THEORIE DES LANGAGES

Claude MOULIN
Université de Technologie de Compiègne

30 janvier 2013

Table des matières

1 Compilation **9**
1.1 Introduction . . . 9
1.1.1 Références . . . 9
1.1.2 Historique . . . 9
1.1.3 Compilateur . . . 9
1.1.4 Interpréteur . . . 10
1.1.5 Langage Intermédiaire . . . 10
1.2 Phases de la compilation . . . 10
1.3 Analyse du programme source . . . 11
1.3.1 Analyse lexicale . . . 12
1.3.2 Analyse syntaxique . . . 12
1.3.3 Analyse sémantique . . . 13
1.4 Environnement du compilateur . . . 14
1.4.1 Préprocesseur . . . 14
1.4.2 Assembleur . . . 15
1.4.3 Chargeur et relieur . . . 15

2 Langages **17**
2.1 Alphabet . . . 17
2.2 Chaîne . . . 17
2.3 Opérations sur les chaînes . . . 18
2.3.1 Extension . . . 18
2.3.2 Propriétés de la concaténation . . . 18
2.3.3 Miroir . . . 18
2.4 Langage . . . 18
2.4.1 Exemples . . . 18
2.4.2 Opérations sur les langages . . . 19
2.4.3 Clôture de Kleene . . . 19

3 Expressions régulières **21**
3.1 Opérateurs $*$ et $|$ . . . 21
3.2 Exemples d'expressions régulières . . . 22
3.3 Extension des notations . . . 22
3.4 Priorité des opérateurs . . . 22
3.5 Lois algébriques sur les expressions régulières . . . 22

3.6 Définition régulière 23

4 Expressions régulières - Extensions 25
4.1 Principales caractéristiques 26
4.1.1 Classes de caractères 26
4.1.2 Groupes 26
4.1.3 Caractères limites 27
4.1.4 Facteurs de répétition 27
4.1.5 Caractères de prévision 27
4.1.6 Drapeaux 28
4.2 Exemples 28
4.3 Implémentation en Java 28
4.3.1 Classes 28
4.3.2 Méthodes de la classe Matcher 29
4.3.3 Classe java.lang.String 30
4.3.4 Exemple de programme 30

5 Automate à états finis 31
5.1 Introduction 31
5.2 Automate fini déterministe - AFD 31
5.2.1 Diagramme de transition - graphe d'automate 32
5.2.2 Table de transition 32
5.2.3 Extension de la fonction de transition 32
5.2.4 Exemple 33
5.3 Automate fini non déterministe - AFN 33
5.3.1 Exemple 33
5.3.2 Définition 34
5.3.3 Extension de la fonction de transition 34
5.3.4 Exemple 35
5.4 Automate fini avec ϵ-transitions (ϵ-AFN) 35
5.4.1 introduction 35
5.4.2 Définition 35
5.5 Automates finis et expressions régulières 36
5.5.1 Passage d'un AFD à une expression régulière 36
5.5.2 Méthode par élimination d'états 38
5.5.3 Passage d'un AFN à un AFD 40
5.5.4 Passage d'un ϵ-AFN à un AFD 41
5.5.5 Conversion d'une expression régulière en ϵ-AFN 42

6 Analyse Lexicale 45
6.1 Rôle d'un l'analyseur lexical 45
6.1.1 Introduction 45
6.1.2 Fonctions de l'analyseur lexical 45
6.1.3 Intérêts de l'analyse lexicale 46
6.2 Description 46
6.2.1 Définitions 46

6.2.2 Exemple . . . 46
6.3 Constructions de l'analyseur lexical . . . 47
6.3.1 Principe . . . 47
6.3.2 Ecœuils . . . 47
6.3.3 Compromis : temps d'exécution - place occupée . . . 48

7 Grammaires hors contexte 49
7.1 Introduction . . . 49
7.1.1 Existence de langages non réguliers . . . 49
7.1.2 Exemple : fragment de grammaire . . . 49
7.2 Définition formelle . . . 49
7.2.1 Conventions d'écriture . . . 50
7.3 Exemples . . . 50
7.3.1 Grammaire des palindromes sur $\{0, 1\}$ . . . 50
7.3.2 Petit sous-ensemble de la grammaire française . . . 50
7.4 Dérivation . . . 51
7.4.1 Extension des règles de production . . . 51
7.4.2 Dérivation la plus à gauche, la plus à droite . . . 52
7.4.3 Arbre de dérivation . . . 53
7.4.4 Ambiguïté . . . 54
7.5 Forme de BACKUS-NAUR . . . 54
7.6 Types de grammaires . . . 55
7.7 Expression régulière sous forme de grammaire . . . 56
7.8 Equivalence . . . 57
7.9 Types d'analyse . . . 57
7.9.1 Exemple d'analyse . . . 57
7.9.2 Analyse LL . . . 58
7.9.3 Analyse LR . . . 58

8 Automates à piles 59
8.1 Introduction . . . 59
8.1.1 Fonctionnement schématique . . . 59
8.1.2 Exemple . . . 60
8.2 Définition formelle . . . 60
8.2.1 Définition . . . 60
8.2.2 Règles de transition . . . 61
8.2.3 Automate à pile déterministe . . . 61
8.3 Exemple . . . 62
8.3.1 Diagramme de transition . . . 62
8.3.2 Simulation . . . 62
8.4 Extension de la fonction de transition . . . 64
8.4.1 Configuration . . . 64
8.4.2 Définition . . . 64
8.5 Langage d'un automate à pile . . . 64
8.5.1 Acceptation par état final . . . 64
8.5.2 Acceptation par pile vide . . . 64

8.5.3 Théorèmes 65

9 Analyse Syntaxique Descendante 67
9.1 Introduction 67
9.2 Analyseur descendant LL(1) 67
9.2.1 Problématique 67
9.2.2 Premiers 69
9.2.3 Suivants 70
9.2.4 Table d'analyse 71
9.2.5 Simulation par un automate à pile 71
9.3 Grammaire LL(1) 73
9.3.1 Récursivité à gauche 73
9.3.2 Factorisation à gauche 74
9.4 Conflits 75
9.4.1 Conflit Premier-Premier 75
9.4.2 Conflit Premier-Suivant 76
9.4.3 Conclusion 77

10 Analyse Ascendante LR(0) 79
10.1 Principe d'analyse ascendante 79
10.1.1 Exemple d'analyse 79
10.1.2 Décalage - Réduction 80
10.1.3 Méthodes d'analyse 80
10.1.4 Manche 80
10.2 Item LR(0) 81
10.2.1 Définition 81
10.2.2 Fermeture d'un ensemble d'items 82
10.3 Automate caractéristique canonique 82
10.3.1 Définition 82
10.3.2 Diagramme de transition 82
10.3.3 Transitions 82
10.3.4 Collection des items d'une grammaire 84
10.3.5 Construction des items 84
10.4 Analyse LR(0) 85
10.4.1 Table des actions 85
10.4.2 Illustration 86
10.4.3 Algorithme d'analyse 86
10.4.4 Simulation de l'automate 87
10.4.5 Conflits 87

11 Analyse Ascendante SLR(1) 89
11.1 Introduction 89
11.2 Algorithme de construction 89
11.3 Tables SLR(1) 90
11.4 Grammaire SLR(1), non LR(0) 91
11.4.1 Premiers - Suivants 92

11.4.2 Tables d'analyse . . . 93
11.4.3 Simulation de l'automate . . . 93
11.5 Conflits . . . 94
11.5.1 Conflit Décaler - Réduire . . . 94
11.5.2 Conflit Réduire - Réduire . . . 94

12 Analyse LR(1) 95
12.1 Exemple . . . 95
12.2 Introduction . . . 95
12.3 Items LR(1) . . . 96
12.3.1 Définition . . . 96
12.3.2 Fermeture d'un ensemble d'items LR(1) . . . 96
12.3.3 Transition . . . 96
12.3.4 Calcul de l'ensemble des items . . . 97
12.3.5 Tables d'analyse LR(1) . . . 97
12.3.6 Illustration . . . 97
12.3.7 Exemples de conflits LR(1) . . . 99
12.4 Analyse LALR(1) . . . 99
12.4.1 Cœur d'un ensemble d'items . . . 99
12.4.2 Fusion de deux états . . . 100
12.4.3 Tables d'analyse LALR . . . 100
12.4.4 Exemple . . . 101
12.4.5 Conflits . . . 102
12.4.6 Grammaire LR(1) non LALR(1) . . . 102

13 Analyse Sémantique 105
13.1 Introduction . . . 105
13.2 Grammaire attribuée . . . 106
13.2.1 Exemple . . . 106
13.3 Attributs . . . 107
13.3.1 Arbre décoré . . . 107
13.3.2 Définition formelle . . . 108
13.3.3 Ordre d'évaluation des attributs . . . 108
13.3.4 Exemple . . . 108
13.4 Traduction dirigée par la syntaxe . . . 108
13.4.1 Analyse ascendante . . . 109
13.4.2 Analyse descendante . . . 110
13.4.3 Principe . . . 110
13.4.4 Définition L-attribuée . . . 111
13.5 Arbre syntaxique abstrait - AntLR (v 3) . . . 112
13.5.1 Exemple : expression arithmétique . . . 113
13.5.2 Arbre abstrait . . . 113
13.5.3 Evaluation récursive . . . 114
13.6 Traduction de langage . . . 114
13.7 AntLR version 4 . . . 116
13.7.1 Evolution d'AntLR . . . 116

13.7.2 Grammaire d'expression . . . 116
13.7.3 Nouvelles caractéristiques d'AntLR . . . 117

A Générateurs d'analyseur 121
A.1 Commandes unix . . . 121
A.2 Générateurs en langage Java . . . 121

B Le générateur AntLR 123
B.1 AntLR version 3 . . . 123
B.1.1 Exemple simple . . . 123
B.1.2 Mise en œuvre . . . 124
B.1.3 Autre exemple . . . 127
B.1.4 Création de l'arbre AST . . . 128
B.1.5 Actions AntLR . . . 130
B.2 AntLR version 4 . . . 132
B.2.1 Exemple : grammaire d'expressions arithmétiques . . . 132
B.2.2 Mise en œuvre . . . 133
B.2.3 Principe du visitor . . . 135

C Introduction à Java 5.0 137
C.1 Introduction . . . 137
C.1.1 Objectifs . . . 137
C.1.2 Environnements de développement . . . 137
C.2 Programmation orientée objet . . . 138
C.2.1 Classes et Objets . . . 138
C.2.2 API Java . . . 138
C.2.3 Entrées-Sorties sur la console . . . 138
C.2.4 Une application en Java . . . 139
C.3 Éléments de base . . . 140
C.3.1 Types primitifs . . . 140
C.3.2 Types des variables . . . 140
C.3.3 Structures de contrôle des méthodes . . . 140
C.4 Construction d'une classe . . . 143
C.4.1 Portée - Visibilité . . . 143
C.4.2 Constructeurs . . . 144
C.4.3 Interface . . . 144
C.4.4 Application simple . . . 145
C.5 Conteneur Générique . . . 146
C.5.1 Déclaration . . . 146
C.5.2 Itération . . . 146
C.6 Héritage de Classes . . . 146
C.7 Sites Utiles . . . 147

Chapitre 1

Compilation

1.1 Introduction

1.1.1 Références

On pourra consulter avec profit les ouvrages référencés dans la bibliographie : [1, 2, 6, 3, 5, 4, 7, 8, 9].

1.1.2 Historique

- Avant 1950
  - Langage machine : suite de nombres écrits dans le système hexadécimal. Certains nombres représentent des instructions, d'autres des valeurs, etc. (0A 43)
  - Langage d'assemblage : il contient des instructions faisant intervenir les registres de l'unité centrale de l'ordinateur, les données.
    Ex : ADD #2, R1
  - Introduction des étiquettes dans le langage d'assemblage dans le but de faciliter les branchements (boucles, sauts).
  - Introduction des variables
    Ex : MOV R1, b
- Définition des Langages de haut niveau (1950)
  - expressions
  - fonctions
  - structures de données
- Définition des langages à objets (1970)
  - Classes, Objets, méthodes, variables d'instance
  - Héritage
  - polymorphisme

1.1.3 Compilateur

Tout programme écrit dans un langage de haut niveau doit être traduit en instructions exécutables par un ordinateur. Le programme effectuant cette traduction est un compilateur.
Il est préférable que le programme soit indépendant du système sur lequel il doit s'exécuter.

Ce n'est pas toujours le cas. On peut introduire des instructions en langage d'assemblage dans le langge C par exemple.
Le programme n'a pas besoin du compilateur lorsqu'il est exécuté.
La compilation est un peu plus compliquée.
fichier source 1 ⟶ compilateur ⟶ fichier objet 1
fichier source 2 ⟶ compilateur ⟶ fichier objet 2
fichier objet 1, fichier objet 2 ⟶ programme exécutable
Exemples de langages compilés : Pascal, C, C++, ADA, Fortran, Cobol

1.1.4 Interpréteur

Un interpréteur travaille simultanément sur les données et sur le programme source. Il analyse une Ãă une les instructions du programme, les “compile” et les exécute.

- L'interpréteur doit être présent sur le système où le programme s'exécute
- L'exécution est plus lente
- On peut modifier le source pendant l'exécution

Exemples de langages interprétés : Basic, Lisp, Scheme, Tcl, Perl, Prolog, Smaltalk

1.1.5 Langage Intermédiaire

Le code source est traduit dans une forme binaire (pseudo-code). Ce code est interprété et exécuté. Exemple de Java : fichier source (.java) ⟶ compilateur (javac) ⟶ fichier objet (.class)
Les fichiers objet (et les ressources) peuvent être regroupés dans des fichiers archives (.jar).
Les fichiers objet ⟶ interpréteur (JVM) ⟶ résultat.
Exemples de langages intermédiaires : Java, Python.

1.2 Phases de la compilation

Le but d'un compilateur est de traduire un programme source écrit dans un certain langage en un programme cible écrit dans un dans un autre langage. Le plus souvent, le langage cible est le langage machine, mais cela peut être seulement un code intermédiaire optimisé (byte code java).

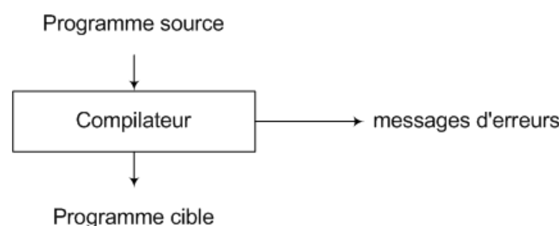


FIGURE 1.1 – Compilation

La compilation se décompose en 6 phases : 3 phases d'analyse et 3 phases de génération de code. La figure 1.2 montre le fonctionnement typique d'un compilateur, même si dans la pratique certaines phases peuvent être regroupées. Deux autres tâches interagissent avec les six phases :

- la gestion d'une table des symboles. Celle-ci contient les identificateurs des variables et leurs attributs (type, portée, etc.).
- la détection des erreurs. Une phase ne doit pas s'interrompre à la première erreur rencontrée mais doit traiter les erreurs de façon à ce que la compilation puisse continuer.

