bullet \beta]$. Cette transition a un bilan nul au niveau de la taille de la pile.
C'est la seule transition de l'automate des items qui ne soit pas une ϵ-transition.

▶ Transition d'expansion

Sur le sommet de la pile, l'item dit qu'il y a un symbole non-terminal à reconnaître. D'un autre côté, une règle indique ce non-terminal en partie gauche. On va alors empiler l'item correspondant à cette règle. Pile de départ :

$$[X \rightarrow \alpha \bullet Y\ \beta]$$

La règle $Y \rightarrow \gamma$ permet d'empiler l'item $[Y \rightarrow \bullet\gamma]$ pour indiquer qu'une façon de reconnaître Y est de reconnaître γ :

$$[X \rightarrow \alpha \bullet Y\ \beta]\ [Y \rightarrow \bullet\gamma]$$

Cette transition a un bilan de $+1$ au niveau de la taille de la pile.

▶ Transition de réduction

Elle est le complémentaire de la précédente. On a réussi à reconnaître γ (par la lecture de symboles), cela nous permet de dépiler l'item associé et d'indiquer que l'on a reconnu Y :

$$[X \rightarrow \alpha \bullet Y\ \beta]\ [Y \rightarrow \gamma\bullet]$$

devient :

$$[X \rightarrow \alpha\ Y \bullet \beta]$$

Cette transition a un bilan -1 au niveau de la taille de la pile.

▶ Théorème

Le langage reconnu par l'automate (non-déterministe) des items d'une grammaire G est le langage associé à cette grammaire.
En pratique, cet automate est inutilisable car non-déterministe. Suivant la forme de la grammaire, son implémentation serait susceptible de tourner en boucle infinie.

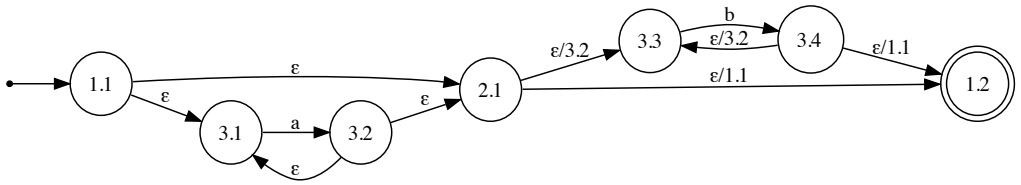


FIGURE 5.1 – Automate à items associé à la grammaire $S \rightarrow A, A \rightarrow aAb \mid \epsilon$

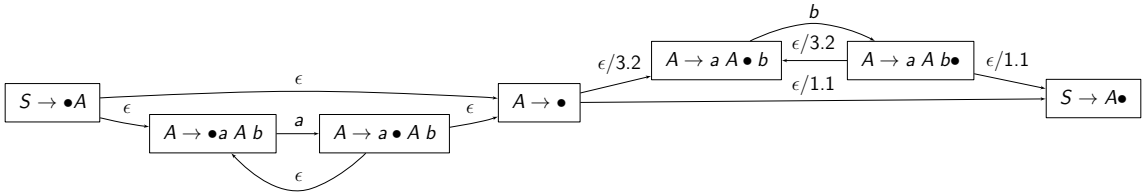


FIGURE 5.2 – Version développée de l'automate à items associé à la grammaire $S \rightarrow A, A \rightarrow aAb \mid \epsilon$

▶ Exemple

Reprenons la grammaire précédente :

$$\begin{array}{lcl} S & \rightarrow & A \\ A & \rightarrow & \epsilon \\ A & \rightarrow & a\ A\ b \end{array}$$

La liste (numérotée) des items associés :
$1.1[S \rightarrow \bullet A]$ $1.2[S \rightarrow A\bullet]$
$2.1[A \rightarrow \bullet]$
$3.1[A \rightarrow \bullet a\ A\ b]$ $3.2[A \rightarrow a \bullet A\ b]$
$3.3[A \rightarrow a\ A \bullet b]$ $3.4[A \rightarrow a\ A\ b\bullet]$
Les transitions de type lecture feront juste apparaître le symbole lu, on remplacera l'état de la pile par le nouvel état cible. Les transitions de type expansion feront juste apparaître le symbole ϵ et empileront l'état cible sans désempiler l'ancien. Les transitions de type réduction feront apparaître le symbole ϵ ainsi que l'état (item) qui doit être présent dans la sous-pile. Cette transition remplacera les deux anciens états par l'état cible.
L'automate à items associé se trouve en figure 5.1 page 49.
La version développée de l'automate à items associé se trouve en figure 5.2 page 49.

▶ Intuition

Dans l'exemple précédent, l'automate des items pourrait être programmé, il suffit de gérer un ensemble de configurations, chaque configuration étant une pile d'items. La lecture d'un symbole modifie cet ensemble et à chaque étape on calcule le point fixe des ϵ-transitions.
C'est ce dernier point qui est problématique.
Imaginons maintenant la grammaire suivante :

$$A \rightarrow A\ b \mid \epsilon$$

Il est assez aisé de voir que l'item $[A \rightarrow \bullet Ab]$ a une transition d'expansion sur lui-même. La phase de stabilisation des ϵ-transitions n'est donc pas possible. Il est à noter que d'autres problèmes que celui-là peuvent apparaître.
Deux solutions s'offrent à nous maintenant :

— Transformer la grammaire pour en supprimer les récursivités gauches.
— Essayer de construire un automate déterministe directement (quitte à restreindre la classe des grammaires utilisables).

La seconde solution est bien évidemment préférable. Remarque : les algorithmes que nous avons étudiés dans le chapitre précédent (analyse descendante) font partie de cette solution.

5.3 Analyseurs LR(0)

Nous avons vu précédemment que l'automate des items pose des problèmes à cause des ϵ-transitions. L'idée des analyseurs LR(0) est de regrouper les items en fonction de ces transitions. Chaque état de l'automate LR(0) est donc stable par les ϵ-transitions.
Certains états ne possèdent qu'un item puits (avec un point en dernière position) et indiquent des réductions (un symbole non-terminal a été reconnu).

▶ Exemple

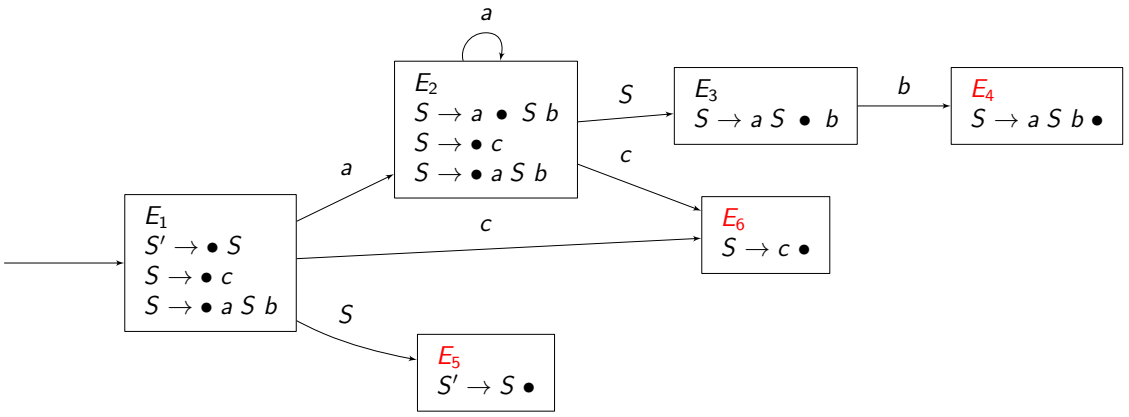
Avec la grammaire suivante :

$$S \rightarrow a\ S\ b \mid c$$

Une fois augmentée, on obtient :

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & c \\ S & \rightarrow & a\ S\ b \end{array}$$

Sans plus attendre voici l'automate LR(0) :



Et son exécution avec le mot `acb` en entrée :

— Initialisation : E_1
— Lecture de `a` : $E_1\ E_2$
— Lecture de `c` : $E_1\ E_2\ E_6$
— Réduction de $S \rightarrow c$, on dépile E_6, et on applique la transition S : $E_1\ E_2\ E_3$
— Lecture de `b` : $E_1\ E_2\ E_3\ E_4$
— Réduction de $S \rightarrow a\ S\ b$, on dépile $|aSb| = 3$ états, et on applique la transition S : $E_1\ E_5$
— L'état de sommet de pile est l'état final, le mot est reconnu.

▶ Méthode de construction

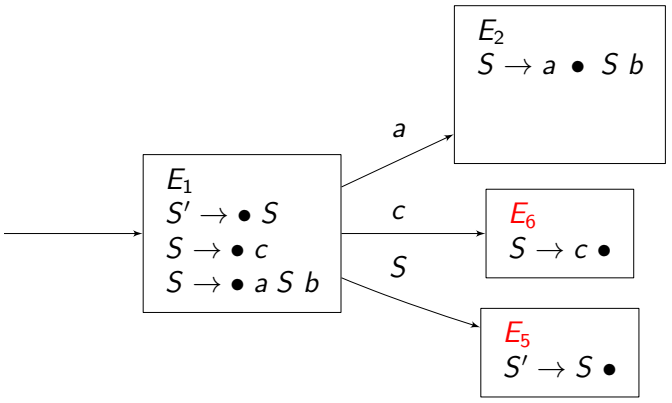
L'algorithme commence avec l'item de départ, dans l'exemple $[S' \rightarrow \bullet S]$. Afin de compléter cet item pour en faire l'état de départ on va chercher de manière récursive toutes les règles dont le symbole non-terminal est celui suivant le point dans la partie droite. Dans l'exemple, on commence donc avec les règles impliquant S. Cela revient à dire que le prochain symbole que l'on

cherche à reconnaître est S donc on va chercher de quels symboles il peut être composé. Dans notre exemple, deux règles mettent en jeu S, on ajoute donc les items suivants :

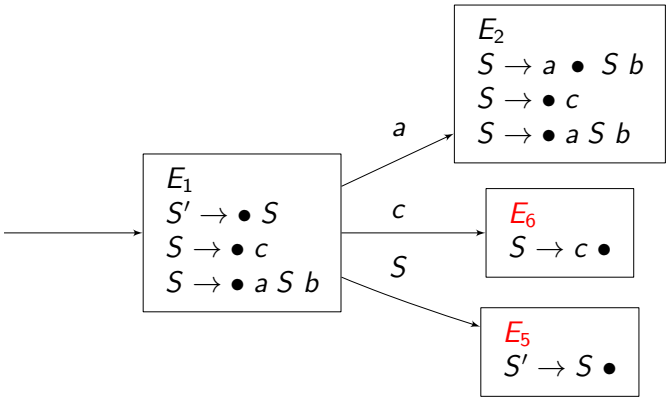
— $[S \rightarrow \bullet c]$
— $[S \rightarrow \bullet a\ S\ b]$

Il faut normalement réitérer le processus avec les nouveaux items, mais dans notre exemple, aucun ne possède de non terminal après le point.

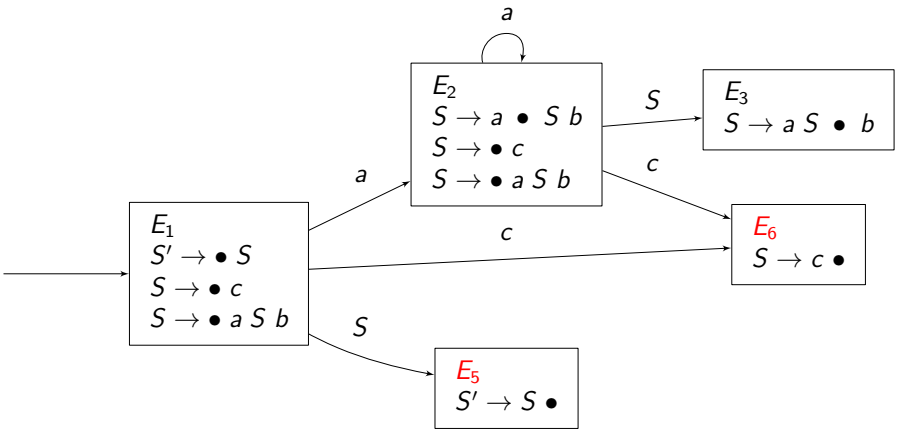
Maintenant que notre état initial est *saturé*, on va pouvoir le nommer (E_1) et chercher les transitions possibles depuis cet état. Pour cela, il faut regarder tous les symboles (terminaux mais aussi non-terminaux) qui suivent les point. Cela nous mène à 3 nouveaux états, dont deux n'ont pas besoin d'être saturés, car le point apparaît en fin d'item (notion d'item puits).



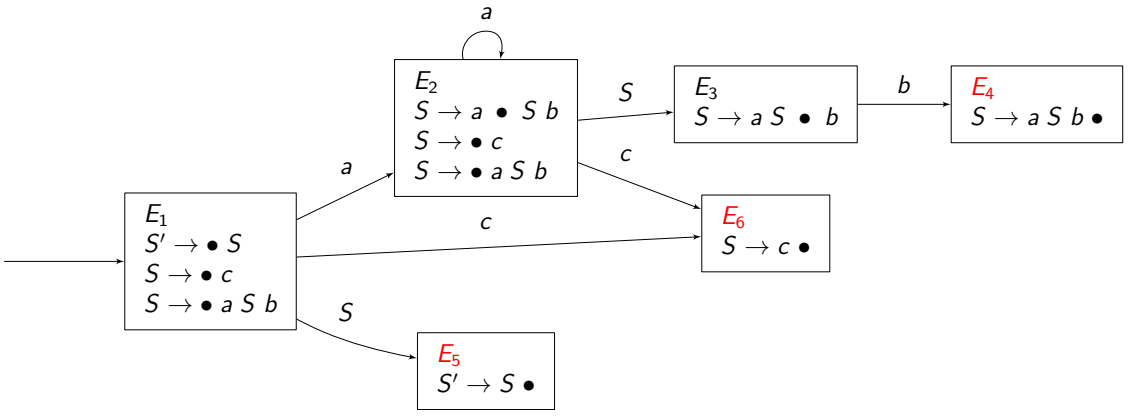
Pour l'item E_2 en revanche, il faut le saturer, grâce aux deux mêmes règles que pour E_1, on arrive à deux nouveaux items dans cet état :



Depuis E_5 et E_6, aucune transition n'est possible, ce sont des états puits qui donneront lieu à des réductions (voire même la fin de la reconnaissance pour E_5). Nous nous intéressons donc maintenant aux transitions depuis E_2. On s'aperçoit rapidement que le symbole a nous mène à nouveau vers E_2, que le symbole c nous mène vers E_6 et que S va donner lieu à un nouvel état :

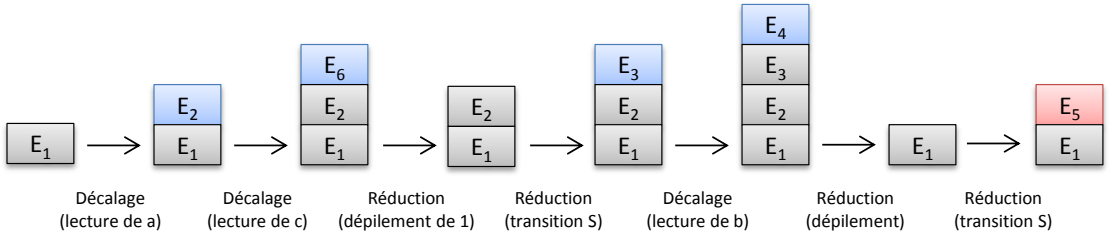


Cet état est déjà saturé car le point est suivi d'un symbole terminal. Aussi il ne possède qu'une seule transition à partir du symbole b vers un nouvel état puits. Notre automate est maintenant complet :



▶ Utilisation

Une fois que l'automate est construit il s'agit maintenant de l'utiliser. On initialise l'automate à pile avec l'état de départ sur la pile (E_1 dans l'exemple). L'automate accepte le mot lorsqu'il a sur le sommet de la pile l'état final (E_5 dans l'exemple). La lecture du mot acb donnera lieu aux étapes suivantes :



Il y a deux types de transitions :

— Lecture (transition de décalage, ou *shift* en anglais) : un symbole terminal est lu et la transition existe dans l'automate, dans ce cas-là il suffit d'empiler le nouvel état destination

— Réduction (*reduce* en anglais) : on arrive dans un état puits (un seul item puits, qui se termine par un point). Cela signifie que l'on a reconnu un symbole non-terminal (celui qui se trouve en partie gauche de l'item). On va alors procéder en deux étapes :

1. Dépiler le nombre de symboles présents en partie droite de l'item ;

2. Appliquer la transition (donc empiler) du symbole non-terminal à l'état qui se trouve maintenant en sommet de pile.

▶ Tables d'analyse

On peut décrire le comportement d'un analyseur LR grâce à des tables d'analyse. Chaque état est décrit par une ligne. Pour chaque symbole terminal, on indique quelle est l'action à effectuer :

— E_n signifie décaler et aller à l'état E_n
— R_n signifie réduire la règle n
— A signifie accepter

Pour chaque symbole non-terminal, on indique simplement le numéro de l'état successeur. Cette table fait donc la séparation entre la réduction et l'application du symbole non-terminal réduit.
Avec l'exemple précédent, on aurait donc la table d'analyse LR(0) suivante :

État	a	b	c	\$	S
E_1	E_2		E_6		E_5
E_2	E_2		E_6		E_3
E_3		E_4			
E_4	R_3	R_3	R_3	R_3	
E_5				A	
E_6	R_2	R_2	R_2	R_2	

On remarque que les états pour lesquels on fait une réduction ont la même information sur toute la ligne, quel que soit le symbole à lire (symbole qui ne sera pas lu). On remarque aussi l'absence de réduction de la règle 1 car cette réduction est équivalente à l'acceptation A.

▶ Implémentation

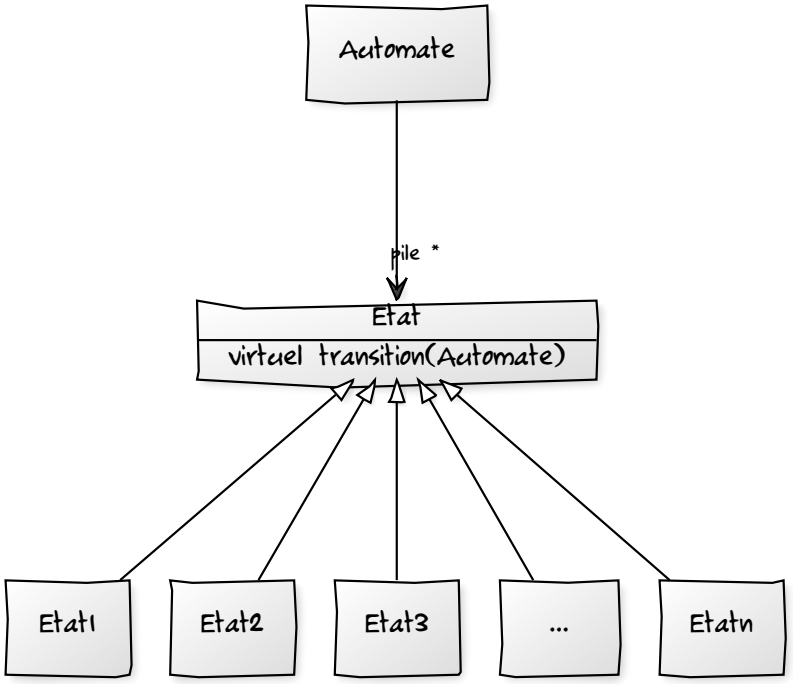
Une seule information par case d'un tableau ne suffit pas à coder ces transitions. Il faut en effet savoir quel est l'état cible pour chaque couple état/symbole mais aussi savoir quelle action il va falloir effectuer parmi les suivantes :

— Décalage : on devra alors uniquement stocker l'état d'arrivée. Cette action aura pour effet d'empiler l'état cible.
— Réduction : il faudra stocker le nombre de dépilements à faire ainsi que le symbole non-terminal concerné. Cette action aura pour effet de dépiler des états, puis de provoquer automatiquement la transition sur le symbole non-terminal au nouvel état qui se trouve en sommet de pile.
— Acceptation.
— Erreur.

Suivant l'implémentation, l'action erreur pourra ou non être une case vide dans le tableau. On remarque aussi que les états donnant lieu à des réductions peuvent être tout simplement omis de l'automate, à partir du moment où l'on stocke le nombre de dépilements ainsi que le symbole non-terminal qui sera réduit.

▶ Design pattern State

Le design pattern State permet de modéliser aisément un automate à états fini. Suivant le contexte (l'état) dans lequel est l'automate, l'action sera différenciée. Avec un automate à piles, l'automate stocke une pile d'états. Chaque état a une fonction de transition pour laquelle le contexte (donc l'automate) est donné afin de pouvoir modifier cette pile :



L'exemple complet en fin de cours reprend ce formalisme.

▶ Limitations

La méthode LR(0) possède des limitations qui l'empêchent de fonctionner dans certaines configurations. En particulier, au moment de la construction de l'automate, il se peut que l'on ait plusieurs possibilités.
C'est ce que l'on appelle des conflits. Une grammaire aussi simple que :

$$A \to a \mid \epsilon$$

va échouer à l'analyse dès le début car elle ne saura s'il faut décaler (donc lire le a) ou réduire par $A \to \epsilon$.

▶ Conflits LR(0)

Il existe deux types de conflits dans un automate LR(0) :

— Conflit décalage/réduction si au sein d'un même état on a des items de cette forme :
$[X \to \ldots \bullet a \ldots]$
$[Y \to \alpha\bullet]$

— Conflit réduction/réduction si au sein d'un même état on a des items de cette forme :
$[X \to \alpha\bullet]$
$[Y \to \beta\bullet]$

La présence même de ces conflits empêche l'utilisation de l'automate.
Pourtant, la simple lecture du prochain caractère peut dans certains cas permettre de lever cette ambigüité. C'est ce que va faire l'algorithme LR(1) et plus particulièrement une version simplifiée SLR(1).

▶ Grammaire LR(0)

Grammaire LR(0) Une grammaire est dite LR(0) si sa table des actions LR(0) ne comprend aucun conflit.

5.4 Analyseurs SLR(1)

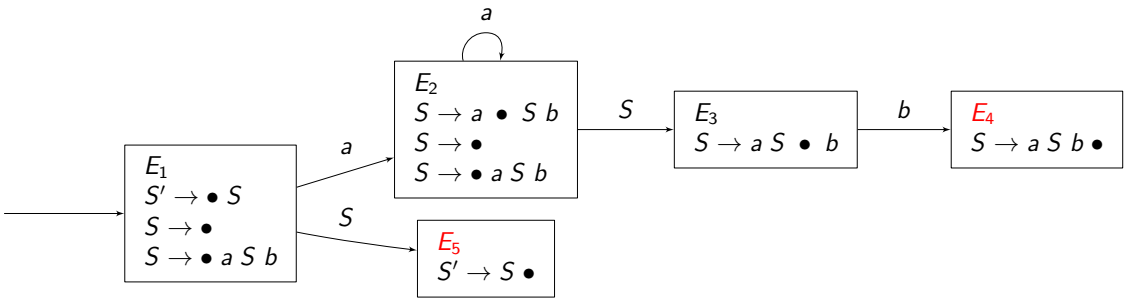
La méthode d'analyse SLR(1) va utiliser la lecture d'un caractère en avance (d'où le (1)) pour lever certaines ambigüités dans la table d'analyse LR(0). Pour résumer, on ne fait une réduction d'un symbole X que si le prochain terminal à lire fait partie de ses suivants (vous vous souvenez ?). L'analyseur SLR(1) utilise exactement le même automate que l'analyseur LR(0), il se contente de lever des conflits dans ce dernier.
Remarque : les analyseurs SLR(1) sont donc strictement plus puissants que les analyseurs LR(0).

▶ Exemple

Pour illustrer le principe des analyseurs SLR(1) nous allons étudier une grammaire qui possède des conflits LR(0) mais aucun conflit SLR(1). La voici :

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & a\ S\ b \\ S & \rightarrow & \epsilon \end{array}$$

Voici son automate LR(0) :



Les états E_1 et E_2 ont des conflits décalage réduction :
— Faut-il lire le symbole a en entrée ?
— Faut-il réduire la règle $S \rightarrow \epsilon$?
Intuitivement, la règle $S \rightarrow \epsilon$ ne devrait être réduite que lorsque le symbole suivant est b car cela indique la fin des a. Cela peut être formalisé grâce à l'ensemble des suivants que nous avons étudiés en analyse prédictive.
Commençons par calculer la liste des non-terminaux qui peuvent être nuls : $\mathcal{N} = \{S, S'\}$. Maintenant les premiers :

$$\mathcal{P}(S) = \mathcal{P}(S') = \{a, \epsilon\}$$

Et les suivants :

$$\begin{array}{l} \mathcal{S}(S) = \{b, \$\} \\ \mathcal{S}(S') = \{\$\} \end{array}$$

Cette analyse nous permet de voir que a ne fait pas partie des suivants de S, donc les conflits sont levés. Dans la table des actions, il faut donc indiquer la réduction de $S \rightarrow \epsilon$ pour les couples $(E_1, \$)$ et (E_2, b).

▶ Table d'analyse

La table d'analyse correspondant à l'exemple précédent sera donc :

État	a	b	$\$$	S
E_1	E_2	R_3	R_3	E_5
E_2	E_2	R_3	R_3	E_3
E_3		E_4		
E_4	R_2	R_2	R_2	
E_5			A	

On voit donc apparaître des actions de réduction qui ne sont pas sur la totalité de la ligne, c'est là que se situe l'apport de SLR(1) par rapport à LR(0).

▶ Implémentation

Concrètement, l'implémentation d'un automate SLR(1) ressemble à celle d'un automate LR(0). Dans notre implémentation de l'automate LR(0) vue précédemment, nous avons simplifié la table en supprimant tous les états puits qui mènent forcément à des réductions. Dans la version SLR(1), il ne faudra pas faire cette simplification. De plus, il faudra disposer d'une fonction permettant de sonder le prochain caractère à lire mais qui ne lit pas ce caractère.

▶ Ambigüité

Une grammaire ambigüe ne peut pas être LR(0) ni SLR(1).

Inversement, une grammaire LR(0) ou SLR(1) ne peut pas être ambigüe.

⚠ Mais il existe des grammaires non ambigües qui ne sont pas SLR(1) (ni LR(0) *a fortiori*) !

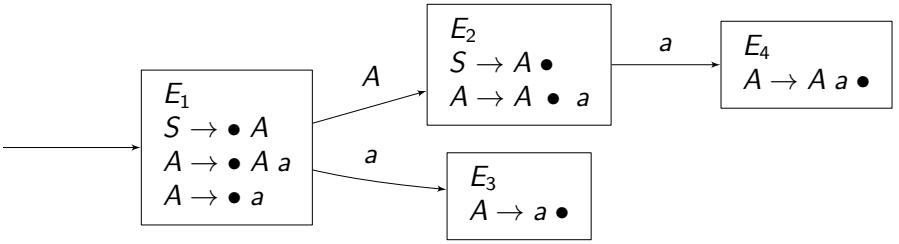
▶ Récursivité gauche et droite

Les analyseurs de type SLR(1) sont capables de traiter des récursivités gauche et droite, cependant, nous allons voir que la première est bien plus préférable que la deuxième par un exemple très simple.

Étudions deux grammaires équivalentes qui génèrent le même langage $\{a^n | n > 0\}$:

G_g	G_d
$S \to A$	$S \to A$
$A \to A\ a$	$A \to a\ A$
$A \to a$	$A \to a$

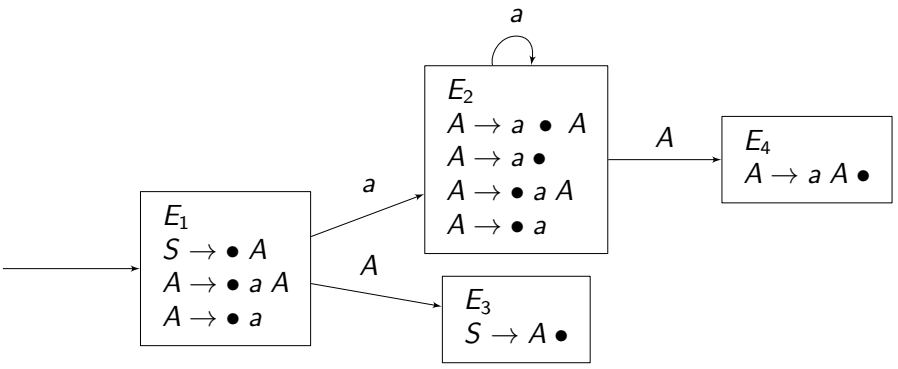
Voici l'automate de G_g :



À la première lecture d'un a, l'automate passe dans l'état E_3 puis réduit $A \to a$ et passe ainsi dans l'état E_2. La réduction $S \to A$ n'a lieu que si le symbole suivant est le symbole de fin \$.

À chaque nouvelle lecture d'un a, l'automate va passer dans l'état E_4 puis réduire $A \to A\ a$ et ainsi repasser dans l'état E_2. Le nombre d'états empilés ne dépend donc pas du nombre de symboles lus en entrée. L'automate est performant.

Avec G_d maintenant :



À la première lecture d'un a, l'automate passe dans l'état E_2 mais n'effectue pas la réduction de $A \to a$ que si on a atteint la fin de la lecture (car $\mathcal{S}(A) = \{\$\}$). On empile donc l'état E_2. Une nouvelle lecture d'un a empilera à nouveau l'état E_2.
Ce n'est qu'à la fin de l'entrée que l'on passera dans l'état E_4 grâce à la réduction de $A \to a$. Le nombre d'états empilés est donc égal à la taille du mot en entrée. Cet analyseur marche mais n'est pas performant. Ce sont les récursivités droites qui sont à la source de ce problème. Il faudra les éviter dans un analyseur de type SLR(1) et d'une manière générale dans les analyseurs LR.

▶ Grammaire SLR(1)

Grammaire SLR(1) Une grammaire est dite SLR(1) si sa table des actions ou table d'analyse SLR(1) ne comprend aucun conflit.

5.5 Analyseurs LR(1)

Les analyseurs SLR(1) sont plus fins que les analyseurs LR(0) car ils utilisent une information sur ce qui peut suivre un symbole non-terminal. Malheureusement, cette information est globale et ne dépend pas du contexte dans lequel on trouve ce non-terminal, ce qui peut mener à des conflits. L'analyseur LR(1) pousse un petit peu plus loin le raisonnement en proposant un mécanisme qui identifie les suivants possibles d'un non-terminal dans le cadre de son item directement.

▶ Items généralisés

Dans un item généralisé, on rajoute une information sur la liste des symboles qui peuvent suivre le non-terminal en partie gauche de l'item :

$$[X \to \alpha \bullet \beta, a/b/\ldots]$$

où $\alpha, \beta \in V^*$ et $a, b, \ldots \in T \cup \{\$\}$.
Cet item généralisé signifie : "En cherchant à dériver de X un mot $w = uv \in T^*$, on a déjà dérivé un mot u de α et il reste à dériver v de β. **Le symbole X sera suivi d'un des terminaux a, b, ...**."

⚠ Dans l'exemple précédent, on aura forcément $a, b \in \mathcal{S}(X)$. Si ce n'est pas le cas, il y a un souci. La finesse introduite dans l'analyse LR(1) est que l'ensemble des suivants n'est pas entièrement représenté dans les items généralisés.

Certaines notations ne fusionnent pas les items généralisés et écrivent un item par symbole suivant :

$$[X \to \alpha \bullet \beta, a]$$
$$[X \to \alpha \bullet \beta, b]$$
$$[\ldots]$$

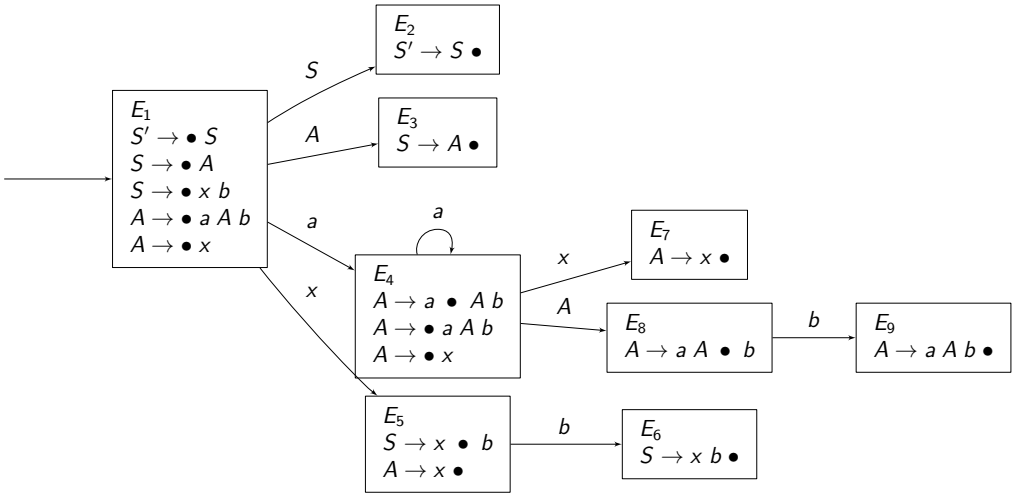


FIGURE 5.3 – Automate SLR(1) de la grammaire G_c

▶ Saturation

Pour saturer les items, il va falloir prendre en compte les suivants. Voici la méthode :
- Considérons l'item suivant à saturer : $[X \to \alpha \bullet Y\beta, a]$
- On ajoute à cet état tous les items correspondant aux productions du type $Y \to \gamma$
- Pour chacune de ces productions on ajoute comme suivants possibles les premiers de $\beta\ a$
- Cela signifie que a peut en faire partie si $\beta \in \mathcal{N}$, mais pas de manière systématique.

▶ Exemple

Pour illustrer la construction LR(1), nous allons étudier la grammaire G_c qui a un conflit au sens SLR(1) :

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

Son automate SLR(1) se trouve figure 5.3, page 58.
L'état E_5 possède un conflit au sens LR(0) car on ne sait pas si on doit réduire $A \to x$ ou non. L'analyse SLR(1) ne va rien apporter car $b \in \mathcal{S}(A)$: on ne sait toujours pas quoi faire. Il faut donc recourir à quelques chose de plus puissant qui saurait faire la différence entre une production $A \to x$ suivie de \$ et la même production suivie d'un b.
L'automate LR(1) se trouve figure 5.4, page 59. Les états E_1, E_2, E_3, E_4 et E_5 sont calculés d'une manière assez similaire à celle de l'automate SLR(1), on se contente d'ajouter l'information des suivants potentiels, ici \$. C'est l'état E_6 qui va être différent, en effet, lors de la saturation de $A \to a\ \bullet\ A\ b,\ \$$, le symbole suivant A est b, l'état saturé est donc :

E_6

$A \to a\ \bullet\ A\ b,$	\$
$A \to \bullet\ a\ A\ b,$	b
$A \to \bullet\ x,$	b

L'état E_{10} est quant à lui quasiment identique à E_6, la seule chose qui le différencie est l'ensemble des suivants de la règle $A \to a\ \bullet\ A\ b$:

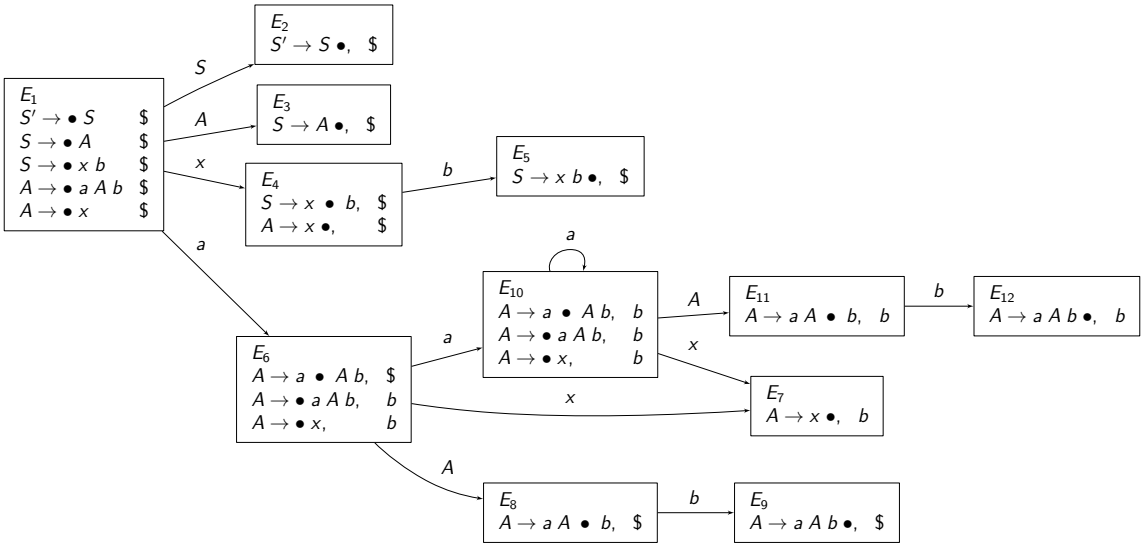
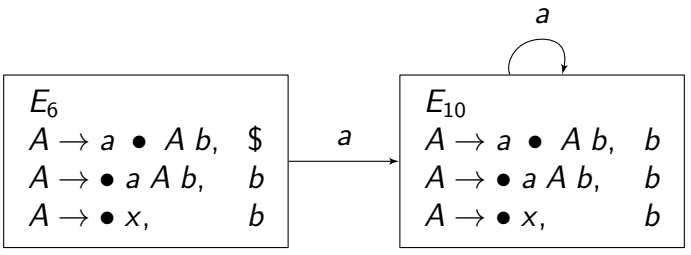


FIGURE 5.4 – Automate LR(1) de la grammaire G_c



Alors que l'état E_6 signifie que l'on a commencé à reconnaître un a dans la règle $A \rightarrow a \bullet A\ b$ et qu'il reste à reconnaître un A et lire b pour terminer l'analyse, la règle E_{10} indique qu'après avoir reconnu un A et un b, on n'aura pas fini l'analyse, on devra forcément avoir un symbole b après.
La lecture du premier a va donc empiler l'état E_6, alors que toutes les suivantes vont empiler l'état E_{10}. Dans l'automate SLR(1) on ne faisait aucune différence entre ces deux actions.