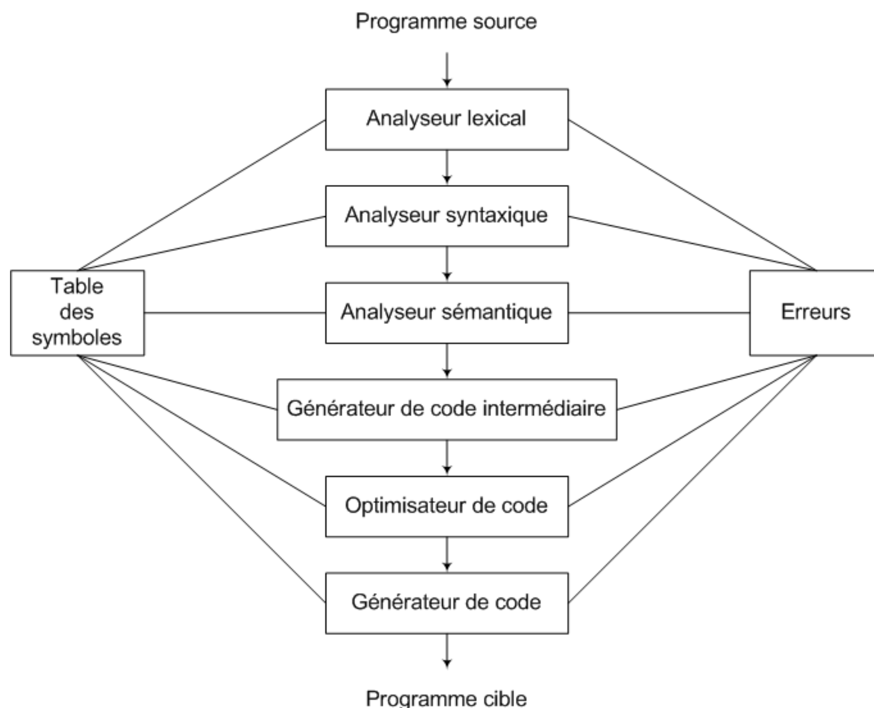


FIGURE 1.2 – Les différentes étapes de la compilation

1.3 Analyse du programme source

L'analyse du programme source est réalisée durant les trois premières phases de la compilation.

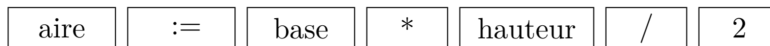
- Analyse lexicale. Le flot des des caractères du programme source est regroupé en suites ayant une signification précise et appartenant à des catégories prédéfinies appelées unités lexicales.
- Analyse syntaxique. Elle regroupe les unités lexicales en structures ou unités grammaticales qui permettent de synthétiser le résultat de la compilation.
- Analyse sémantique. Elle contrôle la cohérence sémantique du programme source. En particulier, elle vérifie la cohérence des types des variables et des expressions et

procède parfois à des conversions (entier vers réel).
La représentation interne du programme source évolue en fonction des phases de la compilation. Nous illustrons ces représentations par l'exemple suivant. Il s'agit de traduire une instruction extraite d'un programme source :

$$aire := base * hauteur/2 \tag{1.1}$$

1.3.1 Analyse lexicale

L'analyse lexicale reconnaît les lexèmes suivants :



aire, base, hauteur sont des lexèmes représentants de la même unité lexicale appelée par exemple identificateur.
La représentation de l'instruction 1.1 peut être suggérée par :

$$id_1 := id_2 * id_3/2 \tag{1.2}$$

Après analyse lexicale, la chaîne qui contient l'instruction source :

aire := base * hauteur / 2

peut être représentée par la suite :
<identificateur> aire
<affectation> :=
<identificateur> base
<opérateur> *
<identificateur> hauteur
<opérateur> /
<nombre> 2

1.3.2 Analyse syntaxique

L'analyse syntaxique regroupe les unités lexicales en structures grammaticales en suivant les règles figurant dans une grammaire. On représente souvent par un arbre syntaxique le résultat produit par l'analyse syntaxique.

La structure hiérarchique d'un programme est exprimée à l'aide de règles. Ainsi :
Tout identificateur est une expression
Tout nombre est une expression

Certaines règles sont récursives :
Si $expr_1$ et $expr_2$ sont des expressions alors $expr_1 * expr_2$ est une expression

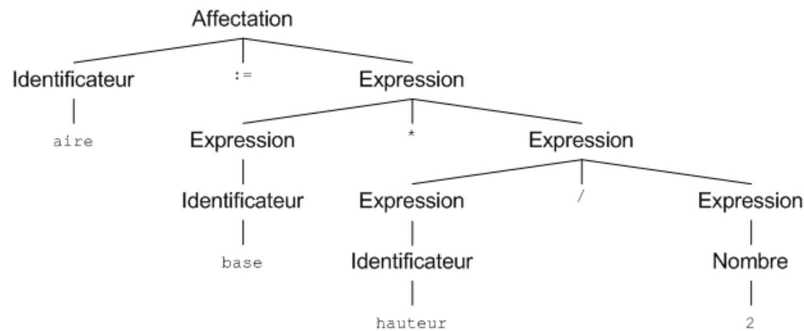


FIGURE 1.3 – L'arbre syntaxique

1.3.3 Analyse sémantique

L'analyse sémantique va contrôler les erreurs sémantique et collecte des informations de type destinées aux phases suivantes de la compilation.
Une partie importante est le contrôle des types des variables (identificateurs de variables : aire, base, hauteur) dans les instructions.

- Affectation
- Opérandes dans les opérations, facteurs dans les expressions
- Type de retour des fonctions

Les erreurs de types sont généralement signalées et nécessitent une correction :
Ex : nombre réel employé comme indice d'un tableau.

Cependant, l'analyse sémantique peut éventuellement réaliser la coercition de valeurs ou de constantes. Dans l'instruction étudiée ci-dessus, les identificateurs base, hauteur sont normalement déclarés comme réels. Les opérations, multiplication, division ont des opérandes de même type (tous deux réels). L'entier 2 va donc être substitué par un réel (2.0). Le codage machine des entiers et des réels n'étant pas le même cette coercition est rendue nécessaire. La figure 1.4 montre l'appel à un opérateur supplémentaire.

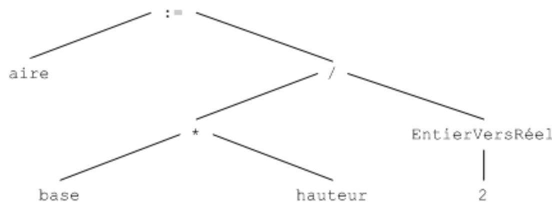


FIGURE 1.4 – Coercition

1.4 Environnement du compilateur

La création d'un programme cible exécutable requiert en fait plusieurs programmes en plus du strict compilateur. La figure 1.5 indique ces différents programmes.

1.4.1 Préprocesseur

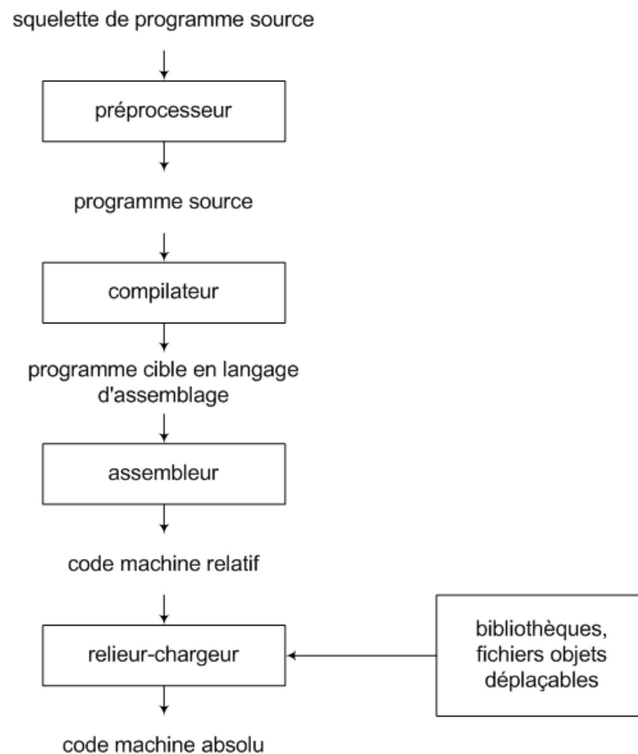


FIGURE 1.5 – Traitement d'un programme source

Le programme source est en général divisé en modules stockés dans différents fichiers. Le préprocesseur a pour tâche de reconstituer le programme source à partir des différents modules le constituant. Certains langages possèdent également des macros-instructions. Elles ont pour but de remplacer des éléments des fichiers d'entrée par des instructions du langage.
Ainsi en langage C, on peut trouver le contenu suivant :

```
#ifndef LISTE_H
#define LISTE_H
#include <stdio.h>
#include <assert.h>
#define CR 13
```

1.4.2 Assembleur

Certains compilateurs produisent du code en langage d'assemblage qui doit être traité par un assembleur. Le langage d'assemblage est une version mnémonique de code machine dans laquelle on utilise des noms au lieu du code binaire pour désigner des opérations et dans laquelle on utilise des noms pour désigner les adresses mémoire. Ainsi le code suivant permet d'additionner 2.

```
MOV a, R1
ADD #2, R1
MOV R1, b
```

a et b sont des emplacements mémoire, R1 est un registre et le code réalise l'instruction : b := a + 2. Le code produit par l'assembleur est un code machine (écrit en langage binaire) où les adresses sont relatives. Il peut donc être chargé en mémoire à partir de n'importe qu'elle adresse.

1.4.3 Chargeur et relieur

Un chargeur est un programme qui charge le programme en mémoire, c'est-à dire qu'il prend le code machine, modifie les adresses relatives en adresse absolues et place les instructions et les données aux endroits appropriés.
Le relieur ou éditeur de liens permet de constituer un unique programme à partir de plusieurs fichiers contenant du code machine.

Chapitre 2

Langages

2.1 Alphabet

Définition 2.1 (Alphabet) *Un alphabet est un ensemble fini non vide de symboles appelés également lettres de l'alphabet.*

Exemples :

1. $\Sigma = \{0, 1\}$
2. $\Sigma = \{a, b, ..., z\}$

2.2 Chaîne

Définition 2.2 (Chaîne) *Une chaîne est une suite finie de symboles choisis dans un alphabet.*

On note une chaîne par la suite des symboles qu'elle contient : $c = x_1x_2...x_n$. Exemples à partir des alphabets 1 et 2 :
1. 01011, 111, 00, ...
2. abc, aa, xyzt, ...

Définition 2.3 (Chaîne vide) *La chaîne vide est la chaîne ne contenant aucun symbole.*

Cette chaîne, notée ϵ, peut être construite sur tout alphabet.

Définition 2.4 (Longueur d'une chaîne) *La longueur d'une chaîne est le nombre de symboles la composant.*

1. $|abc| = 3$
2. $|\epsilon| = 0$

2.3 Opérations sur les chaînes

Définition 2.5 (Concaténation de deux chaînes) *Soient* $u = x_1x_2...x_n$ *et* $v = y_1y_2...y_m$ *deux chaînes non vides définies sur l'alphabet* Σ*, on définit par* $uv = x_1x_2...x_ny_1y_2...y_m$ *la chaîne concaténation de u et de v.*

Préfixe, Suffixe

Soient u, v et w, trois chaînes non vides telles que $w = uv$.
On dit que u est une chaîne préfixe de w et que v est une chaîne suffixe de w. $|u| < |w|$ et $|v| < |w|$.

2.3.1 Extension

On peut étendre la définition de la concaténation à la chaîne vide.
Si u est une chaîne non vide définie sur l'alphabet Σ : $\epsilon u = u\epsilon = u$
On note x^n, $n \in N$ la concaténation de n symboles identiques.
$x^0 = \epsilon$ et $x^1 = x$.
On note également u^n, $n \in N$ la concaténation de n chaînes identiques.

2.3.2 Propriétés de la concaténation

La concaténation est associative : $\forall u, \forall v, \forall w, (uv)w = u(vw)$
La concaténation possède pour élément neutre ϵ
$|uv| = |u| + |v|$
$|u^n| = n.|u|$, $n \in N$
La concaténation n'est pas commutative. En général, $uv \neq vu$.

2.3.3 Miroir

Définition 2.6 (Miroir) *La chaîne miroir d'une chaîne c est la chaîne formée par la suite des symboles composant c mais pris dans l'ordre inverse.*

si $u = 0101$ alors miroir$(u) = 1010$.
si $v = \epsilon$ alors miroir$(v) = v = \epsilon$.

2.4 Langage

Définition 2.7 (Langage) *Un langage défini sur un alphabet* Σ *est un ensemble de chaînes définies sur* Σ*.*

2.4.1 Exemples

1. Σ_2 est le langage composé de toutes les chaînes de longueur 2 sur Σ
2. Σ^* est le langage composé de toutes les chaînes constructibles sur l'alphabet Σ
3. Pour tout langage L construit sur un alphabet Σ on a : $L \subseteq \Sigma^*$

2.4.2 Opérations sur les langages

Définition 2.8 (Réunion, Intersection de deux langages) *Soient L_1 et L_2 deux langages définis sur un alphabet Σ.*
$L_1 \cup L_2$ et $L_1 \cap L_2$ sont aussi deux langages définis sur Σ.

Définition 2.9 (Concaténation de deux langages) *Soient L_1 et L_2 deux langages définis sur un alphabet Σ.*
$L_1L_2 = \{uw/u \in L_1, w \in L_2\}$ est le langage obtenu par concaténation de L_1 et de L_2.

2.4.3 Clôture de Kleene

Définition 2.10 (Clôture d'un langage) *Soit L un langage défini sur un alphabet Σ. La clôture de Kleene du langage L, notée L^* est l'ensemble des chaînes qui peuvent être formées en prenant un nombre quelconque (éventuellement 0) de chaînes de L et en les concaténant.*

$L^0 = \{\epsilon\}$
$L^1 = L$
$L^2 = LL$
$L^i = LL...L$, la concaténation de i copies de L
$L^* = \bigcup_{i \geq 0} L^i$

Chapitre 3

Expressions régulières

Les expressions régulières sont une façon déclarative de décrire la formation des chaînes d'un langage. Si E est une expression régulière on notera $L(E)$ le langage engendré par E.