▶ Table d'analyse

La table d'analyse LR(1) est similaire à celle SLR(1), les réductions sont maintenant très ciblées.

État	a	b	x	\$	A	S
E_1	E_6		E_4		E_3	E_2
E_2				A		
E_3				R_2		
E_4		E_5		R_5		
E_5				R_3		
E_6	E_{10}		E_7		E_8	
E_7		R_5				
E_8		E_9				
E_9				R_4		
E_{10}	E_{10}		E_7		E_{11}	
E_{11}		E_{12}				
E_{12}		R_4				

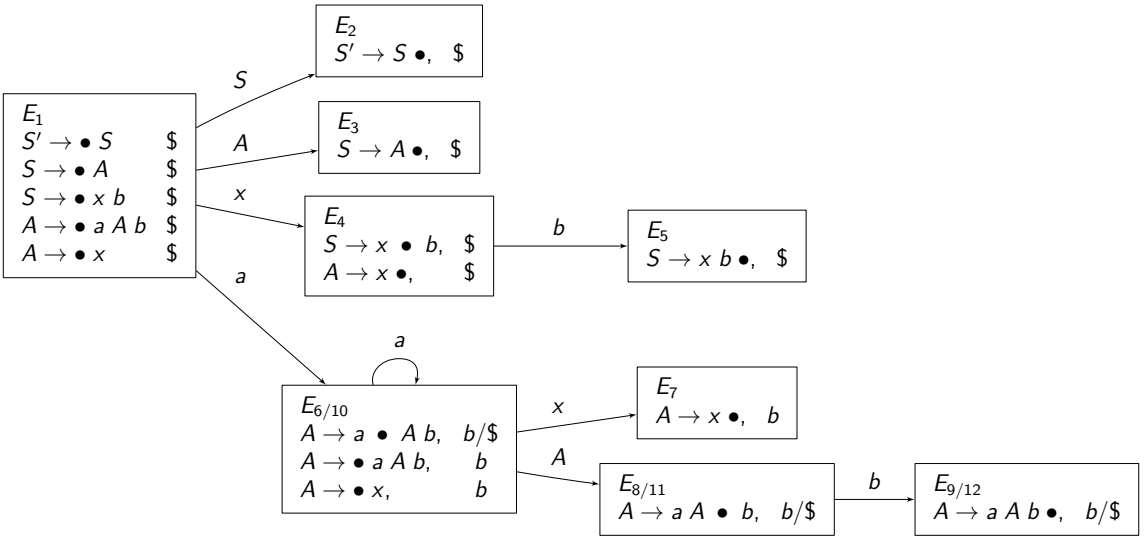


FIGURE 5.5 – Automate LALR(1) issu de la fusion des états de l'automate LR(1) de la grammaire G_c

▶ Grammaire LR(1)

Grammaire LR(1) Une grammaire est dite LR(1) si sa table des actions ou table d'analyse LR(1) ne comprend aucun conflit.

▶ Conclusion

Les analyseurs LR(1) sont strictement plus puissants que les analyseurs LR(0) et SLR(1). D'une manière générale un automate LR(1) aura plus d'états que l'automate LR(0) ou SLR(1). Si une grammaire est SLR(1), il n'y a donc aucun intérêt à construire un analyseur LR(1) (sauf si on vous le demande en DS 🙂). Les analyseurs LR(1) sont puissants mais lourds, c'est pourquoi un intermédiaire entre SLR(1) et LR(1) a été développé, il s'agit des analyseurs LALR(1) qui sont utilisés dans beaucoup de générateurs (comme bison par exemple).

5.6 Analyseurs LALR(1)

▶ Introduction

L'idée des analyseurs LALR(1) et de factoriser les états de l'automate LR(1) en se basant sur leur *cœur*. Le cœur d'un état LR(1) est l'ensemble de ses items sans les symboles de prédiction. Dans l'exemple précédent, les états suivants ont le même cœur : $\{E_6, E_{10}\}$, $\{E_8, E_{11}\}$ et $\{E_9, E_{12}\}$.

Prenons l'exemple trivial des états $\{E_9, E_{12}\}$, on voit assez facilement que les fusionner ne pose aucun souci, car tous deux sont des réductions de la production $A \to aAb$. Pour les fusionner, il suffira d'indiquer dans les symboles de prédiction l'union des symboles de prévision des deux états concernés, donc $b/\$$.

L'automate après fusion des trois paires d'états se trouve figure 5.5 de la page 5.5.

La table d'analyse LALR(1) correspondante :

État	a	b	x	\$	A	S
E_1	$E_{6/10}$		E_4		E_3	E_2
E_2				A		
E_3				R_2		
E_4		E_5		R_5		
E_5				R_3		
$E_{6/10}$	$E_{6/10}$		E_7		$E_{8/11}$	
E_7		R_5				
$E_{8/11}$		$E_{9/12}$				
$E_{9/12}$		R_4		R_4		

Le calcul des transitions est assez intuitif : lorsqu'une transition existe entre deux états de l'automate LR(1), il faut la retrouver dans les états fusionnés. Si ce calcul n'est pas possible, la grammaire n'est pas LALR(1).

▶ Grammaire LALR(1)

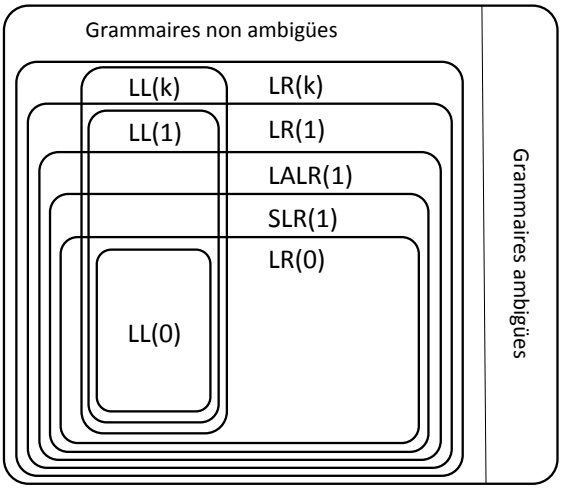
Grammaire LALR(1) Une grammaire est dite LALR(1) si le processus de fusion de ses états LR(1) est possible et n'engendre aucun conflit.

⚠ Une grammaire ne peut être LALR(1) que si elle est LR(1). Logiquement, il existe des grammaires LR(1) qui ne sont pas LALR(1). LR(1) est donc strictement plus puissant que LALR(1).

— Les grammaires SLR(1) sont aussi LALR(1) car le choix de l'action est guidé par une analyse contextuelle dans le cas de LALR(1) comme dans le cas de LR(1) alors que dans SLR(1) cette analyse est statique.

— Sur l'exemple précédent, on voit d'ailleurs un exemple de grammaire qui n'est pas SLR(1), mais qui est LALR(1).

Voici donc sous forme schématique les ensembles de grammaires LR(0), LR(1), SLR(1) et LALR(1) ainsi que leurs homologues LL :



▶ Construction directe

L'inconvénient de la méthode précédente est qu'elle nécessite la construction de l'automate LR(1) pour ensuite faire la fusion des états. Or la construction de l'automate LR(1) est très lourde. Nous allons voir comment construire directement un automate LALR(1) à partir de la grammaire.
Il existe plusieurs façons de construire de manière directe l'automate LALR(1), nous allons parler de deux méthodes :

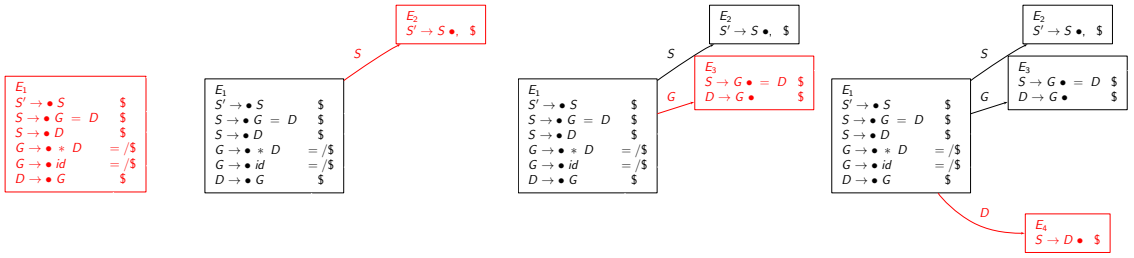


FIGURE 5.6 – Étapes 1, 2, 3 et 4 de la construction LALR(1) incrémentale

— Fusion incrémentale. Cet algorithme consiste à construire l'automate LR(1) mais en fusionnant les états lorsque leur cœur est identique.
— Propagation. Cet algorithme introduit la notion de noyau d'un ensemble d'items. Les symboles de prédiction sont propagés d'un état à un autre.

▶ Fusion incrémentale

C'est la méthode la plus naturelle lorsque l'on sait construire d'automate LR(1). Sa mise en œuvre consiste à :

— Commencer à construire l'automate LR(1)
— Lorsqu'un état va être créé avec le même cœur qu'un état existant, on procède tout de suite à la fusion avec ce dernier en faisant attention aux symboles de prévisions associés.

Illustration sur la grammaire suivante :

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & G = D \\ S & \rightarrow & D \\ G & \rightarrow & *D \\ G & \rightarrow & id \\ D & \rightarrow & G \end{array}$$

Les figures 5.6 à 5.10 montrent le déroulement complet de l'algorithme sur cette grammaire.

▶ Propagation

Cet algorithme commence par introduire la notion de noyau d'un ensemble d'items.

Noyau d'un ensemble d'items Le noyau d'un ensemble d'items LR(0) est un sous-ensemble de celui-ci ne contenant que l'item initial $[S' \rightarrow \bullet S]$ ou les items ne commençant pas par $\bullet$.

On remarque que le noyau d'un ensemble d'items LR(0) représente l'ensemble d'items car tous les items qui n'en font pas partie ont été générés par l'opération de fermeture (saturation).

Voici l'automate LR(0) épuré (noyau uniquement) de la grammaire étudiée précédemment :