3.1 Opérateurs $*$ et $|$

Considérons l'alphabet $\Sigma = \{0, 1\}$

1. Expression 1 : 0*
   Cette expression régulière décrit toutes les chaînes composées uniquement du symbole 0 ainsi que la chaîne vide ϵ.
2. Expression 2 : 110*1
   Cette expression régulière décrit les chaînes contenant deux fois le symbole 1 suivis d'un nombre quelconque de symboles 0, éventuellement nul, et du symbole 1.
3. 0*110*1
   est la concaténation des expressions 1 et 2. Elle décrit les chaînes commençant par un nombre quelconque de 0 suivis de deux 1, suivi d'un nombre quelconque de 0 et d'un 1.
4. 0* | 110*1
   est l'expression régulière décrivant les chaînes générées soit par l'expression 1, soit par l'expression 2.

Les règles suivantes permettent de décrire les expressions régulières définies sur un alphabet Σ :

Définition 3.1 (Expressions régulières) *1. la chaîne vide ϵ et l'ensemble vide $\emptyset$ sont des expressions régulières.*
$L(\epsilon) = \{\epsilon\}$ et $L(\emptyset) = \emptyset$.

2. $\forall a \in \Sigma$, la chaîne a formée de l'unique symbole a est une expression régulière.

3. Si E et E' sont des expressions régulières alors :

   (a) EE' est une expression régulière, concaténation de E et E'.
   Elle décrit $L(EE') = L(E)L(F)$

(b) $E \mid E'$ *est une expression régulière.*
Elle décrit $L(E \mid E') = L(E) \cup L(F)$

(c) (E) *est une expression régulière, décrivant le même langage que* E.
$L((E)) = L(E)$.

(d) E^* *est une expression régulière.* $L(E^*) = L(E)^*$

Définition 3.2 (Langage régulier) *Un langage régulier est un langage représenté par une expression régulière.*

3.2 Exemples d'expressions régulières

$a \mid b$
$(a \mid b)^*abba$
$a^*b(a^*b)^*$
$(a \mid b)^*b$
$(1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)(0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)^*$

3.3 Extension des notations

Soit E une expression régulière :

- Opérateur unaire + : au moins une fois
  $E^+ = EE^*$

- Opérateur unaire ? : 0 ou 1 fois
  $E? = E \mid \epsilon$

- Classes de caractères :
  $[abc] = a \mid b \mid c$
  $[a - z] = a \mid b \mid c \mid ... \mid z$
  $[A - Ca - z] = A \mid B \mid C \mid a \mid b \mid c \mid ... \mid z$

3.4 Priorité des opérateurs

Les opérateurs unaires + et * ont la plus haute priorité.
La concaténation a la deuxième plus haute priorité et est associative à gauche.
| a la plus faible priorité et est associative à gauche.

Exemple : $(a) \mid ((b)^*(c))$ équivaut à $a \mid b^*c$

3.5 Lois algébriques sur les expressions régulières

Définition 3.3 (Expressions égales) *Deux expressions régulières sont égales si elles engendrent le même langage.*

Propriétés

$E \mid E' = E' \mid E$	commutativité de la réunion
$E \mid (E' \mid E'') = (E \mid E') \mid E''$	associativité de la réunion.
$\emptyset \mid E = E \mid \emptyset = E$	$\emptyset$ est l'élément neutre pour la réunion.
$E \mid E = E$	idempotence de la réunion.
$E(E'E'') = (EE')E''$	associativité de la concaténation.
$\epsilon E = E\epsilon = E$	ϵ est l'élément unité pour la concaténation.
$\emptyset E = E\emptyset = \emptyset$	$\emptyset$ est élément absorbant pour la concaténation.
$E(E' \mid E'') = EE' \mid EE''$	distributivité à gauche de la concaténation par rapport à la réunion.
$(E \mid E')E'' = EE'' \mid E'E''$	distributivité à droite de la concaténation par rapport à la réunion.

Quelques conséquences
$(E^*)^* = E^*$
$\emptyset^* = \epsilon$
$\epsilon^* = \epsilon$
$E^+ = EE^* = E^*E$
$E^+ \mid \epsilon = E^*$
$E? = E \mid \epsilon$
$p(qp)* = (pq) * p$
$(p|q)* = (p * |q*)*$
$(p * q*)* = (p|q)* = (p * |q*)*$

3.6 Définition régulière

Pour des raisons de commodité, il est possible de donner des noms explicites à des expressions régulières. On utilise le symbole $\longrightarrow$ pour écrire leur définition.
Exemples :
lettre $\longrightarrow$ [A-Za-z]
chiffre $\longrightarrow$ [0-9]
id $\longrightarrow$ lettre (lettre | chiffre)*
chiffres $\longrightarrow$ chiffre (chiffre)*
fraction $\longrightarrow$. chiffres | ϵ
exposant $\longrightarrow$ E(+ | - | ϵ) chiffres | ϵ
nombre $\longrightarrow$ chiffres fraction exposant
A partir de cette définition :

- id reconnaît : a, a0b, begin
- nombre reconnaît : 0, 1.0, 2E4, 1.5E-8, 0.25E-0
- nombre ne reconnaît pas : 0., .1, 1E2.0

Chapitre 4

Expressions régulières - Extensions

La plupart des langages de programmation intègrent des possibilités d'analyse de chaînes à partir d'expressions régulières. Ces possibilités étendent les opérateurs vus dans le chapitre précédent.
Des méthodes permettent de déterminer si la totalité d'une chaîne satisfait le patron d'une expression régulière ou si seulement une portion de la chaîne le satisfait. Dans ce cas, il est possible de recommencer plusieurs fois la recherche et de trouver l'ensemble des parties de la chaîne qui satisfont l'expression régulière.

4.1 Principales caractéristiques

4.1.1 Classes de caractères

[...]	Union des caractères entre les crochets
[^ ...]	Complémentaire des symboles entre les crochets
$[x - y]$	L'un des symboles entre x et y
\d	[0-9]
\D	[^0-9]
\w	[a-zA-Z0-9]
\W	[^a-zA-Z0-9]
\s ou \p{Space}	[\t\n\x0B\f\r]
\S	[^\t\n\x0B\f\r]
\p{Lower}	Un caractère minuscule
\p{Upper}	Un caractère majuscule
\p{ASCII}	Un caractère dont le code ascii est compris entre 0 et 128
\p{Alpha}	Un caractère minuscule ou majuscule
\p{Alnum}	\p{Alpha} ∪ \p{Digit}
\p{Punct}	[! "#$%&'()*+,-./:;<=>?@[\]^_‘{|}~]
\p{Print}	\p{Alnum} ∪ \p{Punct}
\p{Blank}	Caractères espace ou tabulation
\p{XDigit}	Un chiffre hexadécimal [0-9a-fA-F]
.	Un caractère quelconque
$< cl1 > \mid < cl2 >$	Union des classes cl1 et cl2
$[< cl1 >$ && $< cl2 >]$	Intersection des classes cl1 et cl2

4.1.2 Groupes

Les parenthèses dans une expression régulière sont des méta symboles permettant de délimiter des groupes. Chaque groupe possède un rang unique. Pour connaître le rang d'un groupe dans une expression régulière, il suffit de compter le nombre de parenthèses ouvrantes le précédant. Le rang du groupe est égal à ce nombre de parenthèses.
Il est possible d'indiquer qu'un groupe est non capturant. Dans ce cas, le groupe n'a pas de rang attribué. Pour déterminer le rang d'un groupe, il faut retirer le nombre de groupes non capturant le précédant en appliquant la règle précédente. Dans une expression régulière, il est possible de faire référence à un groupe capturant précédent et de l'insérer dans le patron.

(...)	Groupe capturant
(? :...)	Groupe non capturant
\n	Référence arrière au groupe capturant de rang n

4.1.3 Caractères limites

^ ou \a	Début de chaîne
$ ou \z	Fin de chaîne
(?m)^	Début de ligne ou de chaîne
(?m)$	Fin de ligne ou de chaîne
\b	Début ou fin de mot
\B	Absence de début ou de fin de mot
\G	Fin du précédent groupe satisfaisant le gabarit

4.1.4 Facteurs de répétition

{n}	Exactement n occurrences d'une classe de caractères
+ ou {n,}	Au moins n occurrences d'une classe de caractères
{n,p}	De n à p occurrences d'une classe de caractères
?	0 ou 1 occurrence
*	Un nombre quelconque de fois, éventuellement zéro
*?, +?	Opérateurs passifs
?+, *+, ++	Opérateurs possessifs

Les opérateurs ?, *, +, {n,p} sont avides, c'est-à-dire qu'ils essaient de satisfaire le patron en utilisant le maximum de caractères possible. On construit des opérateurs passifs en leur adjoignant le symbole ? (*?, +?). Dans ce cas, ils essaient de satisfaire le patron en utilisant le minimum de caractères possible. On construit des opérateurs possessifs en leur adjoignant le caractère + (?+, *+, ++). Ils sont alors indépendants du contexte.
Il est à noter que l'opérateur *? est satisfait par une chaîne vide.

4.1.5 Caractères de prévision

(?=...)	Prévision positive avant
(?!...)	Prévision négative avant
(?<=...)	Prévision positive arrière
(?<!...)	Prévision négative arrière

Les séquences de caractères de prévision permettent de faire des assertions, positives ou négatives sur les caractères qui suivent ou précèdent sans consommer de caractères : le pointeur dans la chaîne à partir duquel se fait l'analyse n'est pas avancé ou reculé selon les caractères satisfaisant le patron.

4.1.6 Drapeaux

(?i)	L'analyse est faite indépendamment de la casse des caractères, minuscule ou majuscule
(?m)	Pour rendre `^` et `$` début et fin de ligne ou de chaîne
(?s)	Pour que le caractère `.` satisfasse n'importe quel caractère, y compris le retour à la ligne
(?-i)	Annule (?i)
(?-m)	Annule (?m)
(?-s)	Annule (?s)
(?ims)	Pour plusieurs drapeaux

4.2 Exemples

Les exemples suivant montrent des sous-chaînes satisfaisant un patron.

Patron	\d+
Texte	rge5r43
Résultat	5
Patron	\p{Upper}+
Texte	rgEF5r-4'3
Résultat	EF
Patron	(?i)(\w{1,3})(\d{1,2})\2\1
Texte	-abC4141aBc-
Résultat	abC4141aBc
Patron	<B>.*</B>
Texte	...<B>millions</B> et des <B>milliards</B>
Résultat	<B>millions</B> et des <B>milliards</B>
Patron	<B>.*?</B>
Texte	...<B>millions</B> et des <B>milliards</B>
Résultat	<B>millions</B>
Patron	(?<=\D\d).(?=\d{2,})
Texte	25.3a512b.3.5135
Résultat	a

4.3 Implémentation en Java

4.3.1 Classes

Classe java.util.regex.Pattern

Cette classe permet d'instancier des patrons basés sur une expression régulière. Ainsi, si `patternString` est une chaîne contenant une expression régulière, on peut créer un objet `p` de type `Pattern` :

```
Pattern p = Pattern.compile(patternString);
```

Classe java.util.regex.Matcher

Cette classe permet d'instancier des objets Matcher associés à un objet Pattern et basés sur une chaîne à tester. Ainsi si `p` est un objet de type `Pattern` et `text` une chaîne à tester, on peut créer un objet `Matcher` par :

```
Matcher m = p.matcher(text);
```

4.3.2 Méthodes de la classe Matcher

find

Cette méthode analyse la chaîne d'entrée en cherchant la prochaine séquence qui satisfait le patron.

matches

Cette méthode essaie de satisfaire la chaîne entière en regard du patron.

start

Cette méthode retourne le rang du premier caractère de la séquence qui satisfait le patron.

end

Cette méthode retourne le rang du dernier caractère de la séquence qui satisfait le patron.

group

Cette méthode retourne la chaîne qui satisfait le patron.

groupCount

Cette méthode retourne le nombre de groupes dans la chaîne qui satisfait le patron.

group(int n)

Cette méthode retourne le groupe de rang n dans la chaîne qui satisfait le patron.

appendReplacement(StringBuffer sb,String s)

Cette méthode ajoute au buffer la chaîne s en remplacement de la séquence qui satisfait le patron.

appendTail(StringBuffer sb)

Cette méthode ajoute au buffer la fin de la chaîne d'entrée.

4.3.3 Classe java.lang.String

Plusieurs méthodes de la classe Sting utilisent en paramètre une chaîne contenant une expression régulière.
`public String replaceFirst(String regex,String replacement)`
Cette méthode construit une chaîne en remplaçant la première occurence qui satisfait l'expression régulière par la chaîne `replacement`.

`public String replaceAll(String regex,String replacement)`
Cette méthode construit une chaîne en remplaçant toutes les occurences qui satisfont l'expression régulière par la chaîne `replacement`.

`public String[] split(String regex)`
Cette méthode divise la chaîne selon les occurences qui satisfont l'expression régulière.

4.3.4 Exemple de programme

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
...
public static void scan() {
  String patternString = "^.*?\\p{Alpha}|[.].*?\\p{Alpha}";
  String text = "il. est. conseillé de. souvent. rire.";
  Pattern p = Pattern.compile(patternString);
  Matcher m = p.matcher(text);
  StringBuilder sb = new StringBuilder();
  boolean found = m.find();
  while (found) {
    m.appendReplacement(sb,m.group().toUpperCase());
    found = m.find();
  }
  m.appendTail(sb);
  System.out.println(sb.toString());
}
```

Chapitre 5

Automate à états finis

5.1 Introduction

Les automates sont des modèles mathématiques qui prennent en entrée une chaîne de symboles et qui effectuent un algorithme de reconnaissance de la chaîne. Si l'algorithme se termine dans certaines conditions, l'automate est dit accepter la chaîne correspondante. Le langage reconnu par un automate est l'ensemble des chaînes qu'il accepte. La figure 5.1 représente un automate qui accepte toute chaîne comportant une suite de caractères 01.

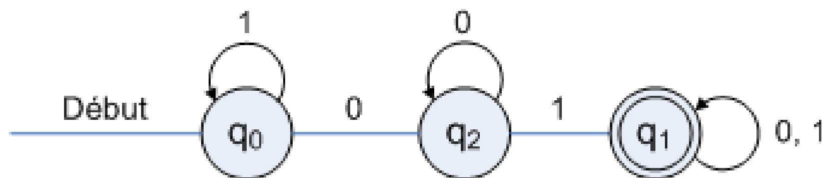


FIGURE 5.1 – Exemple d'automate

5.2 Automate fini déterministe - AFD

Définition 5.1 (AFD) *un automate fini déterministe (AFD) est un modèle mathématique qui consiste en :*

- Q *un ensemble d'états.*
- Σ *un ensemble de symboles d'entrée (alphabet).*
- δ *une fonction de transition qui prend comme argument un état et un symbole d'entrée et qui retourne un état.* $\delta : Q \times \Sigma \longrightarrow Q$.
- q_0*, un état initial appartenant à* Q.
- F*, un ensemble d'états finals ou états d'acceptation,* $F \subset Q$.

Un automate A est un quintuplet : $A = (Q, \Sigma, \delta, q_0, F)$

5.2.1 Diagramme de transition - graphe d'automate

Un automate est représenté par un graphe orienté étiqueté, appelé *graphe de transition* tel que :

- Pour chaque état q, $q \in Q$, il existe un nœud étiqueté q.
- Pour chaque état q et chaque symbole a de Σ tel que $\delta(q, a) = p$, il existe un arc du nœud q vers le nœud p étiqueté a.
- Les nœuds correspondant aux états de satisfaction (états appartenant à F) sont représentés par un cercle double.

Exemple : Soit le langage $L = \{w | w$ possède un nombre pair de 0 et de 1 $\}$

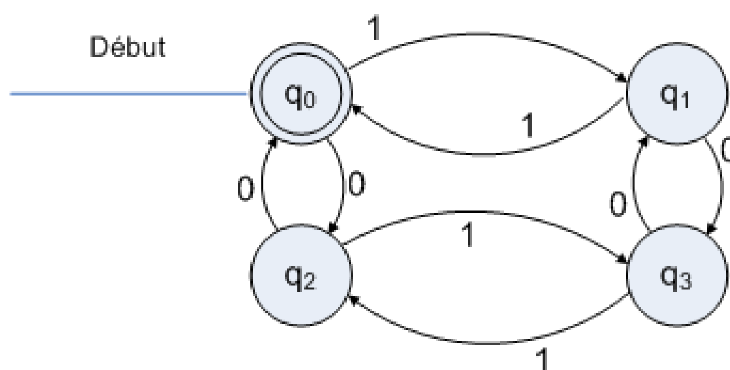


FIGURE 5.2 – Le graphe du langage L

L'état q_0 correspond à un nombre pair de 0 et de 1
L'état q_1 correspond à un nombre impair de 1
L'état q_2 correspond à un nombre impair de 0
L'état q_3 correspond à un nombre impair de 0 et de 1

5.2.2 Table de transition

	0	1	
$* \rightarrow q_0$	q_2	q_1	
q_1	q_3	q_0	
q_2	q_0	q_3	$\rightarrow$ désigne l'état initial
q_3	q_1	q_2	* désigne les états finals

exemple : $\delta(q_0, 0) = q_2$

Définition 5.2 (Langage engendré par un automate - 1) *Le langage d'un automate à états finis déterministe est l'ensemble des chaînes obtenues sur tout chemin du graphe partant du nœud initial et s'achevant sur un nœud final.*

5.2.3 Extension de la fonction de transition

La fonction de transition étendue appelée $\hat{\delta}$ décrit ce qui se passe lorsqu'on se positionne dans un état quelconque de l'automate et que l'on suit une séquence quelconque d'entrées.

$\hat{\delta} : Q \times \Sigma* \longrightarrow Q$ est définie par induction sur la longueur de la chaîne de symboles d'entrée par :

- $\hat{\delta}(q, \epsilon) = q$
- Soit $w = xa$, a le dernier symbole de w et x la chaîne contenant tous les autres symboles de w,
  si $\hat{\delta}(q, x) = p$, alors $\hat{\delta}(q, w) = \delta(p, a) = \delta(\hat{\delta}(q, x), a)$

5.2.4 Exemple

L'automate de la section 5.2.1, à partir de l'entrée 110101 possède la fonction de transition $\hat{\delta}$ telle que :

$\hat{\delta}(q_0, \epsilon) = q_0$

$\hat{\delta}(q_0, 1) = \delta(\hat{\delta}(q_0, \epsilon), 1) = \delta(q_0, 1) = q_1$

$\hat{\delta}(q_0, 11) = \delta(\hat{\delta}(q_0, 1), 1) = \delta(q_1, 1) = q_0$

$\hat{\delta}(q_0, 110) = \delta(\hat{\delta}(q_0, 11), 0) = \delta(q_0, 1) = q_2$

$\hat{\delta}(q_0, 1101) = \delta(\hat{\delta}(q_0, 110), 1) = \delta(q_2, 1) = q_3$

$\hat{\delta}(q_0, 11010) = \delta(\hat{\delta}(q_0, 1101), 0) = \delta(q_3, 1) = q_1$

$\hat{\delta}(q_0, 110101) = \delta(\hat{\delta}(q_0, 11010), 1) = \delta(q_1, 1) = q_0$

Définition 5.3 (Langage engendré par un automate - 2) *Le langage d'un automate à états finis déterministe* $A = (Q, \Sigma, \delta, q_0, F)$ *est défini par :*
$L(A) = \{w \mid \hat{\delta}(q_0, w) \in F\}$

Théorème 5.1 *Pour tout automate à états finis déterministe A, il existe une expression régulière R, telle que* $L(A) = L(R)$

5.3 Automate fini non déterministe - AFN

5.3.1 Exemple

Un automate non déterministe possède un nombre fini d'états, un ensemble fini de symboles d'entrée, un état de départ et un ensemble d'états finals. Il possède une fonction de transition, mais la différence avec un automate déterministe est que cette fonction renvoie un ensemble d'états.

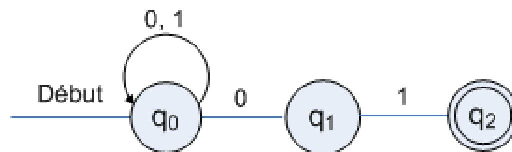


FIGURE 5.3 – Exemple d'automate non déterministe

La figure 5.3 représente un automate non déterministe dont le travail est de n'accepter que les chaînes composées de 0 et de 1 et terminées par la séquence 01. Il reste en l'état q_0 tant qu'il n'a pas deviné que l'ultime séquence n'a pas commencé.

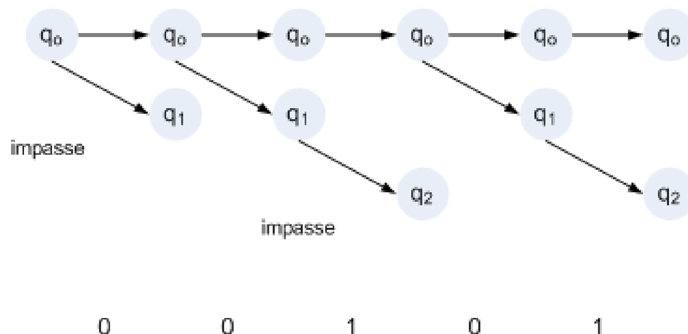


FIGURE 5.4 – Reconnaissance de la chaîne 00101

5.3.2 Définition

Définition 5.4 (Automate non déterministe) *Un automate non déterministe est un quintuplet* $A = (Q, \Sigma, \delta, q_0, F)$ *où :*

- Q *est un ensemble fini d'états.*
- Σ *est un ensemble fini de symboles d'entrée.*
- q_0 *l'état initial* ($q_0 \in Q$).
- *F, sous-ensemble de Q, est l'ensemble des états finals.*
- δ*, la fonction de transition, qui prend en argument un état et un symbole d'entrée et qui retourne un ensemble d'états.* $\delta : Q \times \Sigma \longrightarrow \mathcal{P}(Q)$

La différence entre un AFD et un AFN tient dans la nature de la fonction de transition. L'AFN représenté par la figure 5.3 est spécifié formellement par :

$$\text{A} = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$$

Sa table de transition est la suivante :

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

5.3.3 Extension de la fonction de transition

La fonction de transition δ est étendue à une fonction $\hat{\delta}$ par :

- $\hat{\delta}(q, \epsilon) = \{q\}$

- Pour tout $w \neq \epsilon$, $w = xa$ où a est le symbole final de w et x le reste de w.
  En supposant que

– $\hat{\delta}(q, x) = \{p_1, p_2, \ldots, p_k\}$
– $\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, \ldots, r_m\}$

alors $\hat{\delta}(q, w) = \{r_1, r_2, \ldots, r_m\}$

5.3.4 Exemple

L'automate de la section 5.3.1, à partir de la l'entrée 00101 donne la fonction $\hat{\delta}$ telle que :
$\hat{\delta}(q_0, \epsilon) = \{q_0\}$
$\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$
$\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
$\hat{\delta}(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$
$\hat{\delta}(q_0, 0010) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
$\hat{\delta}(q_0, 00101) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

Définition 5.5 (Langage d'un AFN) *Soit $A = (Q, \Sigma, \delta, q_0, F)$ un AFN, alors le langage engendré par l'automate A est défini par L(A) = $\{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$*

5.4 Automate fini avec ϵ-transitions (ϵ-AFN)

5.4.1 introduction

Les automates finis possédant des ϵ-transitions sont des automates pouvant faire spontanément des transitions. Les arcs correspondants sont étiquetés par le symbole ϵ et ne consomment aucun caractère de la chaîne d'entrée.

5.4.2 Définition

La définition d'un automate avec ϵ-transitions est similaire à celle d'un automate non déterministe. La fonction de transition inclut des informations sur les transitions concernant ϵ.

Définition 5.6 (Automate avec ϵ-transitions) *Un automate non déterministe avec ϵ-transitions est un quintuplet $A = (Q, \Sigma, \delta, q_0, F)$ où :*
- *Q est un ensemble fini d'états*
- *Σ est un ensemble finis de symboles d'entrée*
- *q_0 l'état initial ($q_0 \in Q$)*
- *F, sous-ensemble de Q, est l'ensemble des états finals*
- *δ, la fonction de transition, qui prend en argument un état et un élément de $\Sigma \cup \{\epsilon\}$ et qui retourne un ensemble d'états*

L'ϵ-AFN de la figure 5.5 est représenté de façon formelle par :

$$\text{A} = (\{q_0, q_1, q_2, q_3, q_4, q_5\}, \{., +, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \delta, q_0, \{q_5\})$$

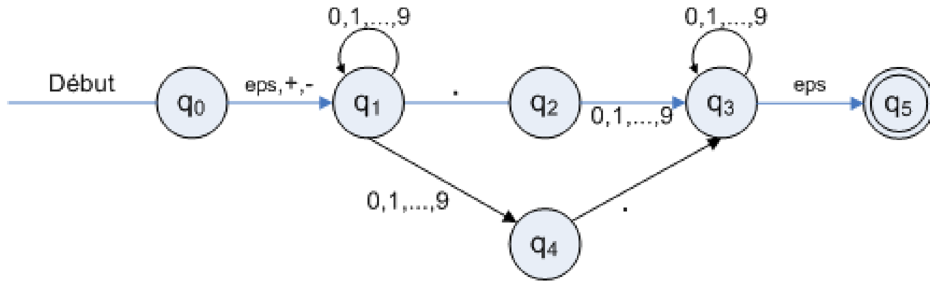


FIGURE 5.5 – Exemple d'automate fini non déterministe avec ϵ-transitions

Sa table de transition est la suivante :

	ϵ	+, -	.	0, 1, ..., 9
$\rightarrow q_0$	$\{q_1\}$	$\{q_1\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\emptyset$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\{q_5\}$	$\emptyset$	$\emptyset$	$\{q_3\}$
q_4	$\emptyset$	$\emptyset$	$\{q_3\}$	$\emptyset$