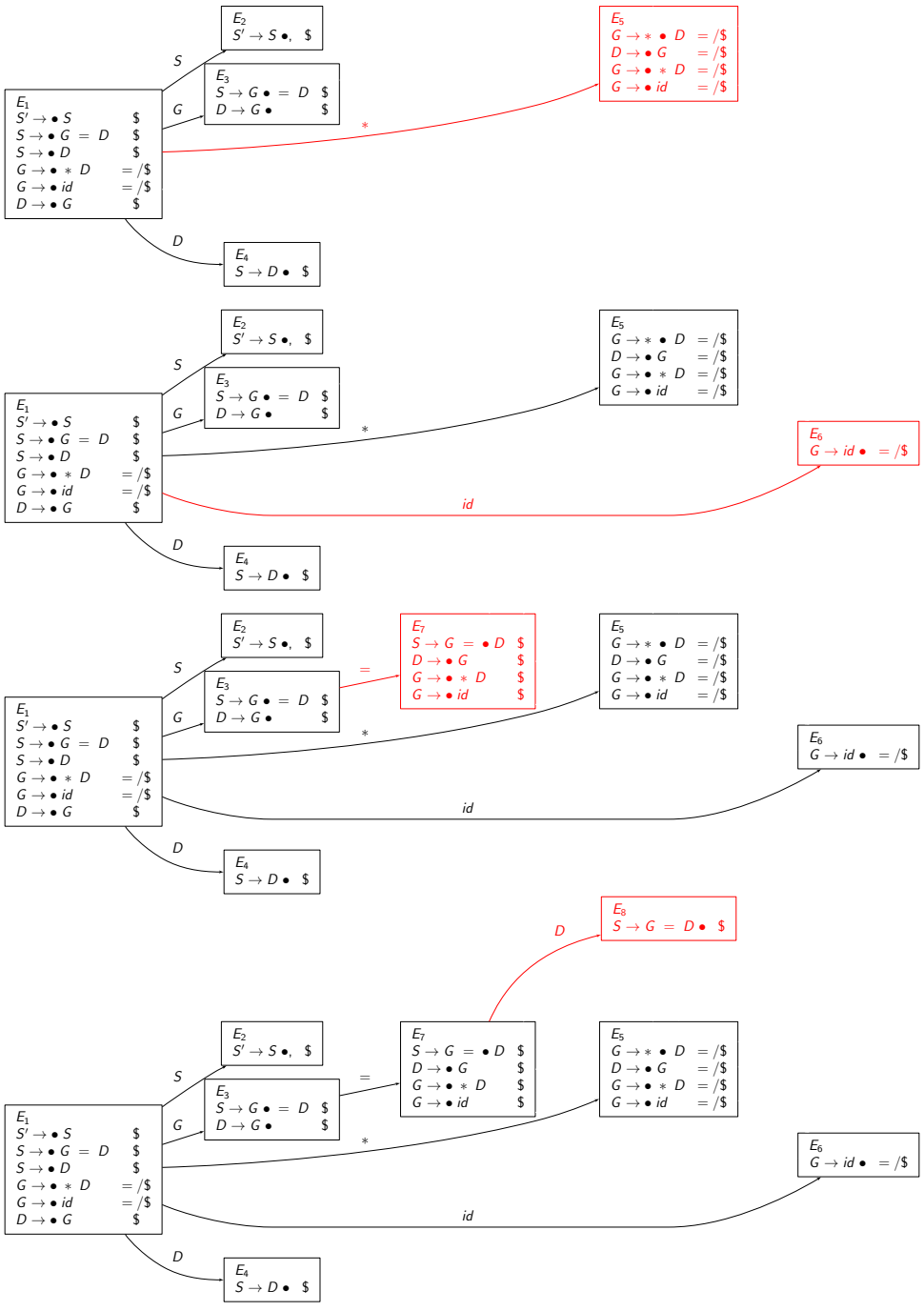


FIGURE 5.7 – Étapes 5, 6, 7 et 8 de la construction LALR(1) incrémentale

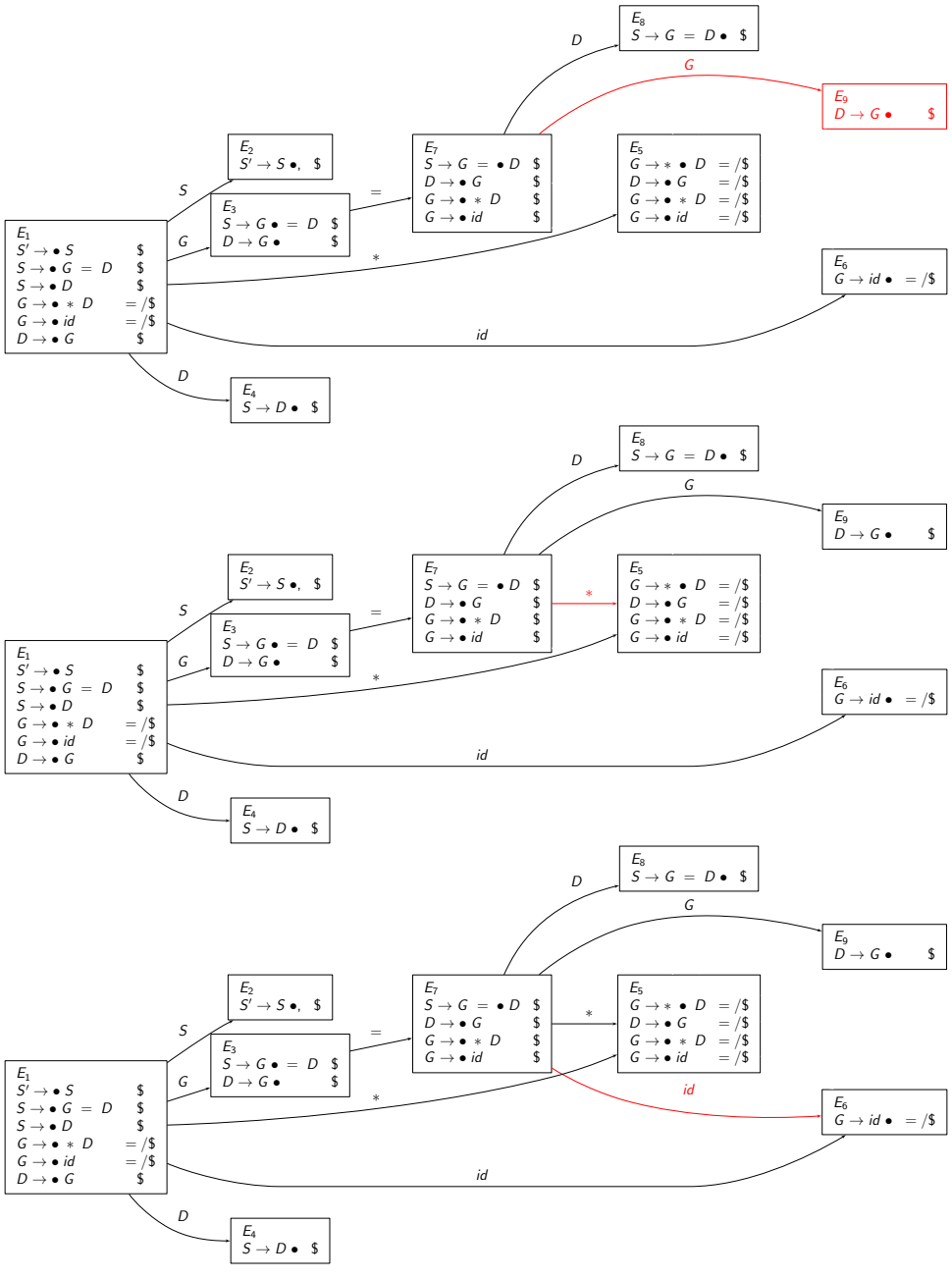


FIGURE 5.8 – Étapes 9, 10 et 11 de la construction LALR(1) incrémentale

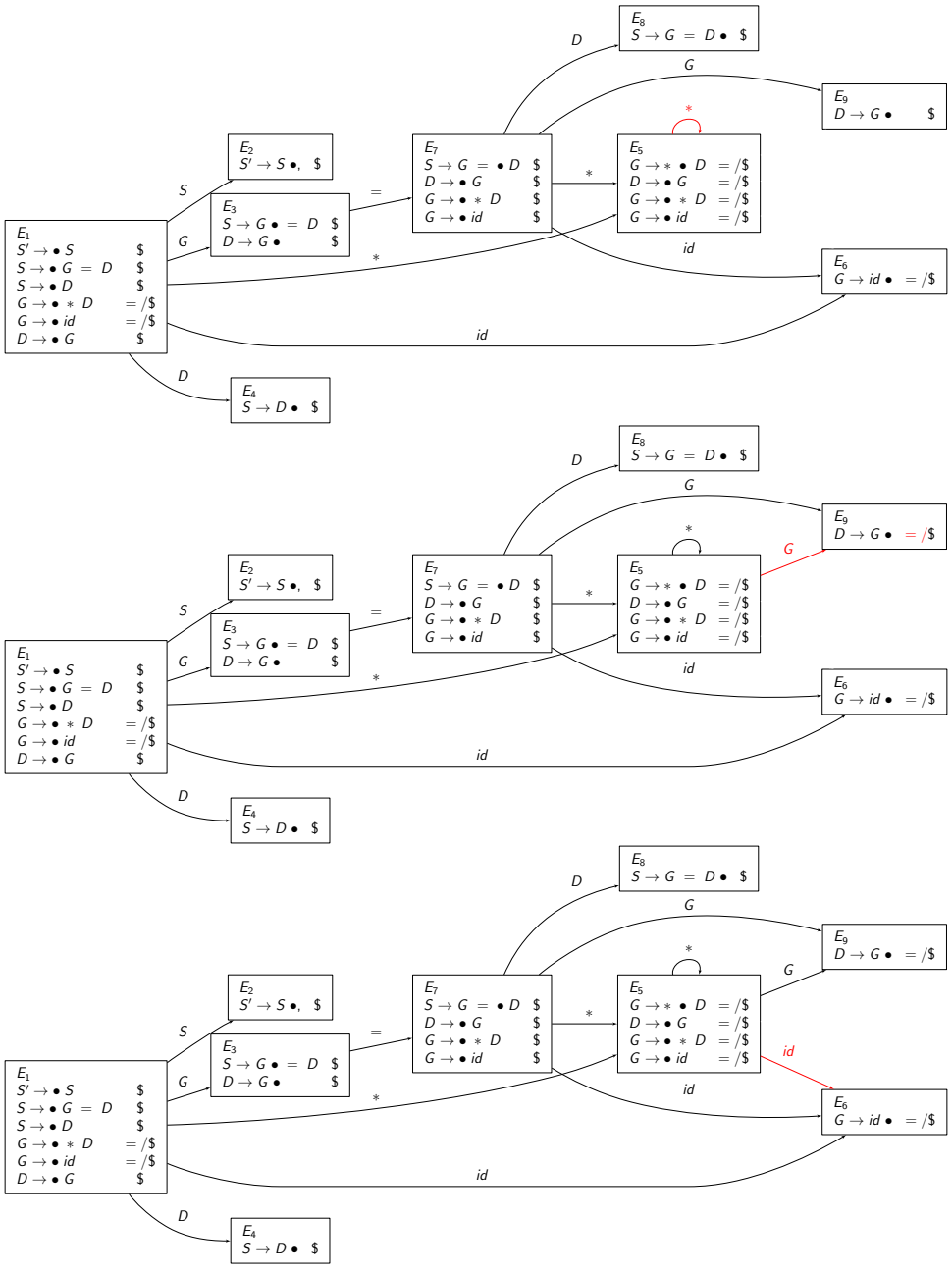


FIGURE 5.9 – Étapes 12, 13 et 14 de la construction LALR(1) incrémentale. Remarque : l'étape 13 ajoute le symbole $=$ à E_9.

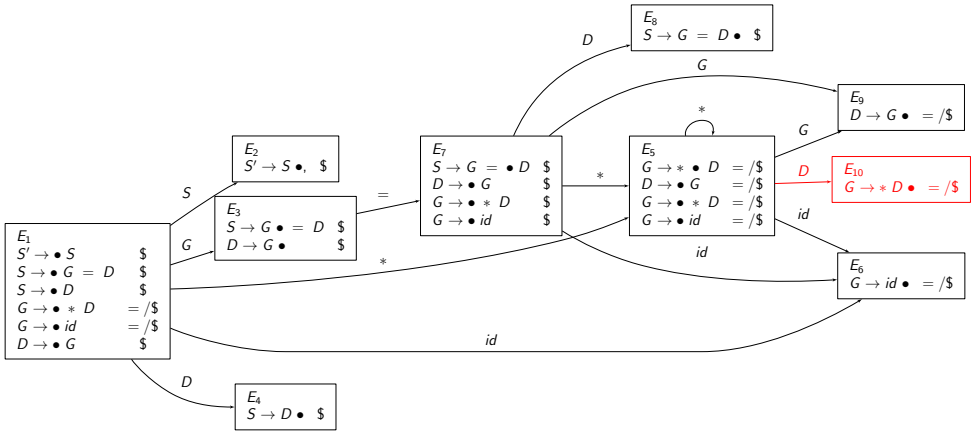
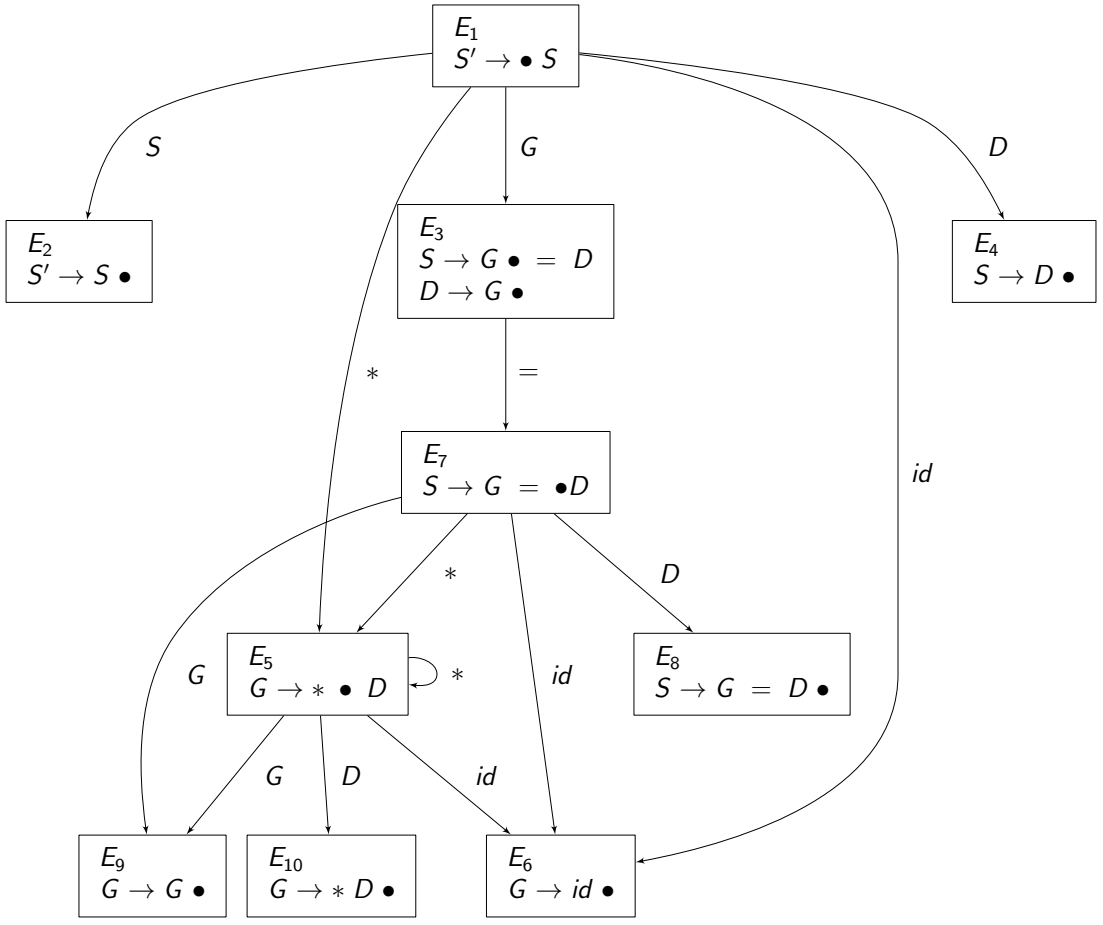


FIGURE 5.10 – Dernière étape de la construction LALR(1) incrémentale



Afin d'étudier les mécanisme de propagation des symboles de prévision sur cet automate, nous allons prendre chacun des item du noyau et former la fermeture fictive de cet item accompagné d'un symbole de prévision factice #.

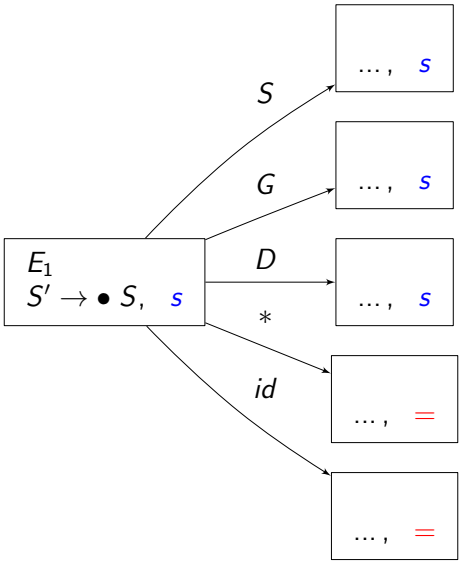
Prenons l'exemple de l'item $S' \rightarrow \bullet S$, nous lui ajoutons le symbole de prévision factice #, ce qui

donne $S' \rightarrow \bullet S, \#$. Voici maintenant la fermeture associée :

$$\begin{array}{ll} S' \rightarrow \bullet S, & \# \\ S \rightarrow \bullet G = D, & \# \\ S \rightarrow \bullet D, & \# \\ G \rightarrow \bullet * D, & \#/= \\ G \rightarrow \bullet id, & \#/= \\ D \rightarrow \bullet G, & \# \end{array}$$

Cet analyse nous permet de déduire :
- Qu'il existe des symboles de prévision spontanés dans les états associés aux transitions sur $*$ et id.
- Que les symboles de prévision associés à cet item vont être propagés vers les états associés aux transitions sur S, G et D.

Ce que l'on peut schématiser par :



Le reste de l'analyse des items du noyau ne fera pas apparaître de nouveaux symboles spontanés et montrera que toutes les transitions LR(0) sont des propagations de symboles.

⚠ Attention, le point de départ de la propagation n'est pas lié à un état mais à un item du noyau de chaque état, il ne faut pas faire la confusion.

Le reste de l'algorithme est simple, on initialise les items avec leurs symboles de prévision spontanés ainsi que \$ pour l'item de départ, et on propage de manière itérative jusqu'à obtention du point fixe.

Les figures 5.11 et 5.12 montrent le déroulement complet de la propagation jusqu'au point fixe figure 5.13.

▶ Implémentation

L'implémentation d'un automate LALR(1) est assez proche de celle d'un automate SLR(1) ou LR(0), il s'agit d'un automate à pile déterministe. La pile pourra au choix contenir :
- Uniquement les états parcourus. Dans ce cas, les actions de réduction devront connaître le nombre de symboles en partie droite ainsi que le symbole non-terminal en partie gauche de la règle réduite.
- Les états ainsi que les symboles lus ou réduits. Lorsqu'un symbole terminal est décalé, on le met dans la pile, lorsqu'une réduction est effectuée, en plus de dépiler les états, on dépile les symboles (terminaux ou non) de la partie droite de la règle et on empile le symbole non-terminal en partie gauche de la production réduite. Cette forme d'implémentation est

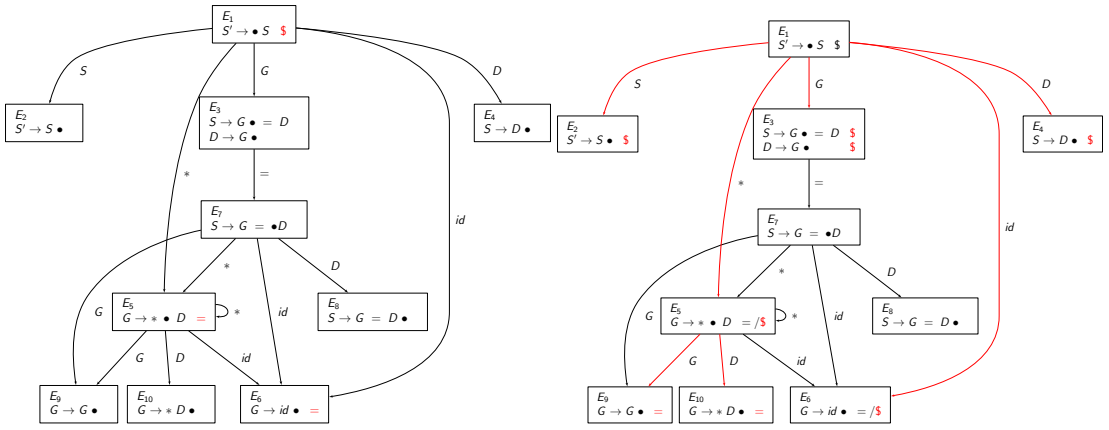


FIGURE 5.11 – Étapes 1 et 2 de la propagation des symboles LALR(1)

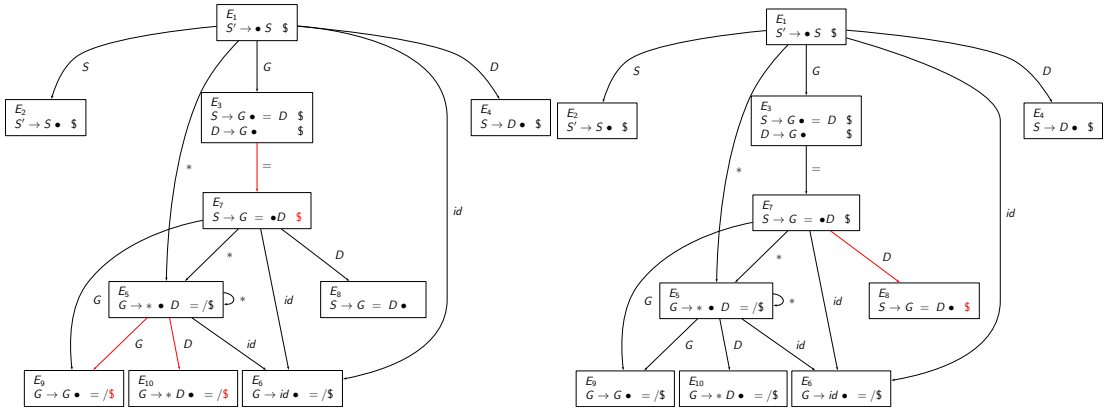


FIGURE 5.12 – Étapes 3 et 4 de la propagation des symboles LALR(1)

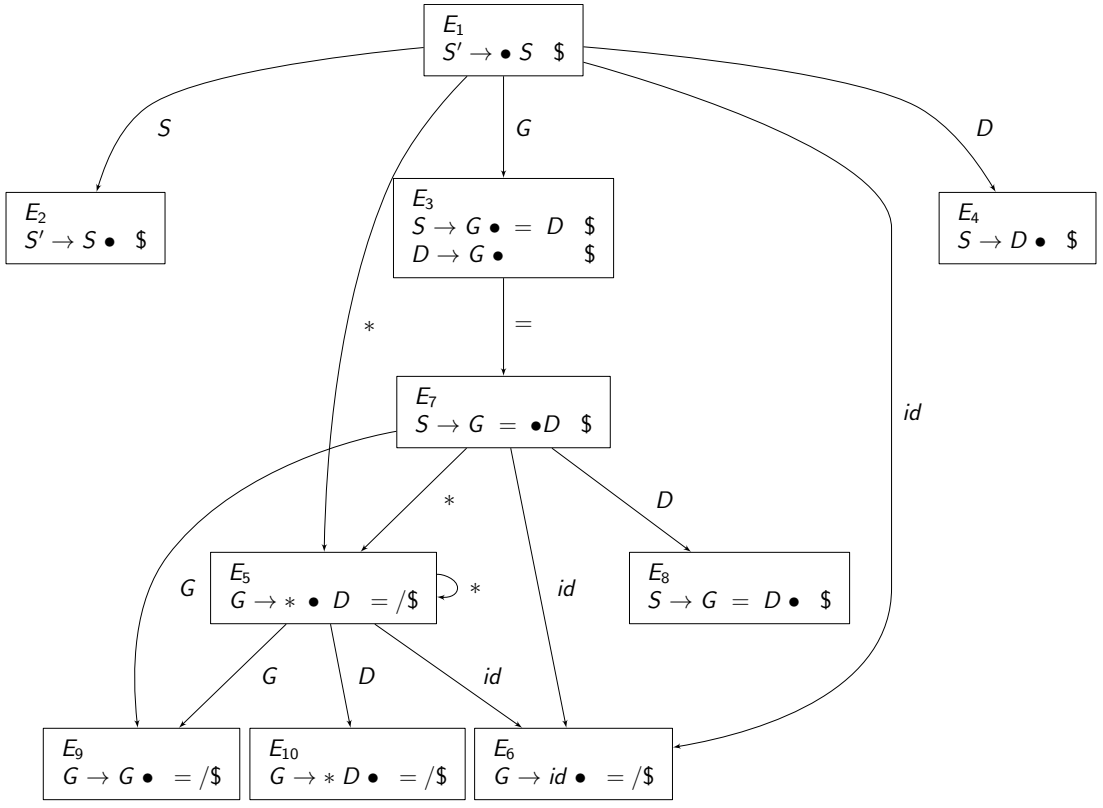


FIGURE 5.13 – Dernière étape (point fixe) de la propagation LALR(1)

surtout utile pour mieux exhiber l'exécution de l'automate ou pour afficher plus facilement des messages d'erreurs, mais la première est suffisante.