$*\ q_5$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

$F = \{q_5\}$

5.5 Automates finis et expressions régulières

Les automates finis et les expressions régulières, bien que d'aspect totalement différents permettent de représenter exactement la même catégorie de langages.

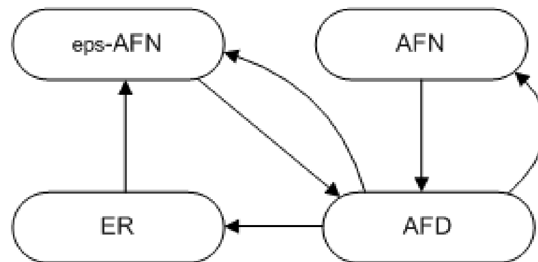


FIGURE 5.6 – Equivalence des quatre notions

5.5.1 Passage d'un AFD à une expression régulière

Théorème 5.2 *Soit L(A) le langage défini par un AFD, A. Il existe une expression régulière R, telle L(R) = L(A).*

Démonstration

Soit A un AFD ayant n états. Soit {1, 2, ..., n} l'ensemble de ses états.
Notons $R_{ij}^{(k)}$ le nom de l'expression régulière dont le langage est l'ensemble des chaînes w

construites sur un chemin allant de l'état i à l'état j qui ne passe pas par un état dont le rang est supérieur à k.
Lorsque k = 0 il n'y a que deux possibilités :
- si $i = j$ on ne considère que le nœud i lui-même ; les chemins légaux sont le chemin de longueur 0 étiqueté ϵ et les boucles sur l'état i.
  - $R_{ij}^{(0)} = \epsilon$ lorsqu'il n'y a pas de transition
  - $R_{ij}^{(0)} = \epsilon \mid a$ lorsqu'il n'y a qu'une transition étiquetée a
  - $R_{ij}^{(0)} = \epsilon \mid a_1 \mid a_2 \mid \ldots \mid a_p$ lorsqu'il y a p transitions
- si $i \neq j$ on ne considère que les arcs reliant le nœud i au nœud j.
  - $R_{ij}^{(0)} = \emptyset$ lorsqu'il n'y a pas de transition
  - $R_{ij}^{(0)} = a$ lorsqu'il n'y a qu'une transition étiquetée a
  - $R_{ij}^{(0)} = a_1 \mid a_2 \mid \ldots \mid a_p$ lorsqu'il y a p transitions étiquetées de a_1 à a_p

Récurrence : Lorsque k $\neq$ 0
On considère les chemins allant du nœud i au nœud j passant par des nœuds dont l'indice est $\leq$ k.
Un chemin peut ne pas passer par le nœud k et dans ce cas son étiquette est décrite par $R_{ij}^{(k)} = R_{ij}^{(k-1)}$
Un chemin peut passer par le nœud k au moins une fois et repasser éventuellement par ce même nœud puis terminer sur le nœud j. Dans ce cas son étiquette satisfait l'expression régulière :

$$R_{ik}^{(k-1)}(R_{kk}^{(k-1)}) * R_{kj}^{(k-1)}$$

Par suite :
$R_{ij}^{(k)} = R_{ij}^{(k-1)} \mid R_{ik}^{(k-1)}(R_{kk}^{(k-1)}) * R_{kj}^{(k-1)}$
$R_{ij}^{(k)}$ ne dépend que d'expressions de degré inférieur à k et l'expression régulière décrivant un AFD est donnée par :

$R_{1p_1}^{(n)} \mid R_{1p_2}^{(n)} \mid \ldots \mid R_{1p_m}^{(n)}$ où les nœuds $p_1, p_2, \ldots, p_m$ sont les nœuds finals.

Illustration

L'automate de la figure 5.7 correspond à la partie droite de l'automate de la figure 5.1 où q_2 et q_1 ont pour noms 1 et 2.

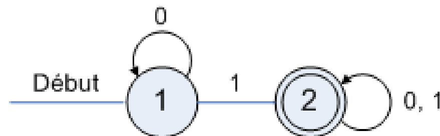


FIGURE 5.7 – Automate simple

D'après les définitons précédentes :

$R_{11}^{(0)}$	$\epsilon \mid 0$
$R_{12}^{(0)}$	1
$R_{21}^{(0)}$	$\emptyset$
$R_{22}^{(0)}$	$\epsilon \mid 0 \mid 1$

De manière générale, d'après l'hypothèse de récurrence : $R_{ij}^{(1)} = R_{ij}^{(0)} \mid R_{i1}^{(0)}(R_{11}^{(0)}) * R_{1j}^{(0)}$
On obtient :

	Par substitution directe	Simplification
$R_{11}^{(1)}$	$\epsilon \mid 0 \mid (\epsilon \mid 0)(\epsilon \mid 0)^*(\epsilon \mid 0)$	0^*
$R_{12}^{(1)}$	$1 \mid (\epsilon \mid 0)(\epsilon \mid 0)^*1$	0^*1
$R_{21}^{(1)}$	$\emptyset \mid \emptyset(\epsilon \mid 0)^*(\epsilon \mid 0)$	$\emptyset$
$R_{22}^{(1)}$	$\epsilon \mid 0 \mid 1 \mid \emptyset(\epsilon \mid 0)^*1$	$\epsilon \mid 0 \mid 1$

Puis : $R_{ij}^{(2)} = R_{ij}^{(1)} \mid R_{i2}^{(1)}(R_{22}^{(1)}) * R_{2j}^{(1)}$

	Par substitution directe	Simplification
$R_{11}^{(2)}$	$0^* \mid 0^*1(\epsilon \mid 0 \mid 1)^*$	0^*
$R_{12}^{(2)}$	$0^*1 \mid 0^*1(\epsilon \mid 0 \mid 1)^*(\epsilon \mid 0 \mid 1)$	$0^*1(0 \mid 1)^*$
$R_{21}^{(2)}$	$\emptyset \mid (\epsilon \mid 0 \mid 1)(\epsilon \mid 0 \mid 1)^* \ \emptyset$	$\emptyset$
$R_{22}^{(2)}$	$\epsilon \mid 0 \mid 1 \mid (\epsilon \mid 0 \mid 1)(\epsilon \mid 0 \mid 1)^*$	
	$(\epsilon \mid 0 \mid 1)$	$(0 \mid 1)^*$

L'expression $R_{12}^{(2)}$ donne le résultat. Reprenons l'automate de la figure 5.1 et rebaptisons respectivement q_0, q_2 et q_1 en 3, 1 et 2. l'expression régulière à déterminer est : $R_{32}^{(3)}$
On démontre aisément que $R_{13}^{(3)} = R_{13}^{(2)} = R_{13}^{(1)} = R_{13}^{(0)} = \emptyset$
Par suite $R_{12}^{(3)} = R_{12}^{(2)} = 0^*1(0 \mid 1)^*$ et $R_{32}^{(3)} = 1^*00^*1(0 \mid 1)^*$

5.5.2 Méthode par élimination d'états

La méthode précédente est très pénalisante en nombre d'expressions à créer. Pour n états, il faut considérer n^3 expressions.
Une autre méthode consiste à éliminer des états de l'automate sans changer le langage qu'il définit. Lorsqu'on élimine un état s, tous les chemins allant d'un état q à un état p et passant par s n'existent plus. Comme le langage de l'automate ne doit pas changer, il faut créer un chemin allant de q à p. L'étiquette de celui-ci comporte une chaîne et non plus un seul symbole. On considère ainsi des automates dont les étiquettes des arcs sont des expressions régulières. Les automates déjà rencontrés en sont des cas particuliers puisque tout symbole peut être considéré comme une expression régulière.
Le langage de l'automate est l'union de tous les chemins allant de l'état initial jusqu'à un état d'acceptation. Pour éliminer un état s, il faut considérer tous les chemins allant d'un état q à un état p et passant par s. L'état q peut-être identique à l'état p mais s est distinct des états q et p.
Cas général : l'arc qs est étiqueté par une expression régulière Q ; l'arc sp est étiqueté par

l'expression régulière P et l'état s possède une boucle étiquetée par l'expression régulière S.
Un tel chemin q, s, p sera remplacé par un chemin q, p étiqueté par $QS * P$.

Elimination d'un état

La figure 5.8 montrent un stade auxquel on peut arriver après avoir supprimé tous les états exceptés l'état initial et les états d'acceptation. Dans ce cas l'expression régulière associée à l'automate est : $(R \mid SU * T) *\ SU *$, (R, U, T peuvent être l'expression $\emptyset$).

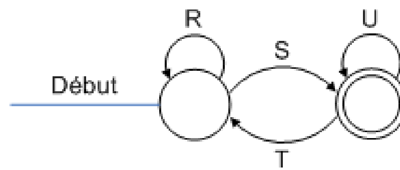


FIGURE 5.8 – Cas général : deux états

La figure 5.9 montre le stade auxquel on arrive lorsque l'état initial est un état d'acceptation. Dans ce cas l'expression régulière associée à l'automate est : $R*$.

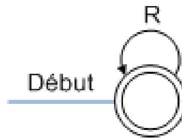


FIGURE 5.9 – Cas général : un état

Dans le cas général, il reste plusieurs chemins issus de l'état intial et s'achevant sur un état d'acceptation différent. L'expression régulière résultante est alors la disjonction des expressions régulières similaires aux précédentes.

La figure 5.10 montre un stade possible lorsque deux états d'acceptation sont situés sur le même chemin. L'expression résultante est la disjonction des expressions régulières obtenues en éliminant un des états d'acceptation. L'expression régulière correspondant à l'automate de la figure 5.10 est : $R * SU * T \mid R * SU *$.

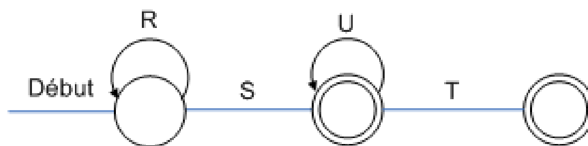


FIGURE 5.10 – Deux états d'acceptation

5.5.3 Passage d'un AFN à un AFD

Algorithme

1. Soit $E = \{q_0\}$.
2. Construire $E^{(1)}(a) = \delta(E, a) = \bigcup_{q \in E} \delta(q, a)$ pour tout $a \in \Sigma$.
3. Recommencer 2 pour toute transition et pour chaque nouvel ensemble $E^{(i)}(a)$.
4. Tous les ensembles d'états $E^{(i)}(a)$ contenant au moins un état final deviennent des états finals.
5. Renuméroter les ensembles d'états en tant que simples états.

Exemple

Considérons l'AFN défini par la table de transition suivante :

état	a	b
$\rightarrow q_0$	q_0, q_2	q_1
q_1	q_3	q_0, q_2
$* q_2$	q_3, q_4	q_2
$* q_3$	q_2	q_1
q_4	$\emptyset$	q_3

q_0 état initial ; $F = \{q_2, q_3\}$

Par application de l'algorithme, on obtient le tableau de gauche puis en substituant par un nom unique les ensembles d'états on obtient le tableau de droite, table de transition de l'automate fini déterministe recherché.

état	a	b
q_0	q_0, q_2	q_1
q_0, q_2	q_0, q_2, q_3, q_4	q_1, q_2
q_1	q_3	q_0, q_2
q_0, q_2, q_3, q_4	q_0, q_2, q_3, q_4	q_1, q_2, q_3
q_1, q_2	q_3, q_4	q_0, q_2
q_3	q_2	q_1
q_1, q_2, q_3	q_2, q_3, q_4	q_0, q_1, q_2
q_3, q_4	q_2	q_1, q_3
q_2	q_3, q_4	q_2
q_2, q_3, q_4	q_2, q_3, q_4	q_1, q_2, q_3
q_0, q_1, q_2	q_0, q_2, q_3, q_4	q_0, q_1, q_2
q_1, q_3	q_2, q_3	q_0, q_1, q_2
q_2, q_3	q_2, q_3, q_4	q_1, q_2

=

état	a	b
0	1	2
1	3	4
2	5	1
3	3	6
4	7	1
5	8	2
6	9	10
7	8	11
8	7	8
9	9	6
10	3	10
11	12	10
12	9	4

0 état initial ; $F = \{1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$

5.5.4 Passage d'un ϵ-AFN à un AFD

ϵ-fermeture

Définition 5.7 *L'ϵ-fermeture d'un état q est l'ensemble des états obtenus en suivant des $\epsilon-$transitions à partir de q.*

Règles d'appartenance :
- q $\in$ ϵ-fermeture(q)
- si p $\in$ ϵ-fermeture(q) et s'il existe une ϵ-transition entre p et un état r alors r $\in$ ϵ-fermeture(q).

Définition 5.8 *L'ϵ-fermeture d'un ensemble d'états Q est l'ensemble des états obtenus en suivant des ϵ-transitions à partir de tout état de Q.*

Construction de l'ϵ-fermeture d'un ensemble d'états

```
pour tout e ∈ Q
        empiler e dans la pile P
initialiser ϵ-fermeture(Q) avec Q
tant que P est non vide faire
        dépiler P dans s
        pour chaque état q tel qu'il existe un arc ϵ de s vers q
                si q n'est pas dans ϵ-fermeture(Q) alors
                        ajouter q dans ϵ-fermeture(Q)
                        empiler q dans P
```

Algorithme

- Partir de H, l'ϵ-fermeture de l'état initial.
- Construire $E^{(1)}$, l'ensemble des états obtenus à partir de H par la transition a. Déterminer l'ϵ-fermeture de $E^{(1)}$ et la réunir avec $E^{(1)}$.
  $H^{(1)} = E^{(1)} \cup \epsilon\text{-fermeture}(E^{(1)})$.
- Recommencer l'étape 2 avec les autres transitions et tant qu'il y a de nouveaux états créés.
- Tous les nouveaux états contenant au moins un état final sont des états finals.
- Renommer les états obtenus.

Exemple

Considérons l'ϵ-AFN défini par la table de transition suivante :

état	ϵ	a	b	c
$\rightarrow q_0$	q_1	q_2	$\emptyset$	q_0
q_1	$\emptyset$	q_3	q_4	$\emptyset$
q_2	q_0	$\emptyset$	$\emptyset$	q_1, q_4
q_3	$\emptyset$	$\emptyset$	q_1	$\emptyset$
$*q_4$	q_2	$\emptyset$	$\emptyset$	q_3

q_0 état initial ; $F = \{q_4\}$

- ϵ-fermeture($\{q_0\}$) = $\{q_0, q_1\}$
- ϵ-fermeture($\{q_2, q_3\}$) = $\{q_0, q_1, q_2, q_3\}$
- ϵ-fermeture($\{q_1, q_4\}$) = $\{q_0, q_1, q_2, q_4\}$
- ...

état	a	b	c
q_0, q_1	q_0, q_1, q_2, q_3	q_0, q_1, q_2, q_4	q_0, q_1
q_0, q_1, q_2, q_3	q_0, q_1, q_2, q_3	q_0, q_1, q_2, q_4	q_0, q_1, q_2, q_4
q_0, q_1, q_2, q_4	q_0, q_1, q_2, q_3	q_0, q_1, q_2, q_4	q_0, q_1, q_2, q_3, q_4
q_0, q_1, q_2, q_3, q_4	q_0, q_1, q_2, q_3	q_0, q_1, q_2, q_4	q_0, q_1, q_2, q_3, q_4

En renommant les états, on obtient :

état	a	b	c
0	1	2	0
1	1	2	2
2	1	2	3
3	1	2	3

$F = \{2, 3\}$

5.5.5 Conversion d'une expression régulière en ϵ-AFN

Théorème 5.3 *Tout langage défini par une expression régulière est également défini par un automate fini.*

Si L(R) est le langage dénoté par une expression régulière R, on démontre qu'il existe un ϵ-AFN, A, tel que L(A) = L(R).
La preuve, basée sur les constructions de Thompson, est construite par récurrence à partir des expressions régulières de base. Tous les automates construits sont des ϵ-AFN ayant un unique état d'acceptation, ce qui n'introduit pas de restriction. Il faut combiner les automates au fur et à mesure pour construire des automates composites acceptant les opérations :

- Union (|)
- Concaténation
- Fermeture (*)

La figure 5.11 montre les automates correspondant aux expressions de base.

- a) - L = $\emptyset$
- b) - L = $\{\epsilon\}$
- c) - L = $\{a\}$

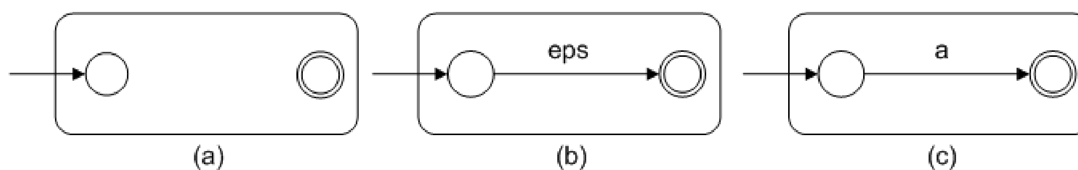


FIGURE 5.11 – $\emptyset$, ϵ, symbole

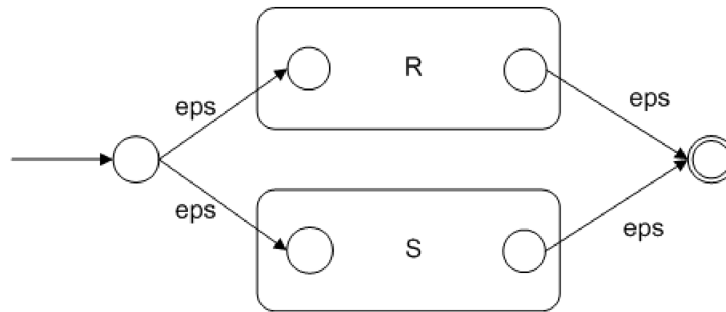


FIGURE 5.12 – R | S

La figure 5.12 montre l'automate construit à partir de la disjonction de deux expressions régilières.

La figure 5.13 montre l'automate construit à partir de la concaténation de deux expressions régulières.

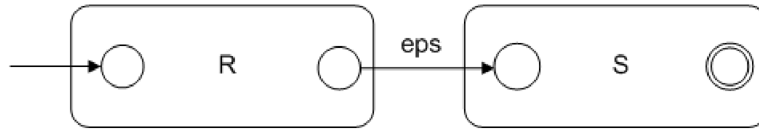


Figure 5.13 – R S

La figure 5.14 montre l'automate construit à partir de la fermeture d'une expression régulière.

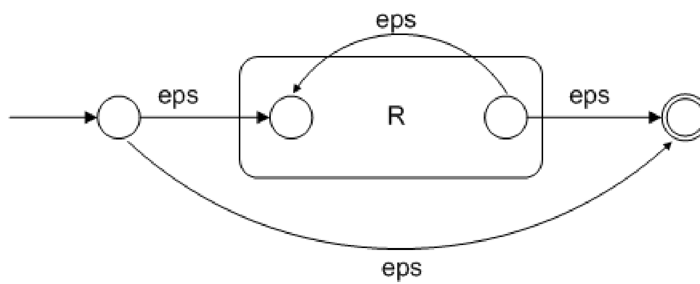


Figure 5.14 – R*

Chapitre 6

Analyse Lexicale

6.1 Rôle d'un l'analyseur lexical

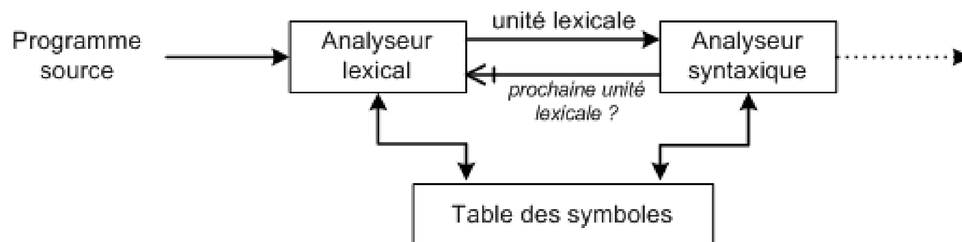


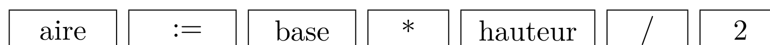
FIGURE 6.1 – Rôle de l'analyseur lexical

6.1.1 Introduction

L'analyse lexicale doit permettre de déterminer correctement les chaînes comprises dans le programme source. Ainsi dans l'instruction Pascal suivante :

$$aire := base * hauteur/2 \tag{6.1}$$

L'analyse lexicale reconnaît les lexèmes suivants :



aire	:=	base	*	hauteur	/	2

6.1.2 Fonctions de l'analyseur lexical

- La tâche principale d'un analyseur lexical est de lire les caractères d'entrée et produire comme résultat une suite d'unités lexicales que l'analyseur syntaxique va utiliser.
- C'est en général un sous-module de l'analyseur syntaxique qui lui commande la prochaine unité lexicale à extraire.
- Autres tâches

- initialiser la table des symboles.
- éliminer les commentaires et les séparateurs.
- relier les erreurs de compilation au programme source ("erreur en ligne ...").

6.1.3 Intérêts de l'analyse lexicale

- Conception modulaire plus simple du compilateur.
- Simplification de l'écriture de l'analyseur syntaxique.
- Efficacité accrue du compilateur.
- Portabilité accrue (modifications de l'alphabet d'entrée).
- Existence de techniques générales d'analyse lexicale.

6.2 Description

6.2.1 Définitions

Définition 6.1 (Lexème) *Un lexème est une chaîne de caractères.*

Définition 6.2 (Unité lexicale) *Une unité lexicale est un type de lexèmes (pour la syntaxe).*

Définition 6.3 (Modèle) *Un modèle est une règle décrivant les chaînes qui correspondent à une unité lexicale.*

Définition 6.4 (Attribut) *Un attribut est une information additionnelle (pour la sémantique).*

Un analyseur lexical est un programme qui reconnaît des unités lexicales. Les expressions régulières permettent d'exprimer les unités lexicales. Donc :

- Un analyseur lexical est un programme qui reconnaît des expressions régulières.
- Pour écrire un analyseur lexical, il suffit d'écrire un programme simulant un AFD (automate fini déterministe).

6.2.2 Exemple

Après l'analyse lexicale de l'expression aire := base * hauteur / 2
La chaîne source est divisée en tokens lexicaux ou lexèmes grâce à un analyseur lexical.

lexèmes	unité lexicale	Attribut
aire	identificateur	pointeur vers la table des symboles
:=	op-affectation	aucun
base	identificateur	pointeur vers la table des symboles
*	op-arithmétique	code
<	op-relation	code

En pratique les unités lexicales n'ont qu'un seul attribut, un pointeur vers la table des symboles dans laquelle l'information sur l'unité lexicale est conservée. Les définitions régulières qui en permettent l'analyse pourraient être :
lettre $\longrightarrow$ [A-Za-z]
chiffre $\longrightarrow$ [0-9]
chiffres $\longrightarrow$ chiffre (chiffre)*
identificateur $\longrightarrow$ lettre (lettre | chiffre)*
fraction $\longrightarrow$. chiffres | ϵ
exposant $\longrightarrow$ E(+ | - | ϵ) chiffres | ϵ
nombre $\longrightarrow$ chiffres fraction exposant
op-affectation $\longrightarrow$:=
op-relation $\longrightarrow$ = | < | > | <> | $\leq$ | $\geq$
op-arithmétique $\longrightarrow$ * | / | + | -

6.3 Constructions de l'analyseur lexical

6.3.1 Principe

La figure 6.2 représente la démarche d'analyseur lexical.

- Les unités lexicales sont définies par des expressions régulières.
- A chaque unité lexicale u_1, u_2, $\ldots u_n$ est associé un AFN.
- Un AFN général est bâti à partir de ces AFN.

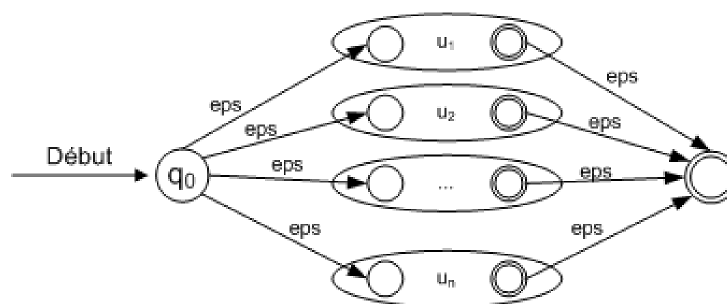


FIGURE 6.2 – ϵ-AFN pour un analyseur lexical

6.3.2 Ecœuils

Prévision

Les analyseurs lexicaux sont obligés pour reconnaître certaines constructions de langages de programmation, de lire des caractères en avant au-delà de la fin d'un lexème avant de déterminer avec certitude l'unité lexicale correspondante.

Ambiguïté

- Plusieurs modèles peuvent représenter le même lexème :

  . Le prochain token le plus long est celui qui vérifie une expression régulière.
  . Le premier modèle (la première expression régulière) qui est vérifié détermine l'unité lexicale du token (Il y a un ordre dans l'écriture des définitions régulières).
- Les mots clés sont des mots réservés et ne peuvent servir d'identificateurs :
  . Comme les mots-clés sont des suites de lettres, ils sont des exceptions au modèle "identificateur". Plutôt que de définir un modèle particulier, une astuce consiste à traiter les mots-clés comme des identificateurs particuliers. On les recherche à l'aide de l'exécution d'un code particulier lorsque l'automate se trouve en état d'acceptation.
- Exemples :
  . if8 est un identificateur (règle du plus long).
  . if 8 est une erreur (une parenthèse est en général requise après if).
  . if est un mot réservé.

6.3.3 Compromis : temps d'exécution - place occupée

Problème : Pour une expression régulière r, déterminer si une chaîne $x \in L(r)$.
- A l'aide d'un AFN
  . Temps de construction : $\mathcal{O}(|r|)$
  . Nombre d'états $\leq |r|$.
  . Au plus deux transitions par état.
  . Taille de la table : $\mathcal{O}(|r|)$
  . Temps d'analyse $\mathcal{O}(|r|$ x $|x|)$.
  . Plutôt utilisé lorsque les chaînes x ne sont pas très longues
- A l'aide d'un AFD
  . On construit d'abord un AFN à partir de la construction de Thompson, puis on le transforme en AFD.
  . Grâce à certains algorithmes, on peut simuler l'AFD résultant sur une chaîne x en un temps proportionnel à $|x|$ indépendamment du nombre d'états. Place occupée : $\mathcal{O}(2^{|r|})$
  . Cette approche est avantageuse lorsque les chaînes d'entrée sont longues.
- Evaluation paresseuse : ne calculer de l'AFD que la partie effectivement utilisée par la chaîne à tester.
  . Bon compromis général.

Chapitre 7

Grammaires hors contexte

7.1 Introduction

7.1.1 Existence de langages non réguliers

Théorème 7.1 *Il existe des langages non réguliers.*

Considérons le langage $L_{01} = \{0^n 1^n \mid n \geq 1\}$. Il consiste en un nombre quelconque de 0 suivi du même nombre de 1. Ce langage n'est pas régulier.

Démonstration rapide

Si ce langage est régulier alors il existe un AFD le dénotant. Cet AFD a un nombre fini d'états (k par exemple). Considérons l'AFD lisant une chaîne commençant par des 0. Il est possible de trouver deux nombres i et j différents tels que, après avoir lu les préfixes 0^i et 0^j, l'AFD soit dans le même état (sinon l'AFD aurait un nombre infini d'états). Considérons qu'à partir de ce moment, l'AFD lise des 1. S'il existait un état final, l'AFD l'attendrait après avoir lu p caractères 1.