▶ bison

Bison est un générateur d'analyseur syntaxique de type LALR, nous l'étudierons en détail dans un chapitre séparé.

5.7 Exercices

▶ Grammaire de grammaire

Voici une grammaire permettant de décrire une grammaire formelle hors-contexte :

$$\begin{array}{lcl} G & \rightarrow & G\ R\ p \\ G & \rightarrow & \epsilon \\ R & \rightarrow & n\ f\ L \\ L & \rightarrow & L\ n \\ L & \rightarrow & L\ t \\ L & \rightarrow & \epsilon \end{array}$$

Les symboles terminaux sont les suivants : n, t, p et f. G symbolise la grammaire, R une règle, et L une liste de symboles terminaux (t) ou non-terminaux (n). f est une flèche et p un séparateur de type point-virgule. Le symbole ϵ indique le mot vide (epsilon production).
Exercice 22 Augmenter cette grammaire
Exercice 23 Donner son automate LR(0)
Exercice 24 Identifier les états en conflit
Exercice 25 Calculer les symboles suivants de G, R et L
Exercice 26 Déduire et donner la table d'analyse SLR(1) qui permet de résoudre les conflits obtenus précédemment

▶ Expression postfixée

Nous allons considérer dans ce problème l'analyse d'une expression postfixée, c'est-à-dire où les opérandes sont données avant les opérateurs. Exemple : `3 b + 5 a - *` signifie en notation traditionnelle infixée `((3+b)*(5-a))`. Cette notation est pratique car elle permet l'évaluation à l'aide d'une simple pile et ne nécessite pas de parenthèses. Seuls les opérateurs binaires seront considérés. Voici la grammaire proposée :

$$\begin{array}{lcl} a & \rightarrow & exp \\ exp & \rightarrow & \text{NUM} \\ exp & \rightarrow & \text{VAR} \\ exp & \rightarrow & exp\ exp\ \text{OP} \end{array}$$

Remarque : les symboles sont typographiés à la façon de bison (terminaux en majuscule, et non-terminaux en minuscule).
Exercice 27 Donner l'automate LR(0) de cette grammaire
Exercice 28 Expliquer comment supprimer le conflit de cet automate grâce à une analyse SLR(1)

CHAPITRE 6
GRAMMAIRES ATTRIBUÉES

6.1 Grammaires attribuées

▶ Principe

À chaque symbole (terminal ou non-terminal) est associé un ou plusieurs attributs :
- Attribut hérité : descend l'arbre syntaxique (passe du nœud parent vers le nœud fils)
- Attribut synthétisé : remonte l'arbre syntaxique (passe de un ou plusieurs nœuds fils vers le nœud parent)

Les attributs permettent d'associer une sémantique aux nœuds.

▶ Exemple classique

L'évaluation d'une expression arithmétique peut se faire grâce à une grammaire attribuée. Les symboles terminaux ont une valeur associée lorsqu'il s'agit de nombres, alors que les opérateurs n'en ont pas. Un symbole non-terminal a un attribut synthétisé à partir de ses feuilles.
Il existe plusieurs façons de formaliser le calcul des attributs, dont une consiste à les écrire entre parenthèses :

$$E(x + y) \rightarrow E(x) \ + \ F(y)$$

Ici, l'attribut résultat est calculé en faisant la somme de x et y qui sont respectivement les attributs des non-terminaux E et F de la règle.
Une autre manière de noter :

$$E(z) \rightarrow E(x) \ + \ F(y), \ \{z = x + y\}$$

Cette dernière permet d'exhiber un algorithme plutôt qu'une formule mathématique.
Encore une autre façon :

$$E \rightarrow E_1 \ + \ F, \ \{E.val = E_1.val + F.val\}$$

qui permet au passage de donner plusieurs attributs à un même symbole.

▶ Types

Soit une règle de la forme :

$$A \rightarrow A_1 \ \ldots \ A_n$$

On considère deux types principaux de grammaires attribuées.

1. Grammaires L-attribuées : l'attribut de A_i dépend uniquement des attributs de symboles situés à sa gauche (A et $A_1 \dots A_{i-1}$).
2. Grammaires S-attribuées : l'attribut de A dépend des attributs de la partie droite.

En compilation, il est très fréquent d'avoir recours à une grammaire L-attribuée pour les déclarations de variables. En langage C par exemple, on indique un type en premier puis une liste d'identificateurs séparés par des virgules :

$$\begin{array}{lcll} D & \rightarrow & T\ L\ ; & \{L.type = T.type\} \\ T & \rightarrow & int \mid float & \\ L & \rightarrow & L_1\ ,\ id \mid id & \{L_1.type = L.type\} \end{array}$$

Le type des variables dans la liste L dépend de ce qui aura été défini pour T ce qui est bien conforme à une grammaire L-attribuée. Les grammaires L-attribuées sont adaptées à une analyse LL car l'information peut transiter de haut en bas et de gauche à droite.

⚠ La grammaire ci-dessus n'est pas du tout adaptée à une analyse LL (exercice, la transformer pour qu'elle le soit)

L'équivalent S-attribué de cette grammaire pourrait être :

$$\begin{array}{lcll} D & \rightarrow & T\ L\ ; & \{\ D.type = T.type \\ & & & \quad D.liste = L.liste\} \\ T & \rightarrow & int \mid float & \\ L & \rightarrow & L_1\ ,\ id & \{\ L.liste = L_1.liste \\ & & & \quad L.liste.insert(id)\} \\ L & \rightarrow & id & \{\ L.liste.insert(id)\} \end{array}$$

Cette version est très adaptée à une analyse LR car l'information ne fait que remonter.

6.2 Calcul

Pour calculer les attributs, plusieurs solutions :

— Après l'analyse, à partir de l'arbre syntaxique grâce à un parcours de l'arbre.
— Pendant l'analyse, en fonction du type d'analyse et des dépendances, cela peut être plus ou moins aisé...

Lorsque le calcul des attributs se fait après l'analyse, il faut d'une manière générale construire un graphe de dépendance entre attributs. Si ce graphe ne contient pas de cycle, alors on peut calculer tous les attributs. Pour un calcul des attributs pendant l'analyse, les grammaires L-attribuées sont adaptées aux analyseurs LL et inversement les grammaires S-attribuées sont adaptées aux analyseurs LR. Mais nous allons voir que même une grammaire S-attribuée peut être analysée en LL.

▶ Analyse descendante

Lorsque les attributs sont synthétisés (grammaire S-attribuée), c'est un peu raisonner à l'envers que de vouloir faire une analyse descendante. Pourtant, les analyseurs descendants sont les plus faciles à mettre en œuvre. Nous allons voir ici par un exemple comment procéder.

Dans une analyse descendante, l'arbre est parcouru de haut en bas, mais il ne faut pas oublier que chaque symbole non-terminal a une fonction associée qui va appeler récursivement les non-terminaux associés à la règle (en fonction du caractère à lire suivant).

Imaginons d'une manière simple que la fonction d'analyse, juste avant de retourner, renseigne un attribut. Cela correspond à un parcours postfixe donc de bas en haut.

Avec la grammaire des expressions étudiée dans le chapitre 4 (id a été enlevé pour permettre un calcul direct de l'expression) :

$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid -\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid /\ F\ T' \mid \epsilon \\ F & \rightarrow & (\ E\ ) \mid nb \end{array}$$

La figure 6.1 donne la table d'analyse de cette grammaire.

	$+$	$-$	$/$	$*$	$($	$)$	nb	\$
E					TE'		TE'	
E'	$+TE'$	$-TE'$				ϵ		ϵ
T					FT'		FT'	
T'	ϵ	ϵ	$/FT'$	$*FT'$		ϵ		ϵ
F					(E)		nb	

FIGURE 6.1 – Table d'analyse LL(1) de la grammaire des expressions

Imaginons maintenant que chaque symbole non-terminal possède sa fonction d'analyse LL(1) et renvoie un booléen indiquant que la reconnaissance s'est correctement passée. Chacune de ces fonctions peut aussi renvoyer (sous forme d'une référence en paramètre) le résultat de son calcul. Pour E :

```
bool analyseE(double & val);
```

Le code complet de cet analyseur se trouve dans le polycopié en annexe.

▶ Analyse ascendante

Le principe même de réduction lié aux analyseurs ascendants est totalement compatible avec les grammaires S-attribuées. En effet, la réduction d'une règle de type $A \rightarrow \gamma$ suppose que tous les composants de γ ont déjà été réduits.
On peut raisonner par induction pour prouver que tous les attributs des composants de γ sont alors connus et que l'on peut calculer l'attribut de A. En effet, les composants de γ sont soit :

— des non-terminaux qui ont été réduits de la même manière que A ;
— des terminaux dont les attributs sont par définition connus.

6.3 Avec bison

Comme nous allons le voir dans le chapitre suivant, bison intègre totalement le mécanisme d'attribution. Les actions associées aux réductions de règles permettent de disposer des attributs de la partie droite afin de calculer les attributs du symbole non-terminal réduit.

CHAPITRE 7

L'OUTIL BISON PLD

7.1 Syntaxe

Structure du fichier en entrée (extension .y) :

```
%{
Pré-déclarations C/C++
%}
Définitions, options
%%
Grammaire (règles et code associé)
%%
Post-code C/C++
```

▶ Détail

Signification de chaque partie

- Pré-déclarations. Il s'agit de code C/C++ qui sera recopié avant le code généré pour l'analyse. Si votre code utilise des bibliothèques spécifiques, c'est l'endroit où il faut les déclarer.
- Définitions et options. Il s'agit de directives propres à Bison qui permettent de paramétrer le fonctionnement de l'analyseur, de donner des priorités d'opérateurs, d'indiquer le type des jetons, *etc.*
- Grammaire. C'est la partie la plus importante, elle définit la grammaire du langage qui doit être reconnu par l'analyseur. Cette grammaire prend la forme de règles de production, éventuellement récursives, pour lesquelles des actions (code C/C++) sont associées.
- Post-code. À l'instar des pré-déclarations, ce code est recopié verbatim après le code de l'analyseur. Son utilisation est limitée.

▶ Grammaire

La définition de la grammaire se fait grâce à une liste de règles qui ont la syntaxe suivante :

```
non_terminal : sequence1 { /* C/C++ */ }
             | sequence2 { /* C/C++ */ }
             | ...
             ;
```

Les symboles sont représentés par des identificateurs

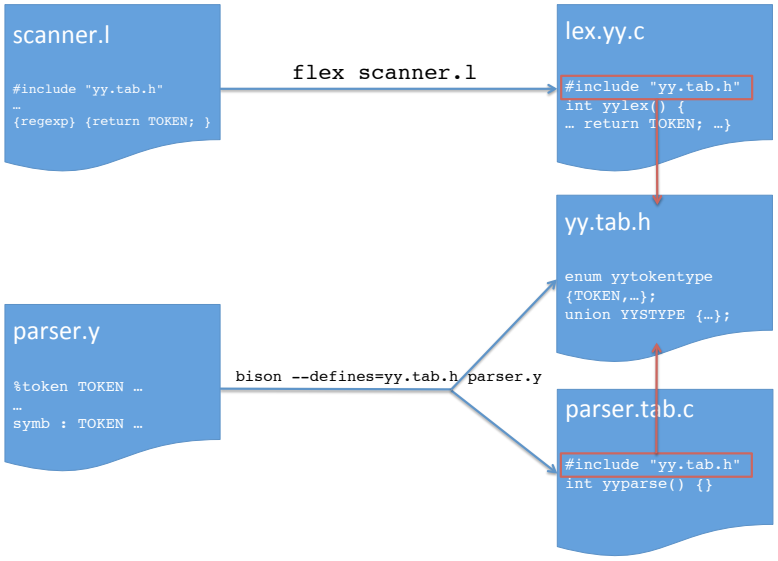


FIGURE 7.1 – Interactions entre flex et Bison

— Symboles terminaux (par convention, en majuscule sous Bison) : ce sont les jetons retournés par l'analyseur lexical
— Symboles non terminaux (par convention, en minuscule sous Bison) : ce sont des symboles qui sont internes (uniquement au niveau syntaxique) à la définition de la grammaire

Le code associé à chaque règle est optionnel. Ce code est exécuté **au moment de la réduction de la règle**.

▶ Communications Bison/flex

Les valeurs de jetons renvoyées par `yylex` doivent être connues de l'analyseur.
— Définition dans le fichier Bison de la directive `%token` indiquant la liste des jetons
— Au moment de l'invocation de `bison`, option `--defines=yy.tab.h`
— Inclusion dans le code généré par `flex` de ce fichier
— C'est Bison qui va associer une valeur à chaque jeton

La figure 7.1 page 76 résume le fonctionnement des interactions entre flex et Bison.

▶ Valeurs sémantiques

Que ce soit au niveau de `flex` ou `bison`, les symboles (terminaux ou non) peuvent avoir une valeur (un attribut) associée. Le type associé à cette valeur est directement paramétré dans les directives de `bison`.
— Pour indiquer la liste des types possibles : directive `%union`
— Pour les symboles terminaux : directive `%token`
— Pour les symboles non terminaux : directive `%type`
— La syntaxe :

```
%union {
   type1 nom1;
   type2 nom2;
   ...
}
%token TERMINAL1 TERMINAL2 ...
```

```
%token <nom1> TERMINAL3 TERMINAL4 ...
%token <nom2> TERMINAL5 TERMINAL6 ...
%type <nom1> nonterm1 nonterm2 ...
%type <nom2> nonterm3 nonterm4 ...
```

Dans une règle de Bison de la forme :

```
symb1 : symb2 symb3 TERM1
```

les valeurs sémantiques associées à chaque éléments seront (si les symboles ont effectivement des valeurs associées ce qui n'est pas obligatoire) :

- `symb1` → `$$` cette variable devra être affectée !
- `symb2` → `$1`
- `symb3` → `$2`
- `TERM1` → `$3`

Exemple de code :

```
expr : VALUE           { $$ = $1; }
     | expr PLUS expr  { $$ = $1+$3; }
     | expr MOINS expr { $$ = $1-$3; }
     ...
```

7.2 Mise au point

▶ Conflits

Au moment où vous allez essayer de générer votre analyseur, des erreurs peuvent être détectées : ce sont des conflits. Ils indiquent que le générateur LR possède dans sa table d'analyse des ambiguïtés.

- Conflit réduction/réduction. Deux règles mettent en jeu les mêmes jetons et à un stade de l'analyse, on ne saura pas quel règle il faut réduire si ce jeton est renvoyé. Ce conflit provient d'une erreur dans la grammaire, il faut le corriger.
- Conflit décalage/réduction. A un certain stade de l'analyse, l'arrivée d'un jeton ne permettra pas de dire s'il faut réduire une règle ou continuer à décaler pour reconnaître une autre règle (plus longue). Ce conflit est un peu moins visible que le précédent, car il implique une autre règle mais doit être supprimé lui aussi.
- **Dans tous les cas, ces conflits n'empêchent pas l'analyseur d'être généré, mais des choix seront faits arbitrairement.** (par exemple Bison fera prioritairement un décalage en cas de conflit avec une réduction)
- Certains conflits (dans des expressions notamment) peuvent être résolus par l'introduction de priorités d'opérateurs (directives `%left` et `%right`)
- Certains conflits sont dus à la présence d'une règle vide en trop

Exemple de conflit classique :

```
expr     : sequence
         | choix ;
sequence : sequence COMMA NAME
         | NAME ;
choix    : choix PIPE NAME
         | NAME ;
```

Quand on a un seul NAME, on ne sait pas s'il s'agit d'une séquence ou un choix. Ici, c'est donc la grammaire elle-même qui est ambigüe. Ce conflit peut être résolu en rajoutant une règle obligeant `choix` à avoir au minimum deux occurrences :

```
expr       : sequence
           | choix_plus ;
choix_plus : choix PIPE NAME ;
sequence   : sequence COMMA NAME
           | NAME ;
choix      : choix PIPE NAME
           | NAME ;
```

▶ Débugage

Plusieurs outils permettent de mettre au point et corriger les erreurs.

- Au moment de la génération de l'analyseur, on peut générer un fichier `.output` qui donne la liste des états et des transitions de l'automate LALR généré. Pour activer cette sortie, il faut indiquer l'option `-v` à `bison`
- Au moment de l'exécution de l'analyseur, on peut activer l'affichage des états, des jetons lus, des décalages et réductions effectués. Pour cela, il faut définir la variable `YYDEBUG` à 1 au moment de la compilation **et** mettre dans le code C/C++ `yydebug = 1;`

7.3 Interfaces avec l'extérieur

▶ Introduction

Par défaut, le code généré par flex et Bison repose sur des variables globales, nous allons voir ici comment personnaliser le comportement des outils afin d'en éviter toutes les limitations.

▶ Analyseurs multiples

Problème : si un outil doit utiliser plusieurs analyseurs lexicaux/syntaxiques générés par flex/Bison, se posent les problèmes suivants :

- Chaque analyseur est généré par flex/Bison
- Les identificateurs de chacun des analyseurs sont les mêmes
- Édition des liens impossible car on ne peut savoir quel analyseur est concerné

La solution consiste à utiliser l'option `-P` de `flex` et `-p` de `bison`

- Indique un préfixe pour remplacer `yy`
- Exemple avec `flex -P foo` :
  - `yylex()` → `foolex()`
  - `yyin` → `fooin`
  - *etc.*
- Exemple avec `bison -p bar` :
  - `yyparse()` → `barparse()`
  - `yydebug` → `bardebug`
  - *etc.*
- Il est évidemment recommandé d'utiliser le même préfixe pour la partie lexicale et syntaxique d'un même analyseur

▶ Fichiers multiples

Le principe est le suivant :

- Par défaut, `yylex` va chercher sur l'entrée standard et l'analyse prend fin à la fin du fichier
- Pour indiquer un autre fichier, il faut modifier `yyin` de type `FILE *`
- Exemple de code :

```
// extern FILE * yyin;
FILE * fid;
int err;
```

```
fid = fopen(nomfichier,"r");
if (!fid) {
   /* gestion d'erreurs */
}
yyin = fid;
err = yyparse();
fclose(fid);
/* exploitation du résultat */
```

La fonction `yywrap` renvoie un entier qui indique à l'analyseur lexical quoi faire en cas de fin de fichier lue :

- Si la valeur renvoyée est 1, alors la lecture s'arrête et la fin de fichier est renvoyée comme jeton (comportement par défaut)
- Si la valeur renvoyée est 0, alors la lecture va continuer, donc il faut repositionner `yyin` dans la fonction

▶ Récupération du résultat

La valeur renvoyée par l'axiome du langage est perdue si elle n'est pas utilisée dans le code associé. Pour remédier à cela, il faut utiliser la fonctionnalité qui permet de modifier le prototype de la fonction `yyparse` grâce à la directive `%parse-param { }`.
Par exemple si l'on indique :

```
%parse-param { Tree ** tree }
```

L'appel à la fonction `yyparse` se fera de la manière suivante :

```
int val;
Tree * a;
val = yyparse(&a);
```

Et on pourra inclure dans les actions associées aux règles des instructions du type :

```
exp: ....  { *tree = new Tree();
             *tree->SetRoot($1); }
```

⚠ Attention, dans cet exemple, il ne faudra faire qu'une seule fois l'instanciation de `*tree`, dans la règle associée à l'axiome par exemple.

On peut indiquer plusieurs paramètres à la fonction `yyparse` en faisant plusieurs appels à `%parse-param` :

```
%parse-param { int * res1 }
%parse-param { int * res2 }
```

Dans ce cas là, l'appel à `yyparse` sera :

```
int val;
int res1,res2;
val = yyparse(&res1,&res2);
```

⚠ Attention, les paramètres supplémentaires doivent être aussi passés dans la fonction `yyerror` en plus de la chaîne de caractères.

CHAPITRE 8
APPLICATIONS

8.1 Synthèse d'image

Les grammaires sont très présentes en synthèse d'image, sous des formes souvent dérivées. On va par exemple modifier directement le système de règles, ou alors la signification des symboles, ou introduire des éléments extérieurs afin de modifier les productions.

▶ Systèmes de réécriture

Les systèmes de réécriture diffèrent légèrement de ce que l'on a pu étudier jusqu'à présent avec les grammaires formelles :

- Pas de distinction entre symbole non-terminal et terminal
- On peut appliquer plusieurs fois les règles de production sur un même mot
- Plusieurs règles peuvent être applicables avec des probabilités différentes

À cela on va ajouter une interprétation graphique des symboles, et on obtient un L-system.

Exemple, génération d'une ile de Koch avec seulement les deux règles suivantes :

$$\begin{array}{ll} \omega : & F - F - F - F \\ p : & F \rightarrow F - F + F + FF - F - F + F \end{array}$$

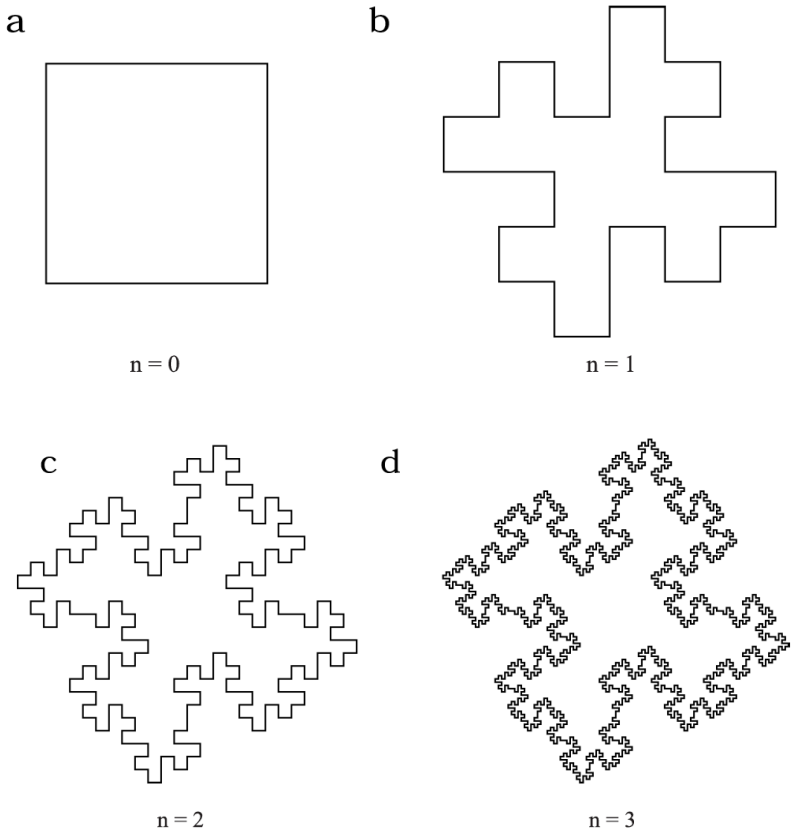
La première règle est là uniquement pour indiquer une chaîne de départ, la deuxième se contente de réécrire tous les F en une liste de symboles dont nous allons expliquer la signification graphique (style tortue) :

- F : avancer d'une distance d
- $-$: tourner à droite d'un angle δ
- $+$: tourner à gauche d'un angle δ

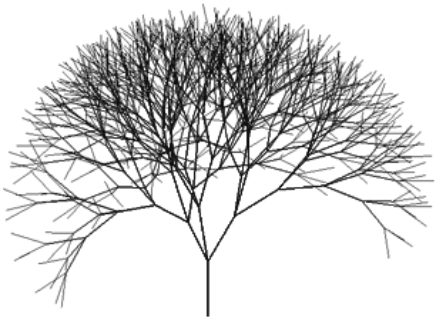
Entre chaque réécriture, on va prendre soin de diviser la distance d par 4 afin d'avoir une figure dont la taille est toujours la même.

Si on choisit $\delta = 90°$, on obtient les itérations suivantes [1] :

1. Illustrations tirées de *The Algorithmic Beauty of Plants*, Springer Verlag, 1990



Les L-systems sont très connus pour leur facilité à générer des structures de type croissances d'arbres [2] :



▶ Shape grammar

Le principe des *shape grammars* est dérivé de celui des L-systems : il s'agit d'un mécanisme de réécriture comme pour les L-systems, mais les symboles utilisés sont directement des formes géométriques.
Dans une version de base, les règles des *shape grammar* s'écrivent donc de manière graphique, les opérations que l'on peut appliquer sont de nature diverses :

— Ensemblistes : addition (union), intersection, soustraction
— Transformations géométriques : translation, rotation, réflexion, homothétie, *etc.*

En quoi, les *shape grammar* sont différentes de grammaires habituelles ?

2. Illustration `http://www.sidefx.com/docs/houdini10.0/nodes/sop/lsystem`

À tout moment, il faut analyser les formes produites pour voir si de nouvelles formes ont été introduites. Par exemple, l'union de deux carrés translatés peut produire un autre carré de taille plus petite.

Cela suppose que l'on a à disposition un moteur graphique puissant capable de reconnaître ces formes. En pratique, les *shape grammar* sont assez difficiles à exploiter.

▶ Génération procédurale de bâtiments

L'idée des CGA-shape est de combiner divers concepts :

- La réécriture des L-systems (mais en gardant le principe de terminaux et non-terminaux)
- La notion d'attributs (dans le cas des CGA-shape, il s'agit d'attributs géométriques)
- Les primitives géométriques des *shape grammars*
- Des opérations de répétition, subdivision, proches des *shape grammars*

On obtient un système de règles complexes permettant de modéliser des bâtiments. Voici un exemple de résultats [3] :

L'inconvénient de ces méthodes est le manque d'intelligibilité des règles qui permettent de générer les bâtiments. Des interfaces ont par la suite été faites pour rendre l'édition plus interactive donnant lieu à la création d'un logiciel nommé CityEngine [4].

▶ Open L-systems

Les L-systems sont très pratiques pour modéliser des plantes, mais on souffre souvent d'un manque de contrôle sur le résultat. L'idée des open L-systems est d'introduire la notion d'environnement. Les règles de réécriture sont guidées par un oracle extérieur qui donne une indication sur l'environnement.

On peut par exemple contraindre un arbre à ne pousser que dans une certaine zone ou à tenir compte d'autres arbres qui poussent en même temps (notion de partage de ressources).

Exemple de résultat [5] :

3. tiré de *Procedural Modeling of Buildings*, Müller et al, ACM Transactions on Graphics, 25(3), 2006.
4. `http://www.esri.com/software/cityengine/`
5. tiré de *Self-organizing tree models for image synthesis*, Palubick et al, ACM Transactions on Graphics 28(3), 2009.

8.2 Compilation

Nous allons dans cette section montrer comment les éléments que nous avons étudiés peuvent servir dans un contexte de compilation d'un langage source.

▶ Principe

Voici les différentes étapes d'un compilateur (partie dite frontale) :
- Analyse lexicale
- Analyse syntaxique
- Génération d'une représentation intermédiaire (machine abstraite à pile par exemple)

La partie de compilation dite finale dépend de la machine cible et va permettre de générer le code assembleur en effectuant au passage des optimisations.

▶ Table des symboles

L'analyseur lexical, à chaque fois qu'il va retourner un jeton de type identificateur, va aller ajouter une entrée dans la table des symboles.
L'analyseur syntaxique va venir compléter cette entrée avec par exemple un type associé à l'identificateur lorsqu'il va rencontrer une structure de déclaration.
Plus tard dans l'analyse, on a des informations sur le type de l'identificateur quand on le rencontre à nouveau.

▶ Gestion des erreurs

Suivant le type d'analyseur choisi, la production de messages d'erreurs sera plus ou moins facilitée. Dans le cas d'un analyseur ascendant, l'affichage des erreurs est plus naturel.

▶ Compression de tables

La grammaire d'un langage de programmation standard comporte plusieurs centaines de règles (cela peut varier suivant quel type d'analyseur on utilise). L'automate d'analyse généré peut comprendre facilement plusieurs milliers voir dizaines de milliers d'états. Le codage de la table d'analyse devient crucial en termes de place mémoire mais aussi de temps de calcul d'une transition.
Il existe diverses techniques pour compresser les tables d'analyse :
- Ne représenter les lignes identiques qu'une seule fois et y faire référence
- Ne représenter que les transitions utiles : une liste plutôt qu'un tableau à deux dimensions

8.3 Langue naturelle

On s'aperçoit qu'une grammaire hors-contexte est difficilement utilisable pour décrire la grammaire d'une langue naturelle. On utilise souvent le concept de grammaire affixe pour cela.

▶ Grammaire affixe

Il existe plusieurs types de grammaires affixes, mais elles ont toutes le point commun de pouvoir ajouter derrière un symbole une information. Par exemple, pour le sujet d'une phrase, on peut ajouter la notion de nombre (au sens grammatical) indiquant le singulier ou le pluriel. Cela va permettre de faire l'accord entre le sujet et le verbe sans avoir à décrire toutes les combinaisons possibles. Exemple :

$$
\begin{array}{lcl}
Phrase & \rightarrow & Sujet + nombre \\
 & & Predicat + nombre \\
Sujet + nombre & \rightarrow & Nom + nombre \\
Predicat + nombre & \rightarrow & Verbe + nombre\ Objet \\
Objet & \rightarrow & Nom + nombre
\end{array}
$$

$$
\begin{array}{lcl}
Nom + singulier & \rightarrow & Jean \\
Nom + singulier & \rightarrow & Marie \\
Nom + pluriel & \rightarrow & Les\ enfants \\
Nom + pluriel & \rightarrow & Les\ parents
\end{array}
$$

$$
\begin{array}{lcl}
Verbe + singulier & \rightarrow & aime \\
Verbe + pluriel & \rightarrow & aiment \\
Verbe + singulier & \rightarrow & aide \\
Verbe + pluriel & \rightarrow & aident
\end{array}
$$

Ce type de grammaire n'apporte pas d'expressivité par rapport aux grammaires hors-contexte, il permet juste de gagner en concision.

▶ Ambigüité

D'une manière générale, l'analyse de la langue naturelle est très difficile de par l'ambigüité intrinsèque de celle-ci.
Les ambigüités peuvent intervenir à plusieurs niveaux :

- Homonymie : c'est l'ambigüité dès le niveau lexical, un mot peut avoir plusieurs sens qui n'ont pas le même rôle dans la phrase (ici des homographies)
  - *as* : le verbe avoir ou la carte ?
  - *est* : le point cardinal ou le verbe être ?
  - *bois* : le verbe boire ou la forêt ?
  - *avocat* : le fruit ou le professionnel ?
  - *mousse* : verbe mousser, nom masculin, féminin ?
  - *etc.*
- Syntaxique : plusieurs constructions syntaxiques peuvent mener à la même phrase mais engendrent des sens différents
  - *Elle emmène les clefs de la maison au garage* : ce sont les clefs de la maison ou non ?
  - *La petite brise la glace* : où est le verbe ?
  - *C'est la fille du cousin qui boit* : qui boit, la fille ou le cousin ?
  - *etc.*

Comment procédons-nous lorsque nous lisons ou écoutons quelqu'un parler ? La levée d'ambigüité lorsqu'elle est possible relève de plusieurs facteurs :

- La globalité de la phrase aide à lever l'ambigüité de type homonymie (ou homophonie pour le langage parlé). Exemple : *le ciel est bleu*, il s'agit du verbe être et non du point cardinal

— Le contexte. Les phrases précédentes (voire suivante) aident à replacer la phrase dans son contexte. Exemple :
  — *La petite brise la glace. Le froid l'envahit bientôt de la tête aux pieds.*
  — *Elle n'en peut plus de ce silence. La petite brise la glace.*
— La probabilité. Certains sens possibles de la phrase sont peu probables. Exemple : *Il a parlé de déjeuner avec Paul*. Il est peu probable de parler du sujet *déjeuner* ensemble, par contre ils ont certainement planifié de manger ensemble.

▶ Conclusion

L'analyse automatique de la langue naturelle est extrêmement difficile de par la nature même de celle-ci. Cette analyse automatique rend la traduction automatique extrêmement difficile elle aussi.

CHAPITRE 9
EXEMPLE COMPLET PLD

9.1 Introduction

Nous allons dans ce chapitre montrer sur un exemple assez simple la conception d'une application mettant en œuvre une programmation C++. L'accent ne sera pas mis sur la conception de la grammaire mais plus sur la conception d'ensemble et la cohésion globale.

9.2 Cahier des charges rapide

Le but de l'application exemple est de construire un interpréteur d'expressions arithmétiques littérales. L'interpréteur devra traiter les opérateurs suivants :

- Addition, soustraction
- Multiplication, division
- Les parenthèses permettent de grouper des éléments pour modifier localement la priorité des opérateurs

Les opérandes sont des valeurs numériques ou des variables littérales. Une fonction d'évaluation permet de calculer la valeur de l'expression en associant aux variables littérales une valeur.

9.3 Conception de la grammaire

▶ Syntaxique

Commençons par nous intéresser au côté syntaxique en considérant que l'analyseur lexical fournit les terminaux suivants : nombre, variable, et tous les opérateurs. On peut former une grammaire qui ne contient qu'un seul non-terminal E :

$$
\begin{aligned}
E &\rightarrow nombre \\
E &\rightarrow variable \\
E &\rightarrow E \ + \ E \\
E &\rightarrow E \ - \ E \\
E &\rightarrow E \ * \ E \\
E &\rightarrow E \ - \ E \\
E &\rightarrow (\ E\ )
\end{aligned}
$$

Cette grammaire est récursive à gauche, donc serait inutilisable dans un analyseur descendant. De la même manière, elle est ambiguë et ne prend donc pas en compte les priorités d'opérateurs. Ces

deux problèmes seront résolus par la priorité des opérateurs ainsi que leur associativité à gauche. En revanche, la grammaire ne comporte aucun conflit décalage/réduction ni réduction/réduction et est utilisable dans un contexte d'analyse ascendante.

▶ Lexical

La conception du niveau lexical impose de fournir pour chacun des symboles terminaux une expression régulière associée. Voici le résultat de ce travail :

— Nombre entier : `[0-9]+`
— Nombre flottant : `[0-9]+\.[0-9]*`
— Variable : `[a-zA-Z]+`
— Symboles composés d'un seul caractère : `+ - / * ( )`

▶ Valeur sémantique

Pour chacun des symboles terminaux, il faut déterminer le type associé.