Or p ne pourra être simultanément égal à i et à j.

7.1.2 Exemple : fragment de grammaire

S $\longrightarrow \epsilon$
S $\longrightarrow 0$
S $\longrightarrow 1$
S $\longrightarrow 0S0$
S $\longrightarrow 1S1$

Cette grammaire permet de définir le langage des palindromes sur $\{0, 1\}$

7.2 Définition formelle

Définition 7.1 (Grammaire hors contexte - CFG) *Une grammaire hors contexte (en anglais Context Free Grammar) est définie par 4 éléments :*

- *T : un alphabet, ensemble fini non vide de symboles permettant de composer les chaînes du langage. Cet alphabet est appelé ensemble des symboles terminaux ou terminaux.*
- *V : Un ensemble fini non vide de variables, appelées non terminaux ou catégories syntaxiques. $T \cap V = \emptyset$.*
- *S : une variable particulière appelée symbole de départ*
- *P : un ensemble fini non vide de règles de production*
  . *de forme $A \longrightarrow \alpha$*
  . *où A est une variable et α une chaîne de 0 ou plus terminaux et variables.*
  . *Les parties gauche et droite de la flèche sont appelées les tête et corps de la production.*
  . *$P \subset \{(A, \alpha) \mid A \in V; \alpha \in (V \cup T)*\}$*

Une grammaire G est donc un quadruplet $G = (T, V, S, P)$

7.2.1 Conventions d'écriture

Minuscules de début d'alphabet (a, b, c)	symboles terminaux
Majuscules de début d'alphabet (A, B, C)	variables
Minuscules de fin d'alphabet (w, x, z)	chaînes de terminaux
Majuscules de fin d'alphabet (W, X, Z)	terminaux ou variables
Lettres grecques (α, β)	chaînes de terminaux et de variables

7.3 Exemples

7.3.1 Grammaire des palindromes sur $\{0, 1\}$

T = $\{0, 1\}$
V = $\{S\}$
S : symbole de départ
P = {
  S $\longrightarrow \epsilon$
  S $\longrightarrow$ 0
  S $\longrightarrow$ 1
  S $\longrightarrow 0S0$
  S $\longrightarrow 1S1$
}

7.3.2 Petit sous-ensemble de la grammaire française

T = { d_m, d_f, nom_m, nom_f, vb }
V = { PH, GN, N, V, D }
S = PH
P = {
  PH $\longrightarrow$ GN V GN
  GN $\longrightarrow$ D N
  N $\longrightarrow$ nom_m

N ⟶ nom_f
D ⟶ d_m
D ⟶ d_f
V ⟶ vb
}
Les symboles terminaux sont des identifiants d'unités lexicales définies par exemple par :

d_m	⟶	"le"
d_f	⟶	"la"
nom_m	⟶	"chat"
nom_f	⟶	"balle"
vb	⟶	"regarde"

Quelques chaînes syntaxiquement correctes

1. le chat regarde la balle (sémantiquement correct)
2. la balle regarde le chat (sémantiquement incorrect)
3. le chat regarde le balle (accord incorrect)

Utilisation des règles

PH ⟹ GN V GN ⟹ D N V GN
⟹ d_m N V GN ⟹ d_m nom_m V GN
⟹ d_m nom_m vb GN ⟹ d_m nom_m vb D N
⟹ d_m nom_m vb d_f N ⟹ d_m nom_m vb d_f nom_f

Et après substitution par les lexèmes correspondants :
"le chat regarde la balle"

7.4 Dérivation

7.4.1 Extension des règles de production

Pour définir le langage engendré par une grammaire, on définit chacune des chaînes ainsi :
- Choisir une règle dont la tête est le symbole de départ
- Remplacer une des variables du corps de la chaîne produite par le corps de l'une de ses productions
- Continuer jusqu'à ce que le corps ne contiennent plus que des symboles terminaux

Soient $G = (T, V, S, P)$ une grammaire et $\alpha A \beta$ une chaîne telle que :
. $\alpha \in (V \cup T)^*$,
. $\beta \in (V \cup T)^*$
. $A \in V$
. et $A \longrightarrow \gamma$ une production de G

alors :
. $\alpha A \beta$ se dérive selon G en $\alpha\gamma\beta$

et on écrit :
. $\alpha A \beta \underset{G}{\Rightarrow} \alpha\gamma\beta$ ou plus simplement $\alpha A \beta \Rightarrow \alpha\gamma\beta$

Il est possible d'étendre la relation $\Longrightarrow$ en $\overset{*}{\Longrightarrow}$ pour représenter 0, 1 ou plusieurs pas de dérivation.
Pour tout $\alpha \in (V \cup T)^*$ on écrit

. $\alpha \overset{*}{\Longrightarrow} \alpha$

. Si $\alpha \overset{*}{\Longrightarrow} \beta$ et $\beta \Longrightarrow \gamma$ alors $\alpha \overset{*}{\Longrightarrow} \gamma$

Définition 7.2 (Langage engendré par une grammaire) *Le langage défini par la grammaire $G = (T, V, S, P)$ appelé $L(G)$ est l'ensemble des chaînes dérivées du symbole de départ :*
$L(G) = \{w \in T^* \mid S \underset{G}{\overset{*}{\Longrightarrow}} w \}$

Remarque importante : Une grammaire définit un seul langage. Par contre, un même langage peut être engendré par plusieurs grammaires différentes.

Définition 7.3 (Syntagme) *Etant donnée une grammaire $G = (T, V, S, P)$, un syntagme α est une chaîne de $(T \cup V)*$ telle que $S \overset{*}{\Longrightarrow} \alpha$.*

Définition 7.4 (Dérivation terminale) *Une dérivation terminale w est un syntagme appartenant à $T*$: $S \overset{*}{\Longrightarrow} w$ (un programme correct).*

7.4.2 Dérivation la plus à gauche, la plus à droite

Etant donné que plusieurs variables peuvent apparaître dans le corps d'une règle de production, il est possible d'imposer le choix de la variable à remplacer. On peut ainsi imposer celle figurant le plus à gauche (respectivement le plus à droite). La dérivation de la section 7.3.2 est une dérivation la plus à gauche.
Soit la grammaire suivante :
Symboles terminaux : $\Sigma = \{a, b, 0, 1, +, *, (,)\}$. Variables : $V = \{E, I\}$. Symbole de départ E. Règles de production P, telles que :

$E \longrightarrow I$
$E \longrightarrow E + E$
$E \longrightarrow E * E$
$E \longrightarrow (E)$
$I \longrightarrow a$
$I \longrightarrow b$
$I \longrightarrow Ia$
$I \longrightarrow Ib$
$I \longrightarrow I0$
$I \longrightarrow I1$

Dérivation la plus à gauche de : $a * (b + a0)$

$$E \underset{lm}{\Longrightarrow} E * E \underset{lm}{\Longrightarrow} I * E \underset{lm}{\Longrightarrow} a * E \underset{lm}{\Longrightarrow} a * (E) \underset{lm}{\Longrightarrow} a * (E + E) \underset{lm}{\Longrightarrow} a * (I + E)$$
$$\underset{lm}{\Longrightarrow} a * (b + E) \underset{lm}{\Longrightarrow} a * (b + I) \underset{lm}{\Longrightarrow} a * (b + I0) \underset{lm}{\Longrightarrow} a * (b + a0)$$

Dérivation la plus à droite de : $a * (b + a0)$

$$E \underset{rm}{\Longrightarrow} E * E \underset{rm}{\Longrightarrow} E * (E) \underset{rm}{\Longrightarrow} E * (E + E) \underset{rm}{\Longrightarrow} E * (E + I) \underset{rm}{\Longrightarrow} E * (E + I0) \underset{rm}{\Longrightarrow} E * (E + a0)$$
$$\underset{rm}{\Longrightarrow} E * (I + a0) \underset{rm}{\Longrightarrow} E * (b + a0) \underset{rm}{\Longrightarrow} I * (b + a0) \underset{rm}{\Longrightarrow} a * (b + a0)$$

7.4.3 Arbre de dérivation

Pour déterminer si une chaîne terminale appartient au langage engendré par une grammaire, on établit un arbre de dérivation dont la racine est l'axiome, les feuilles sont des terminaux formant la chaîne donnée et les nœuds sont des variables décrivant les règles utilisées.
Soit la grammaire suivante :
Symboles terminaux : $\Sigma = \{a, b, 0, 1, +, *, (,)\}$. Variables : $V = \{E, I\}$. Symbole de départ E. Règles de production P, telles que :

$E \longrightarrow I$
$E \longrightarrow E + E$
$E \longrightarrow E * E$
$E \longrightarrow (E)$
$I \longrightarrow a$
$I \longrightarrow b$
$I \longrightarrow Ia$
$I \longrightarrow Ib$
$I \longrightarrow I0$
$I \longrightarrow I1$

La figure 7.1 représente l'arbre de dérivation de l'expression : $a*(a+b00)$. La dérivation la plus à gauche et la plus à droite fournissent le même arbre.

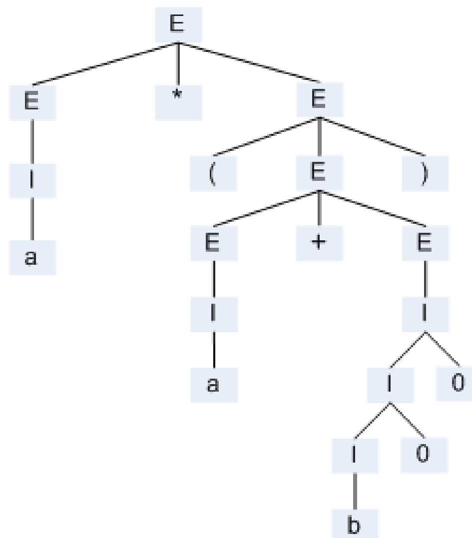


FIGURE 7.1 – Arbre de dérivation

Définition 7.5 (Arbre de dérivation) *On appelle arbre de dérivation (ou arbre syntaxique), tout arbre tel que :*
. la racine est le symbole de départ.
. les feuilles sont des symboles terminaux ou ϵ.
. les nœuds sont des non-terminaux.
. les fils d'un nœud X sont $Y_0, Y_1, \ldots, Y_n$ si et seulement si $X \longrightarrow Y_0 Y_1 \ldots Y_n$ avec les $Y_i \in T \cup V$.

7.4.4 Ambiguïté

Définition 7.6 (Ambiguïté) *Une grammaire G est ambiguë s'il existe une chaîne du langage $L(G)$ qui possède plusieurs arbres de dérivation.*

Exemple de grammaire ambiguë

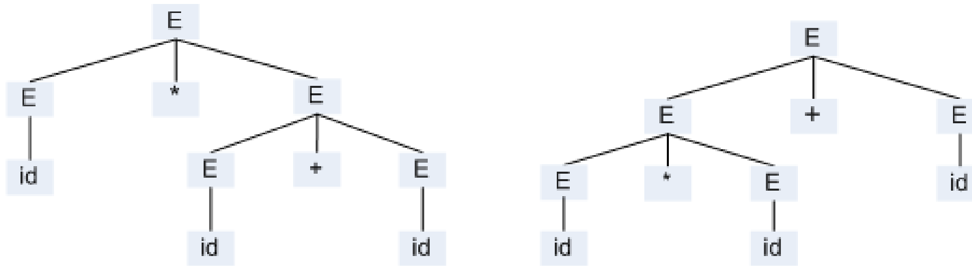


Figure 7.2 – Ambiguïté de la dérivation

La grammaire suivante :
Symboles terminaux : $\Sigma = \{id, +, *, (,)\}$. Variables : $V = \{E\}$. Symbole de départ E. Règles de production P, telles que :

$E \longrightarrow E + E$
$E \longrightarrow E * E$
$E \longrightarrow (E) \mid id$

est ambiguë. La figure 7.2 montre les deux arbres de dérivation possibles de l'expression : $id * id + id$.

$$E \Rightarrow E * E \Rightarrow E * E + E \Rightarrow id * E + E \Rightarrow id * id + E \Rightarrow id * id + id$$

$$E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow id * E + E \Rightarrow id * id + E \Rightarrow id * id + id$$

7.5 Forme de BACKUS-NAUR

La notation de Backus-Naur (en anglais Backus-Naur Form, ou BNF) a été utilisée dès 1960 pour décrire le langage ALGOL 60, et depuis est employée pour définir de nombreux langages de programmation. L'écriture BNF des règles de grammaire est définie comme suit :

Le symbole des règles de réécriture est remplacé par le symbole “ : :=”.
Les symboles désignant les éléments non-terminaux sont inclus entre un chevron ouvrant “<” et un chevron fermant “>”, ceci afin de les distinguer des terminaux.
Un ensemble de règles dont les parties gauches sont identiques, telles que :

A : := α_1
A : := α_2
A : := ...
A : := α_p

peut être écrit de manière abrégée : A : := $\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_p$
Certains auteurs proposent des extensions (forme EBNF) :
L’opérateur ? (0 ou 1 fois)
Les items optionnels sont placés entre les méta symboles “[“ et “]”.
Exemple :