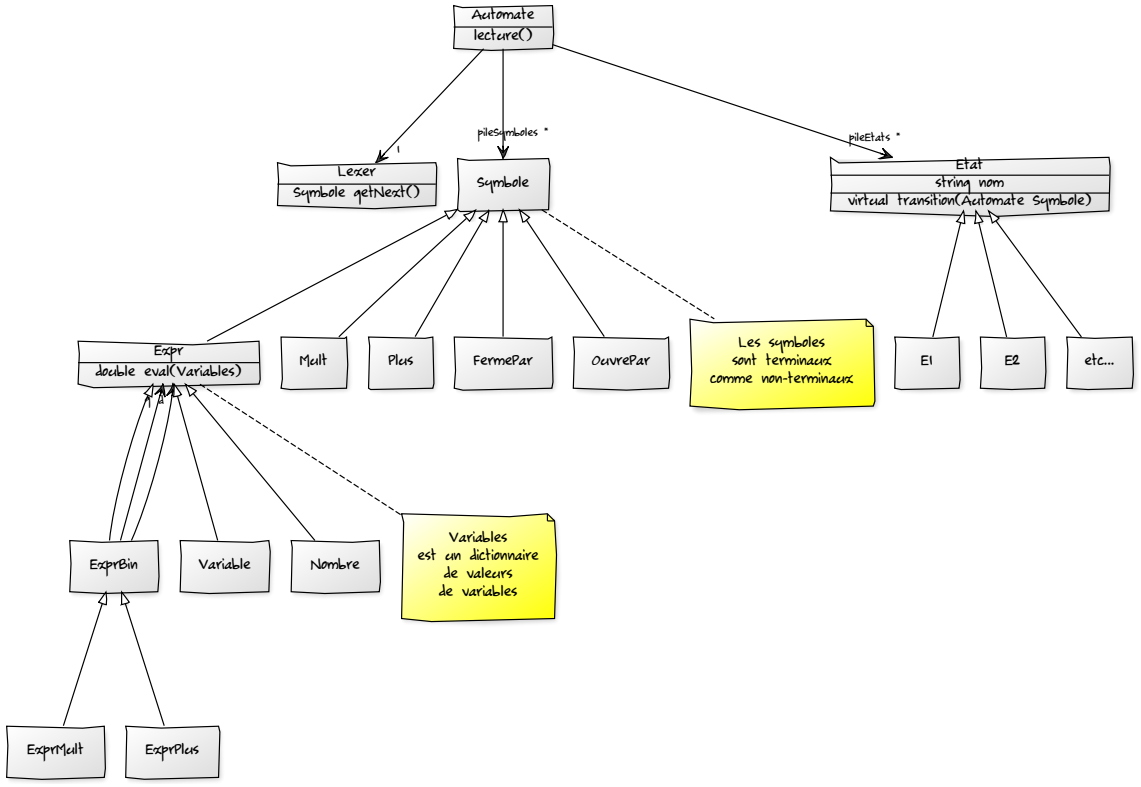
— Nombre : `double`
— Variable : `string`
— Symboles composés d'un seul caractère : pas de valeur associée (opérateurs et parenthèses)

9.4 Conception

On se propose ici de faire la conception du modèle de données qui va nous servir au stockage et à l'évaluation d'une expression telle que décrite dans le cahier des charges.

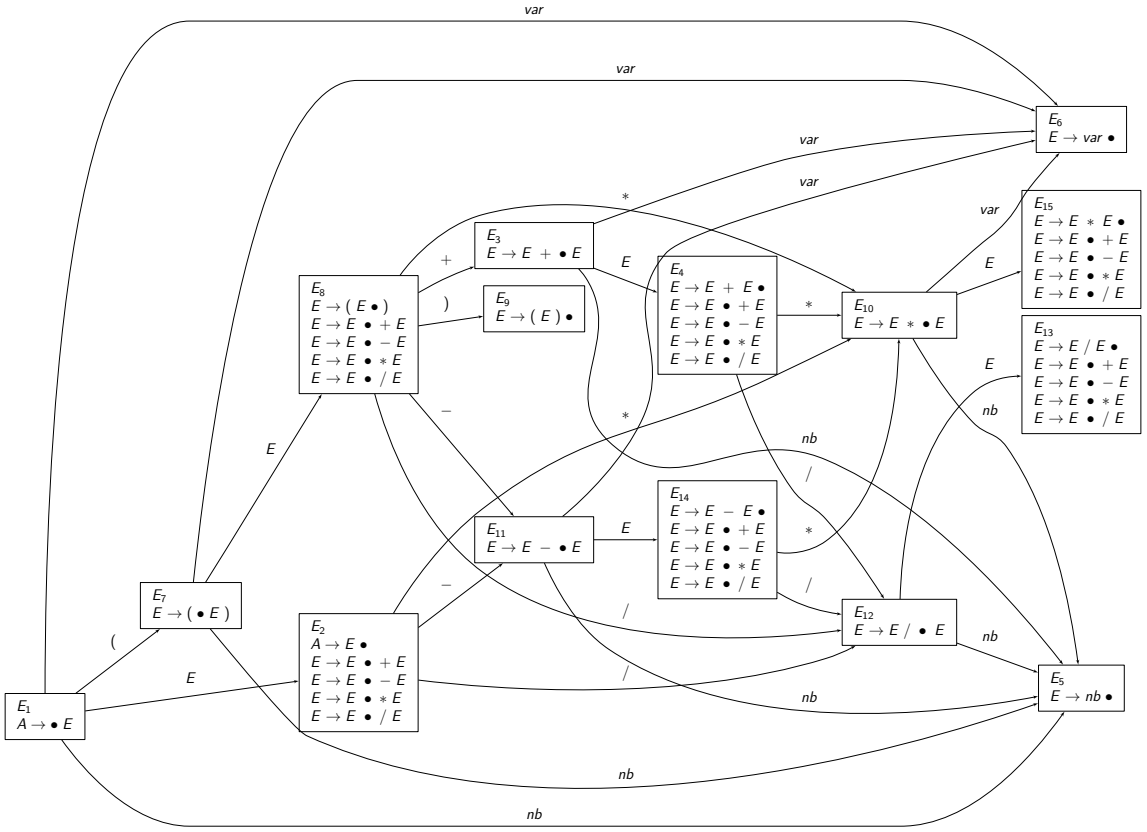
Nous allons utiliser la modélisation grâce à une hiérarchie de classes. La méthode d'évaluation sera polymorphe afin de rendre l'application la plus évolutive possible. Cette hiérarchie est incluse dans la hiérarchie des symboles associée à l'automate. Nous reprenons dans cet exemple le design pattern State pour représenter l'automate.

Voici le diagramme de classe proposé :



9.5 Automate LR

À partir de la grammaire, il faut construire l'automate d'analyse ascendante. Nous allons construire un automate LR(0) que nous allons de manière intuitive transformer en automate LR(1) grâce à des règles de priorité d'opérateurs.



Sur cet automate ne sont pas représentés les items au complet, seulement les noyaux. Les conflits sont dans les états E_4, E_{13}, E_{14} et E_{15}. En effet, on ne sait pas s'il faut réduire la règle ou non. Prenons le cas de figure de l'état E_4, on a reconnu $E+E$ mais suivant ce qui va se trouver derrière, il se peut qu'il ne faille pas le réduire. Si la phrase à reconnaître était $1+2+3$, réduire consistera à dire que l'on considère $1+2$ comme un tout et que ce tout sera ensuite combiné à la valeur 3, cela revient donc à $(1+2)+3$. Dans cet exemple, si nous avions décidé de ne pas réduire cela aurait signifié $1+(2+3)$ ce n'aurait rien changé au résultat mais l'arbre syntaxique aurait été différent. Retenons simplement que lorsque ce sont des opérateurs identiques, la réduction correspond à une associativité à gauche alors que la non-réduction à une associativité à droite. Si la phrase à reconnaître était $1+2*3$, réduire consistera à dire que l'on considère le groupement $(1+2)*3$ ce qui n'est pas conforme aux règles de priorités d'opérateurs arithmétiques communément considérées. Si le symbole suivant est un $*$ il est donc impératif de ne pas réduire la règle ! Il en est de même pour la division. Cela est dû aux priorités d'opérateurs. Nous pouvons donc déduire que pour les états E_4 et E_{14}, il faut réduire seulement si le prochain symbole n'est pas un $*$ ou un $/$. Pour les états E_{13} et E_{15} nous pouvons réduire dans tous les cas de figure.

9.6 Réalisation - tout C++

Nous allons voir dans cette section quelques éléments de la réalisation de l'analyseur en C++ sans recourir à un outil de génération comme bison. Dans le TP associé au module Grammaires et Langages, vous aurez à implémenter cet automate dans une séance de 4h.

▶ Classes C++

Tout d'abord la classe abstraite `Symbole` :

```
class Symbole {
```

```
   protected:
      int ident;
   public:
      Symbole(int id) : ident(id) {}
      virtual ~Symbole() {}
      void print();
      operator int() const { return ident; }
};
```

La surcharge de l'opérateur `int()` permettra de caster implicitement un symbole en entier, pour connaître son identifiant. Ce sera pratique dans l'implémentation des méthodes de transition.
La classe abstraite `Expr` :

```
class Expr : public Symbole {
   public:
      Expr():Symbole(EXPR) {}
      virtual ~Expr() {}
      virtual double eval(const map<string, double>
                                                & valeurs) = 0;
};
```

On a choisi de coder la liste des valeurs de variables par une `map` de la STL.
La classe abstraite `Etat` :

```
class Etat {
public:
  Etat(string name);
  virtual ~Etat();
  void print() const;
  virtual bool transition(Automate & automate,
                          Symbole * s) = 0;
protected:
  string name;
};
```

exemple d'une fonction de transition :

```
bool E1::transition(Automate & automate,
                    Symbole * s) {
  switch (*s){
  case NOMBRE:
  case VARIABLE:
  case EXPR:
    automate.decalage(s, new E2);
    break;
  case OUVREPAR:
    automate.decalage(s, new E7);
    break;
  }
  return false;
}
```

Les états E_5 et E_6 sont court-circuités dans cette implémentation et font directement l'objet d'un décalage.
La fonction de décalage est assez simple, elle se contente d'empiler un symbole et un état donnés en argument :

```
void Automate::decalage(Symbole * s, Etat * e) {
  symbolstack.push_back(s);
  statestack.push_back(e);
}
```

Les fonctions de réduction sont plus complexes car elles ont du travail spécifique à effectuer pour construire un symbole. Elles doivent d'abord dépiler un certain nombre de symboles, les utiliser pour en construire un nouveau et appeler la fonction de transition du nouveau symbole créé. On peut factoriser la fin de la réduction dans une fonction de ce type :

```
void Automate::reduction(int n,Symbole * s) {
  for (int i=0;i<n;i++)
```

```
  {
    delete(statestack.back());
    statestack.pop_back();
  }
  lexer->putSymbol(s);
}
```

La dernière ligne sert à imiter le comportement d'un symbole sur la tête de lecture. C'est un moyen détourné de faire la prochaine opération de lecture sur ce symbole créé plutôt que d'appeler directement dans la réduction la transition associée.
Exemple d'appel de la fonction de réduction pour la règle $E \rightarrow E + E$:

```
bool E4::transition(Automate & automate, Symbole * s) {
  switch (*s) {
  case MULT:
    automate.decalage(s, new E10);
    break;
  case DIV:
    automate.decalage(s, new E12);
    break;
  default:
    automate.putSymbol(s); // LR(1), on ne consomme pas ce symbole
    Expr * s1 = (Expr*) automate.popSymbol();
    automate.popAndDestroySymbol();
    Expr * s2 = (Expr*) automate.popSymbol();
    automate.reduction(3,new ExprPlus(s2,s1));
    break;
  }
  return false;
}
```

9.7 Réalisation - Bison/flex PLD

Nous allons voir dans cette section comment implémenter la même application avec les outils flex et Bison. Une partie de l'application est semblable (pour l'implémentation des classes correspondant à l'expression arithmétique notamment). Nous allons juste voir les fichiers flex et bison nécessaires au fonctionnement de l'application.
Flex

```
%{
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "expr.h"
#include "expr.tab.h"
%}
%option noyywrap
digit    [0-9]
ident    [a-zA-Z]+
%%
{digit}+"."{digit}* { yylval.d = atof(yytext);
                      return DOUBLE; }
{digit}+            { yylval.i = atoi(yytext);
                      return INT; }
{ident}             { yylval.s = strdup(yytext);
                      return VAR; }
"+"                 { return PLUS; }
"-"                 { return MOINS; }
"/"                 { return SLASH; }
"*"                 { return MULT; }
"^"                 { return EXP; }
"("                 { return OPEN; }
")"                 { return CLOSE; }
.                   { printf("Erreur : %c\n",*yytext); }
%%
```

Bison

```
%{
#include "expr.h"
#include <map>
#include <string>
#include <iostream>

extern int yylex(void);
void yyerror(Expression ** e, const char * msg);

%}

%union {
   char * s;
   double d;
   int i;
   Expression * e;
}
%token PLUS MOINS MULT SLASH EXP OPEN CLOSE
%token <s> VAR
%token <d> DOUBLE
%token <i> INT
%type <e> expr

%left MOINS PLUS
%left MULT SLASH
%right EXP

%parse-param {Expression ** e}

%%
main : expr             { *e = $1; }
     ;

expr: DOUBLE            { $$ = new Valeur($1); }
    | INT               { $$ = new Valeur($1); }
    | VAR               { $$ = new Variable($1);
                          free($1); }
    | expr PLUS expr    { $$ = new OperateurPlus($1,$3); }
    | expr MOINS expr   { $$ = new OperateurMoins($1,$3); }
    | expr SLASH expr   { $$ = new OperateurDiv($1,$3); }
    | expr MULT expr    { $$ = new OperateurMul($1,$3); }
    | expr EXP expr     { $$ = new OperateurExp($1,$3); }
    | OPEN expr CLOSE   { $$ = $2; }
    ;
%%
```

Programme principal

```
void yyerror(Expression ** e, const char * msg) {
   cout<<msg;
}

int main(void) {
   map<string,double> var;
   Expression * e=0;
   //yydebug = 1;
   yyparse(&e);
   if (e) {
      var["x"] = 1.0;
      var["y"] = 2.0;
      cout<<"Valeur : "<<e->Evaluation(var)<<endl;
      delete e;
   }
   return 0;
}
```

Comment construire l'executable ?

```
flex expr.l
bison -v -d expr.y
g++ -DYYDEBUG -g -o expr expr.cpp lex.yy.c expr.tab.c
```

CHAPITRE 10

CORRECTIONS DES EXERCICES

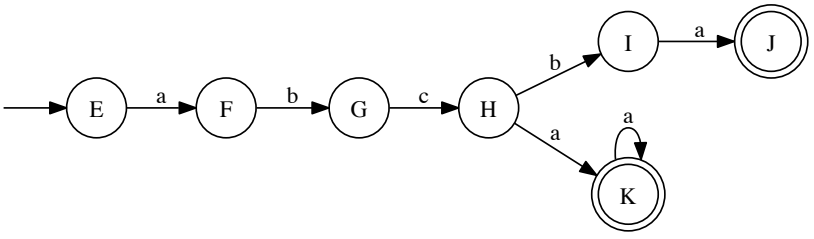
Analyse lexicale

▶ Dérivées

Correction 1

- $D_a(E) = bc(a^*|b)a = F$
- $D_b(F) = c(a^*|b)a = G$
- $D_c(G) = (a^*|b)a = H$
- $D_b(H) = a = I$
- $D_a(I) = \epsilon = J$
- $D_a(H) = a^* = K$
- $D_a(K) = a^* = K$

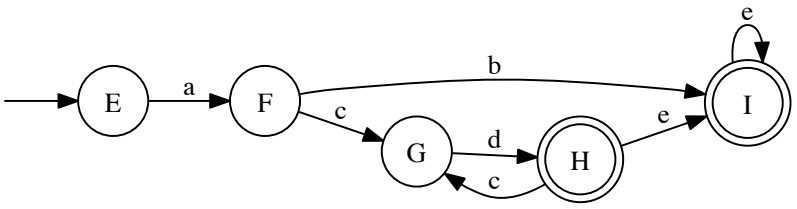
Correction 2



Correction 3

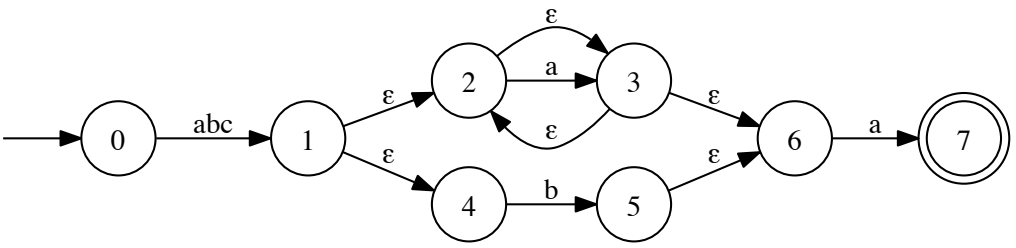
- $D_a(E) = (b|(cd)^*)e^* = F$
- $D_b(F) = e^* = I$
- $D_c(F) = d(cd)^*e^* = G$
- $D_d(G) = (cd)^*e^* = H$
- $D_c(H) = d(cd)^*e^* = G$
- $D_e(H) = e^* = I$
- $D_e(I) = e^* = I$

Correction 4

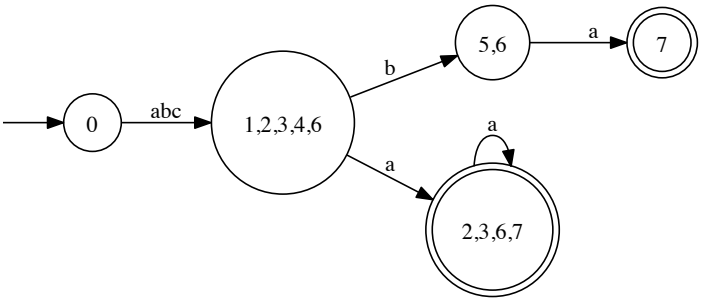


▶ Construction d'automate

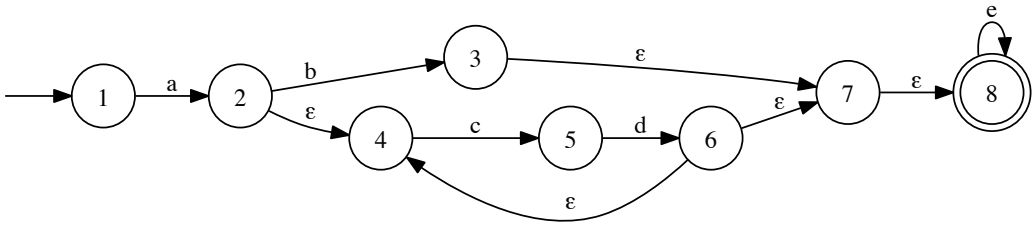
Correction 5 Construction de Thompson



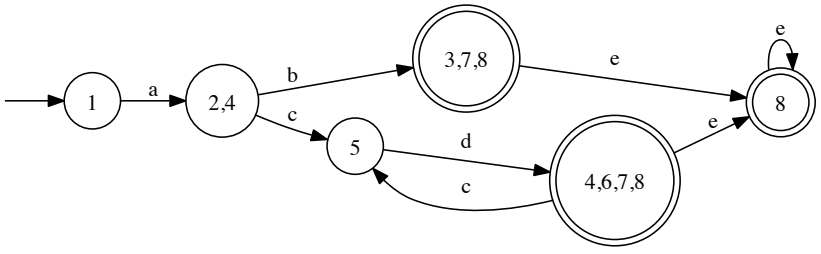
Correction 6 Automate déterministe



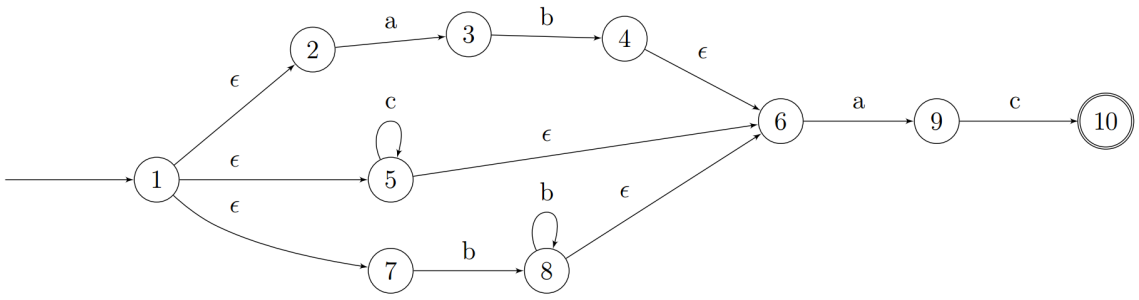
Correction 7 Construction de Thompson



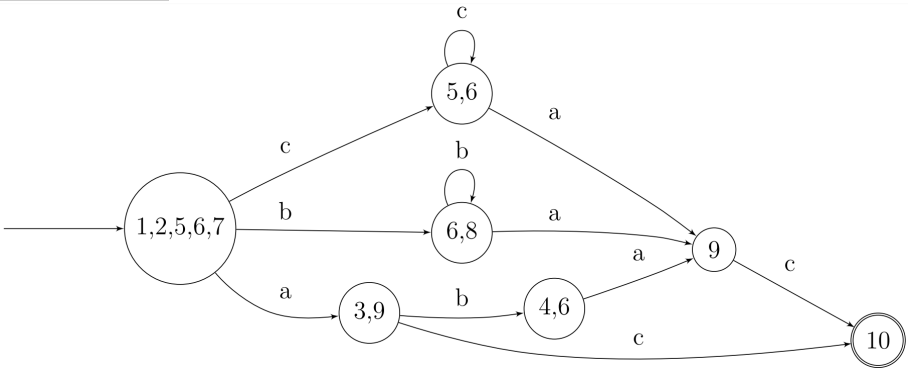
Correction 8 Automate déterministe



Correction 9 Construction de Thompson



Correction 10 Automate déterministe



▶ Expressions régulières

Correction 11 $(ab*d|ba*c)a*b$
Correction 12 $a(ba)*c$
Correction 13 $(a|b)(a(a|b))*cd*$
Correction 14 $a(b|c)(e(a(b|c)))*(c|d)$
Correction 15 $ac*(bab*a|bda)a*$ qui donne $ac*b(ab*|d)a+$
Correction 16 $abb+$
Correction 17 $(a*|a+b|b*a)$

Analyse descendante

▶ Grammaire de grammaire

Correction 18

$$\mathcal{N} = \{G, L\}$$
$$\mathcal{P}(G) = \{\ n, \epsilon\ \}$$
$$\mathcal{P}(R) = \{\ n\ \}$$
$$\mathcal{P}(L) = \{\ n, t, \epsilon\ \}$$
$$\mathcal{S}(G) = \{\ \$\ \}$$
$$\mathcal{S}(R) = \{\ p\ \}$$
$$\mathcal{S}(L) = \{\ p\ \}$$

Correction 19

	$\$$	p	n	t	f
G	ϵ		RpG		
R			nfL		
L		ϵ	nL	tL	

▶ Pseudo XML

Correction 20

$$\begin{aligned}
&\mathcal{N} = \{C\}\\
&\mathcal{P}(E) = \{\text{ open }\}\\
&\mathcal{P}(C) = \{\text{ open, data, } \epsilon\ \}\\
&\mathcal{P}(I) = \{\text{ open, data }\}\\
&\mathcal{S}(E) = \{\text{ \$, open, data, close }\}\\
&\mathcal{S}(C) = \{\text{ close }\}\\
&\mathcal{S}(I) = \{\text{ open, data, close }\}
\end{aligned}$$

Correction 21

	open	data	close	$
E	open C close			
C	$I\ C$	$I\ C$	ϵ	
I	E	data		

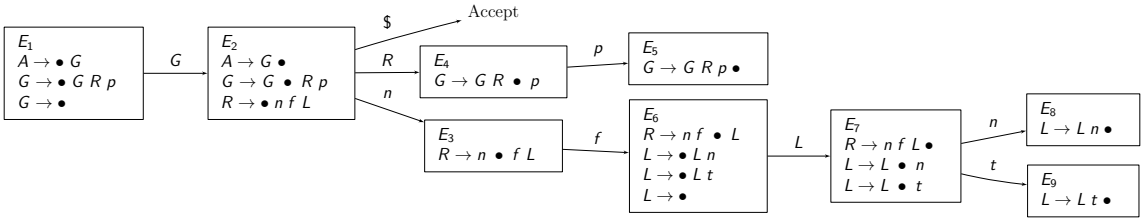
Analyse ascendante

▶ Grammaire de grammaire

Correction 22 Grammaire augmentée, on ajoute un axiome A et une règle associée :

$$\begin{aligned}
A &\rightarrow G\\
G &\rightarrow G\ R\ p\\
G &\rightarrow \epsilon\\
R &\rightarrow n\ f\ L\\
L &\rightarrow L\ n\\
L &\rightarrow L\ t\\
L &\rightarrow \epsilon
\end{aligned}$$

Correction 23 Automate LR(0) :

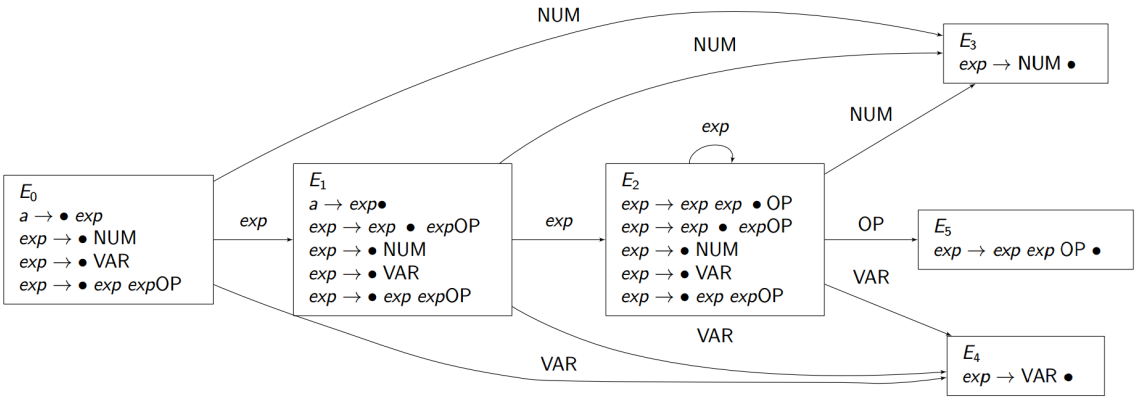


Correction 24 Etats en conflit : E_1, E_2, E_6 et E_7.
Correction 25 $\mathcal{S}(G) = \{n, \$\}$. $\mathcal{S}(R) = \{p\}$. $\mathcal{S}(L) = \{n, t, p\}$.
Correction 26 La règle R_0 est celle de l'axiome, les autres sont numérotées comme dans l'ordre du sujet.

	$	p	n	t	f	G	R	L
E_1	R_2		R_2			E_2		
E_2	R_0(A)		E_3				E_4	
E_3					E_6			
E_4		E_5						
E_5	R_1		R_1					
E_6		R_6	R_6	R_6				E_7
E_7		R_3	E_8	E_9				
E_8		R_4	R_4	R_4				
E_9		R_5	R_5	R_5				

▶ Expression postfixée

Correction 27 Automate LR(0) :



Correction 28 Le seul conflit est dans l'état 1, on ne sait pas s'il faut réduire ou non le premier item. Il suffit de voir que le symbole a ne peut être suivi que de la fin de flux, pour lever le conflit.

CHAPITRE 11
BIBLIOGRAPHIE

11.1 Ouvrages de référence

— *Compilers : Principles, Techniques and Tools*, Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Addison-Wesley, 1988. [1]
— *Compilateurs, Principes techniques et outils*, Traduction française de l'ouvrage précédent. InterÉditions, Paris, 1989.
— *Parsing Techniques, A Practical Guide*, Dick Grune, Ceriel Jacobs. Ellis Horwood Limited, 1990. La première édition de ce livre est en ligne `http://dickgrune.com/Books/PTAPG_1st_Edition/`. La deuxième édition date de 2008 et est éditée par Springer.

11.2 Cours en ligne

— Cours de compilation, L3S5, Université Lille 1, Mirabelle Nebut, `http://www.fil.univ-lille1.fr/portail/index.php?dipl=L&sem=S5&ue=COMPIL`
— Flex/bison, cours à l'ENIB, Fabrice Harrouet, `http://www.enib.fr/~harrouet/courses.html`
— Cours de compilation, ENS, M1, Tanguy Risset, `http://perso.citi.insa-lyon.fr/trisset/cours/compil2004.html`
— Théorie des langages, notes de cours, François Yvon et Akim Demaille, `http://www.lrde.epita.fr/~akim/thl/theorie-des-langages-1.pdf`
— Diverses ressources liées au *Dragon Book* se trouvent ici : `http://dragonbook.stanford.edu/`

11.3 Manuels d'utilisation

— Manuel de flex, `http://flex.sourceforge.net/manual/`
— Manuel de bison, `http://www.gnu.org/software/bison/manual/`

1. Ce livre est une véritable bible connue sous le nom de *Dragon book*

CHAPITRE 12
ANNEXES

⚠ Attention, les codes présents dans cette annexe ne respectent pas (volontairement, pour plus de concision) le guide de style C++ du département

12.1 Automate à pile pour l'analyse LL(1)

Listing complet d'un analyseur LL avec automate à pile d'une grammaire d'expressions.

```
#include <iostream>
#include <deque>
#include <map>
using namespace std;

typedef enum {E_, Ep_, T_, Tp_, F_, nb_, id_,
   plus_, moins_, mult_, div_, open_, close_,
   end_} Token;

const string TokenLabels[] =
    {"E","E'","T","T'","F","nb","id",
     "+","-","*","/","(",")","$"};

typedef deque<Token> Pile;

Pile mkPile(Token t1, Token t2, Token t3) {
   Pile p;
   p.push_front(t3);
   p.push_front(t2);
   p.push_front(t1);
   return p;
}

Pile mkPile(Token t1, Token t2) {
   Pile p;
   p.push_front(t2);
   p.push_front(t1);
   return p;
}

Pile mkPile(Token t) {
   Pile p;
   p.push_front(t);
   return p;
}

struct Etat {
   Pile pile;
```

```
   Pile alire;
};


ostream  & operator<< (ostream & os,
                       const Pile & p) {
   Pile::const_iterator i;
   for (i=p.begin();i!=p.end();i++) {
      os<<TokenLabels[*i]<<" ";
   }
   return os;
}

void AffichePile(Pile p) {
   Pile::iterator i;
   for (i=p.begin();i!=p.end();i++) {
      cout<<TokenLabels[*i]<<" ";
   }
}

typedef map<Token,map<Token,Pile> > Transitions;

bool Existe(Transitions & t, Token i, Token j) {
   return (t.find(i)!=t.end()) and
          (t[i].find(j)!=t[i].end());
}

bool LL1AAP(const Pile & mot,
            Transitions & transitions,
            Token axiome)
{
  Etat etat;
  etat.pile.push_front(axiome);
  etat.alire = mot;
  Token a,b;
  if (etat.alire.back() != end_) {
     etat.alire.push_back(end_);
  }

  while (!etat.pile.empty()) {
    a = etat.pile.front();
    b = etat.alire.front();
    if (a==b) {
      cout<<"Lecture de "<<a<<endl;
      etat.pile.pop_front();
      etat.alire.pop_front();
    } else if (!Existe(transitions,a,b)) {
      cout<<"Transition non trouvée"<<endl;
      // si on veut juste rejeter le symbole
      // on peut faire ça
      // etat.pile.pop_front();
      return false;
    } else {
      etat.pile.pop_front();
      cout<<"Transition"<<endl;

      Pile::const_reverse_iterator i;
      for (i=transitions[a][b].rbegin();
           i!=transitions[a][b].rend();i++) {
        etat.pile.push_front(*i);
      }
    }
    cout<<"Pile : "<<etat.pile<<endl;
    cout<<"A lire : "<<etat.alire<<endl;
  }
  return true;
}
```

```
int main(void) {

   Transitions trans;
   Pile epsilon;
   #define Trans(x,y,z) trans[x][y]=mkPile z
   #define ETrans(x,y) trans[x][y]=epsilon
   #include "transitions.h"
   #undef Trans
   Pile mot;
   mot.push_back(id_);
   mot.push_back(mult_);
   mot.push_back(nb_);
   mot.push_back(plus_);
   mot.push_back(id_);
   //mot.push_back(open_);
   if (LL1AAP(mot,trans,E_)) {
      cout<<"Mot reconnu"<<endl;
   }
   else {
      cout<<"Mot non reconnu"<<endl;
   }

   return 0;
}
```

Fichier contenant la table des transitions :

```
Trans(E_,open_,(T_,Ep_));
Trans(E_,nb_,(T_,Ep_));
Trans(E_,id_,(T_,Ep_));
Trans(Ep_,plus_,(plus_,T_,Ep_));
Trans(Ep_,moins_,(moins_,T_,Ep_));
ETrans(Ep_,close_);
ETrans(Ep_,end_);
Trans(T_,open_,(F_,Tp_));
Trans(T_,nb_,(F_,Tp_));
Trans(T_,id_,(F_,Tp_));
ETrans(Tp_,plus_);
ETrans(Tp_,moins_);
Trans(Tp_,div_,(div_,F_,Tp_));
Trans(Tp_,mult_,(mult_,F_,Tp_));
ETrans(Tp_,close_);
ETrans(Tp_,end_);
Trans(F_,open_,(open_,E_,close_));
Trans(F_,nb_,(nb_));
Trans(F_,id_,(id_));
```

12.2 Analyseur LL(1) et grammaire S-attribuée

Listing complet de l'analyseur d'expressions LL(1) avec grammaire S-attribuée permettant le calcul direct du résultat.

```
#include <iostream>
using namespace std;
//#define debug(s) cout<<__FILE__<<":"<< \
//__LINE__<<" "<<__STRING(s)<<"="<<(s)<<endl
#define debug(s) ((void)(0))
#define fin(c) (c==EOF or c=='\n')

bool analyseE(double & val);
bool analyseTEp(double & val,int signe);
bool analyseEp(double & val);
bool analyseT(double & val);
bool analyseFTp(double & val,int inv);
bool analyseTp(double & val);
bool analyseF(double & val);
```

```
int next(void) {
   return cin.peek();
}

void shift(void) {
   cin.get();
}

bool analyseE(double & val) {
   debug("AnalyseE");
   if (next()=='(' or isdigit(next())) {
      if (analyseTEp(val,1)) {
         debug(val);
         return true;
      }
   }
   return false;
}

bool analyseTEp(double & val,int signe) {
   debug("AnalyseTEp");
   double valT,valEp;
   bool resT,resEp;
   resT = analyseT(valT);
   if (!resT)
      // pas la peine de continuer dans ce cas
      return false;
   resEp = analyseEp(valEp);
   if (resT and resEp) {
      val = signe*valT+valEp; // terme=addition
      debug(val);
      return true;
   }
   return false;
}

bool analyseEp(double & val) {
   debug("AnalyseEp");
   int signe = 1;
   if (next()=='+' or next()=='-') {
      if (next()=='-')
         signe = -1;
      shift();
      if (analyseTEp(val,signe)) {
         debug(val);
         return true;
      }
   }
   else if (next()==')' or fin(next())) {
      val = 0; // élément neutre de l'addition
      debug(val);
      return true;
   }
   return false;
}

bool analyseT(double & val) {
   debug("AnalyseT");
   if (next()=='(' or isdigit(next())) {
      if (analyseFTp(val,0))  {
         debug(val);
         return true;
      }
   }
   return false;
}

bool analyseFTp(double & val,int inv) {
```

```
   debug("AnalyseFTp");
   double valF,valTp;
   bool resF,resTp;
   resF = analyseF(valF);
   if (!resF)
      return false;
   resTp = analyseTp(valTp);
   if (resF and resTp) {
      if (inv)
         valF = 1.0/valF;
      val = valF*valTp; // facteur = mult.
      debug(val);
      return true;
   }
   return false;
}

bool analyseTp(double & val) {
   int inv;
   debug("AnalyseTp");
   if ((inv=(next()=='/')) or next()=='*') {
      shift();
      if (analyseFTp(val,inv)) {
         debug(val);
         return true;
      }
   }
   else if (next()=='+' or next()=='-'
            or next()==')' or fin(next())) {
      val = 1.0; //élement neutre de la mult.
      debug(val);
      return true;
   }
   return false;
}

bool analyseF(double & val) {
   debug("AnalyseF");
   if (next()=='(') {
      shift();
      if (analyseE(val)) {
         shift(); // la parenthèse fermante
         return true;
      }
   }
   else if (isdigit(next())) {
      double v=next()-'0';
      // lecture d'un entier
      // et conversion en double
      shift();
      while (isdigit(next())) {
         v = 10*v+next()-'0';
         shift();
      }
      val = v;
      debug(val);
      return true;
   }
   return false;
}

int main(void) {
   double val;
   analyseE(val);
   cout<<"La valeur résultat est "<<val<<endl;
   return 0;
}
```

INDEX