```
<if_statement> ::=  if <boolean_expression> then
                         <statement_sequence>
                    [ else
                         <statement_sequence> ]
                    end if ;
```

L’opérateur * (0 ou plusieurs fois)
Les items à répéter sont placés entre les méta symboles “{” et “}”.
Exemple :

```
<identifier> ::= <letter> { <letter> | <digit> }
```

7.6 Types de grammaires

En introduisant des critères plus ou moins restrictifs sur les règles de production, on obtient des classes de grammaires hiérarchisées, ordonnées par inclusion. La classification des grammaires, définie en 1957 par Noam CHOMSKY [1], distingue les quatre classes suivantes :

Type 0

Pas de restriction sur les règles. Elles sont de la forme ;
$\alpha \longrightarrow \beta$ avec $\alpha \in (V \cup T)^+$ et $\beta \in (V \cup T)^*$
Toute sous-chaîne de α peut être remplacée par une chaîne quelconque.

Type 1

Grammaires sensibles au contexte ou contextuelles (CSG). Les règles sont de la forme :
$\alpha A \beta \longrightarrow \alpha\gamma\beta$ avec $A \in V$; $\alpha, \beta \in (V \cup T)^*$ et $\gamma \in (V \cup T)^+$.
Autrement dit, le symbole non terminal A est remplacé par la forme γ de $(V \cup T)^+$ si les contextes α et β sont présents à gauche et à droite.

1. Bibliographie : http ://web.mit.edu/linguistics/www/biography/noambio.html

Type 2

Grammaires hors-contexte (CFG). Les règles sont de la forme :
$A \longrightarrow \alpha$ avec $A \in V$ et $\alpha \in (V \cup T)^*$.
Autrement dit, le membre de gauche de chaque règle est constitué d'un seul symbole non terminal.

Type 3

Grammaires régulières à droite (respectivement à gauche). Les règles sont de la forme :
$A \longrightarrow aB$, (respectivement $A \longrightarrow Ba$) avec $A, B \in V$ et $a \in T$.
ou de la forme : $A \longrightarrow a$ avec $A \in V$ et $a \in T$. Autrement dit, le membre de gauche de chaque règle est constitué d'un seul symbole non terminal, et le membre de droite est constitué d'un symbole terminal éventuellement suivi (respectivement précédé) d'un seul non terminal.

7.7 Expression régulière sous forme de grammaire

Théorème 7.2 *Pour toute expression régulière R, il existe une grammaire G telle que L(G) = L(R).*

La démonstration se fait par récurrence sur le nombre n d'opérateurs de l'expression régulière. Soit R une expression régulière sur un alphabet Σ.

- n = 0
  R = ∅ ou R = ϵ ou R = x où $x \in \Sigma$
  On crée un symbole S et une grammaire $G = (T, V, S, P)$ ayant au plus une règle de dérivation : T = Σ, V = {S}
  - R = ∅, pas de règle, L(G) = ∅
  - R = ϵ, 1 règle, $S \longrightarrow \epsilon$, L(G) = {$\epsilon$}
  - R = x, 1 règle, $S \longrightarrow x$, L(G) = {x}
- n ≥ 0
  On suppose l'hypothèse de récurrence vraie pour des expressions comportant au plus n occurrences d'opérateurs. Soit R une expressions ayant n+1 occurrences d'opérateurs.
  $R = R_1|R_2$ ou $R = R_1R_2$ ou $R = R_1*$ où R_1 et R_2 possèdent au plus n occurrences d'opérateurs.
  Selon l'hypothèse de récurrence :
  il existe $G_1 = (T, V_1, S_1, P_1)$ et $G_2 = (T, V_2, S_2, P_2)$ telles que $L(R_1) = L(G_1)$ et $L(R_2) = L(G_2)$ (si G_1 et G_2 n'ont pas le même ensemble de terminaux, T est pris comme la réunion de ces ensembles).
  Il est possible de construire G_1 et G_2 telles que $V_1 \cap V_2 = \emptyset$, en renommant éventuellement quelques variables.
  - $R = R_1|R_2$
    Soit $G = (T, V, S, P)$ telle que : $V = V_1 \cup V_2 \cup \{S\}$
    $P = P_1 \cup P_2 \cup P'$ où P' contient la règle : $S \longrightarrow S_1|S_2$
    $L(G) = L(R_1) \cup L(R_2)$

- $R = R_1R_2$
  Soit $G = (T, V, S, P)$ telle que : $V = V_1 \cup V_2 \cup \{S\}$
  $P = P_1 \cup P_2 \cup P'$ où P' contient la règle : $S \longrightarrow S_1S_2$
  $L(G) = L(R_1)L(R_2)$
- $R = R_1*$
  Soit $G = (T, V, S, P)$ telle que : $V = V_1 \cup \{S\}$
  $P = P_1 \cup P'$ où P' contient la règle : $S \longrightarrow S_1S|\epsilon$
  $L(G) = L(R_1)*$

7.8 Equivalence

Théorème 7.3 *Etant données une grammaire $G = (T, V, S, P)$, une variable A et une chaîne de terminaux w, les quatre propositions suivantes sont équivalentes :*
. $A \overset{*}{\Longrightarrow} w$
. $A \underset{lm}{\overset{*}{\Longrightarrow}} w$
. $A \underset{rm}{\overset{*}{\Longrightarrow}} w$
. *Il existe un arbre de dérivation de racine A et qui produit w*

Remarque : le théorème est également vrai si w est remplacée par une chaîne α de $T \cup V$

7.9 Types d'analyse

Il existe deux grandes classes de techniques différentes pour construire un arbre syntaxique et vérifier si une chaîne appartient au langage. L'analyse de la chaîne se fait toujours de la gauche vers la droite (Left scanning).
- L'approche descendante. Un analyseur gauche construit l'arbre de dérivation à partir de sa racine et en effectuant des dérivations en considérant la tête des règles de production et en faisant des dérivations les plus à gauche. C'est la famille des analyseurs LL (left scanning, leftmost derivation).
- L'approche ascendante. Un analyseur droit construit l'arbre de dérivation à partir de ses feuilles et en effectuant des dérivations en considérant la partie droite des règles de production et en faisant des dérivations les plus à droite. C'est la famille des analyseurs LR (left scanning, rightmost derivation).

7.9.1 Exemple d'analyse

Soit la grammaire G suivante :
$T = \{a, c, d\}$ $V = \{S, T\}$
$P = \{$

1. $S \longrightarrow aSbT$
2. $S \longrightarrow cT$
3. $S \longrightarrow d$
4. $T \longrightarrow aT$

5. $T \longrightarrow bS$
6. $T \longrightarrow c$

}

Considérons la chaîne : $w = accbbadbc$.

7.9.2 Analyse LL

La dérivation permettant de construire l'arbre est la suivante :
$S \underset{1}{\Longrightarrow} aSbT \underset{2}{\Longrightarrow} acTbT \underset{6}{\Longrightarrow} accbT \underset{5}{\Longrightarrow} accbbS \underset{1}{\Longrightarrow} accbbaSbT \underset{3}{\Longrightarrow} accbbadbT \underset{6}{\Longrightarrow} accbbadbc$

7.9.3 Analyse LR

Pour le même exemple une analyse ascendante donnerait :
$accbbadbc \underset{6}{\Longleftarrow} acTbbadbc \underset{2}{\Longleftarrow} aSbbadbc \underset{3}{\Longleftarrow} aSbbaSbc \underset{6}{\Longleftarrow} aSbbaSbT \underset{1}{\Longleftarrow} aSbbS$
$\underset{5}{\Longleftarrow} aSbT \underset{1}{\Longleftarrow} S$

Chapitre 8

Automates à piles

8.1 Introduction

8.1.1 Fonctionnement schématique

Un automate à pile est un automate (en général non déterministe) ayant la capacité en plus de pouvoir gérer une pile de “symboles”. Cela permet à l’automate de mémoriser un nombre important d’informations. L’accès à l’information est cependant de type *Last In First Out*. Ces automates ne reconnaissent qu’un certain type de langages, ceux générés par les grammaires hors contexte.

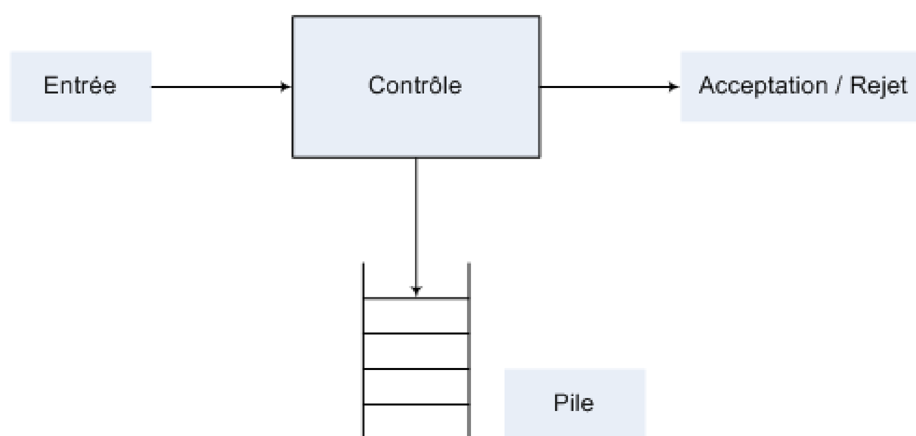


Figure 8.1 – Suggestion d’un automate à pile

La figure 8.1 montre le fonctionnement schématique d’un automate à pile. Le contrôle de l’automate à pile lit un à un les symboles d’entrée. Il base sa transition sur son état courant, le prochain symbole d’entrée et le symbole en sommet de pile. Il peut faire des transitions spontanées consumant ϵ au lieu du prochain symbole d’entrée.

8.1.2 Exemple

Considérons le langage $L_{wcw^R} = \{wcw^R; w \in (0|1)*\}$. Déterminons un automate à pile permettant de reconnaître les mots de ce langage. Sa stratégie sera de stocker dans la pile les 0 et les 1 tant qu'il n'a pas rencontré le marqueur central (c). Il dépilera ensuite les symboles à condition qu'ils correspondent aux caractères d'entrée. S'il réussit à vider la pile sans qu'il reste de caractères à traiter, alors il accepte la chaîne en entrée.
La figure 8.2 simule le fonctionnement de l'automate sur la chaîne 10c01, faisant partie du langage L_{wcw^R}.

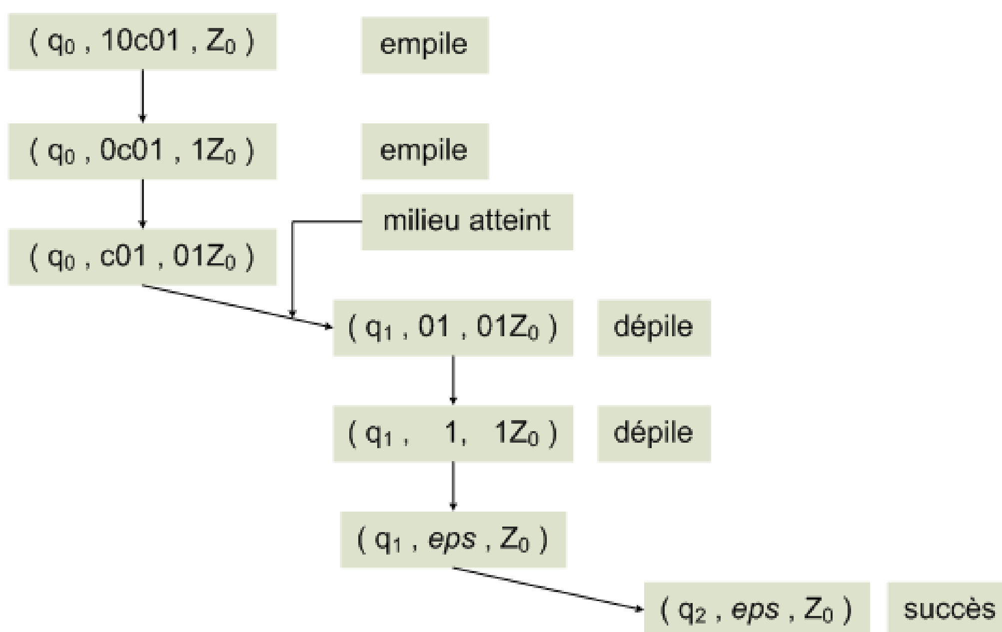


FIGURE 8.2 – Simulation de l'automate

La figure 8.3 représente le diagramme de transition de cet automate.

8.2 Définition formelle

8.2.1 Définition

Définition 8.1 (Automate fini à pile - AFP) *Un automate fini à pile comprend sept composants :*
$P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

- Q *est un ensemble fini d'états*
- Σ *est un ensemble fini de symboles d'entrée*
- Γ *est un ensemble fini dit alphabet de pile. C'est l'ensemble des symboles qui peuvent être empilés*
- δ *est la fonction de transition. Elle prend en argument trois paramètres :*
  - q *un état de* Q
  - a *un symbole d'entrée ou* ϵ

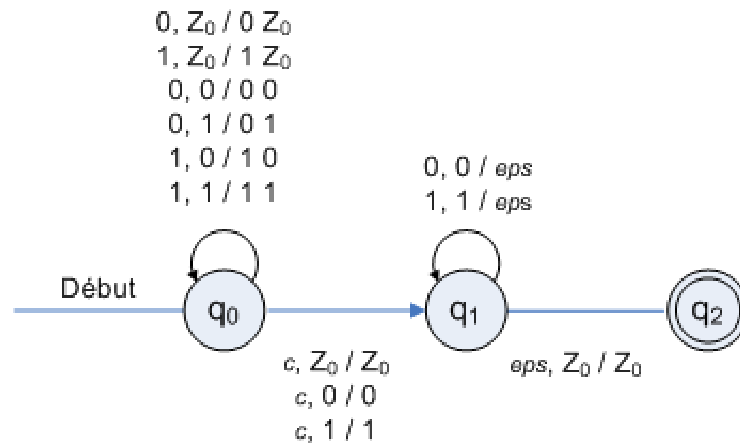


FIGURE 8.3 – Diagramme de transition

- *X un symbole de pile*
  Elle renvoie un ensemble fini de couples (p, γ) où p est le nouvel état et γ est la nouvelle chaîne de symboles qui remplace X au sommet de la pile.
- *q_0 est l'état initial*
- *Z_0 est le symbole de départ de la pile*
- *F est l'ensemble des états finals*

8.2.2 Règles de transition

Lors d'une transition l'automate :

. consume le symbole utilisé dans la transition. Si ϵ est utilisé aucun symbole d'entré n'est consumé.
. passe dans un nouvel état qui peut être le même que le précédent.
. remplace le symbole au sommet de la pile par un nombre quelconque de symboles. Cette chaîne de symboles peut être :
  . ϵ ; ce qui correspond à un dépilement (pop).
  . le même symbole que le sommet actuel ; aucun changement dans la pile.
  . un autre symbole (pop du sommet suivi de push).
  . une suite de symboles ; dépilement du sommet (pop) et empilement (push) des symboles.

8.2.3 Automate à pile déterministe

Un automate à pile est déterministe s'il n'existe qu'une transition possible à chaque état.

Un automate à pile $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ est déterministe, si et seulement si :

- $\delta(q, a, X)$ contient au plus un élément, pour tout état $q \in Q$, $a \in \Sigma$ ou $a = \epsilon$ et $X \in \Gamma$.
- si $\delta(q, a, X)$ est non vide pour un élément $a \in \Sigma$, alors $\delta(q, \epsilon, X)$ doit être vide.

8.3 Exemple

Considérons le langage qui accepte les chaînes palindromes de longueur paire sur l'alphabet $\Sigma = \{0, 1\}$. $L_{ww^r} = \{ww^r | w \in \Sigma*\}$
Un AFP qui engendre ce langage peut être décrit par :

$$P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})$$

La fonction δ est définie par les égalités suivantes. Le fond de la pile est à droite :

Transition		Commentaire
$\delta(q_0, 0, Z_0) =$	$\{(q_0, 0Z_0)\}$	
$\delta(q_0, 1, Z_0) =$	$\{(q_0, 1Z_0)\}$	lecture, empilement
$\delta(q_0, 0, 0) =$	$\{(q_0, 00)\}$	
$\delta(q_0, 0, 1) =$	$\{(q_0, 01)\}$	
$\delta(q_0, 1, 0) =$	$\{(q_0, 10)\}$	
$\delta(q_0, 1, 1) =$	$\{(q_0, 11)\}$	lecture, empilement
$\delta(q_0, \epsilon, Z_0) =$	$\{(q_1, Z_0)\}$	
$\delta(q_0, \epsilon, 0) =$	$\{(q_1, 0)\}$	pas de lecture
$\delta(q_0, \epsilon, 1) =$	$\{(q_1, 1)\}$	pile inchangée
$\delta(q_1, 0, 0) =$	$\{(q_1, \epsilon)\}$	dépilement si le sommet de pile
$\delta(q_1, 1, 1) =$	$\{(q_1, \epsilon)\}$	correspond à la lecture
$\delta(q_1, \epsilon, Z_0) =$	$\{(q_2, Z_0)\}$	fond de pile $\rightarrow$ état final

Il est à remarquer que cet automate n'est pas déterministe.

8.3.1 Diagramme de transition

Le diagramme de transition illustré sur le figure 8.4 généralise celui défini pour les automates finis. Il comporte :

- Un ensemble de nœuds correspondant aux états de l'AFP.
- Une flèche étiquetée “début” indiquant l'état initial.
- Un état final est indiqué par deux cercles concentriques.
- Les arcs correspondent aux transitions. Un arc étiqueté $a, X/\alpha$ de l'état q vers l'état p signifie que $\delta(q, a, X)$ contient le couple (p, α), parmi d'autres couples. L'étiquette de l'arc indique ce qui est utilisé et les ancien et nouveau sommets de la pile.
- Il faut convenir du symbole de fond de pile (par défaut Z_0).

8.3.2 Simulation

La figure 8.5 simule une séquence entière du fonctionnement de l'automate à pile à partir de l'entrée 1111. La configuration initiale est (q_0, 1111 , Z_0). L'automate a l'opportunité de se tromper plusieurs fois. A partir de la configuration initiale, l'automate a deux choix possibles :

- Estimer que le milieu de la chaîne n'a pas été atteint. Cela conduit à la configuration : (q_0, 111 , $1Z_0$).
- Estimer que le milieu de la chaîne a été atteint. Cela conduit à la configuration : (q_1, 1111 , Z_0).

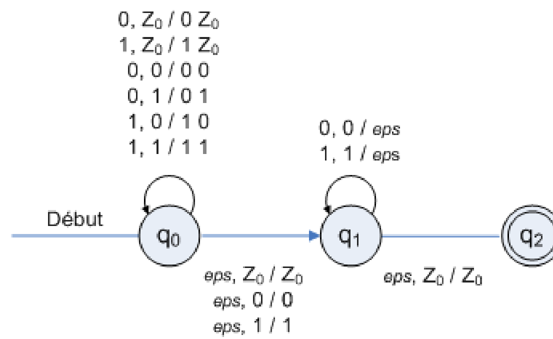


FIGURE 8.4 – Diagramme de transition généralisé

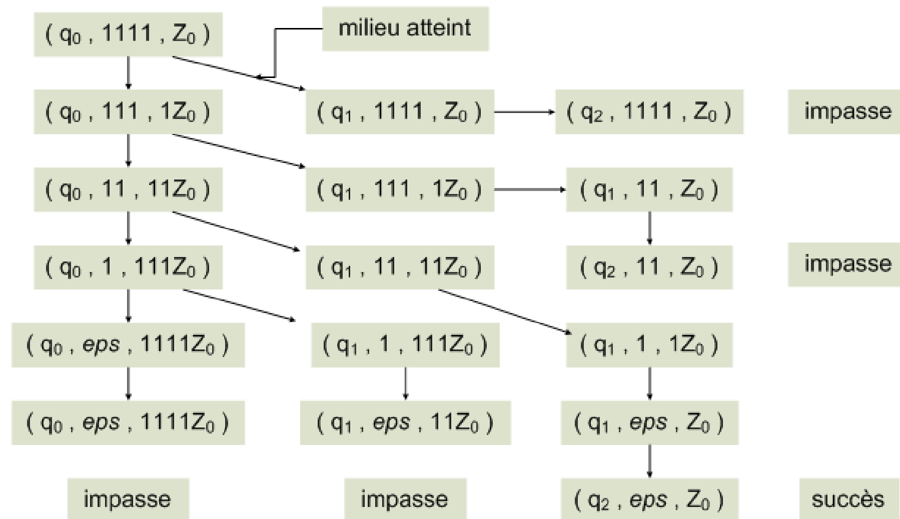


FIGURE 8.5 – Simulation du fonctionnement de l'automate

8.4 Extension de la fonction de transition

8.4.1 Configuration

Un AFP passe de configuration en configuration en réponse aux symboles d'entrée ou ϵ. Une configuration comprend l'état actuel de l'automate ainsi que le contenu de la pile.

Définition 8.2 (Configuration) *Une configuration d'un AFP est un triplet (q, w, γ) où :*
. q est l'état de la configuration.
. w est la chaîne d'entrée restant à analyser.
. γ est le contenu de la pile.

8.4.2 Définition

Soit $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un automate à pile, on symbolise par $\underset{P}{\vdash}$ ou seulement $\vdash$, lorsque l'automate est sous-entendu, la fonction définie par :
. On suppose que $\delta(q, a, X)$ contienne (p, α)
. $\forall w \in \Gamma* : (q, aw, X\beta) \vdash (p, w, \alpha\beta)$

Ce qui signifie que, en consumant le symbole d'entrée a (ce pourrait être éventuellement ϵ) et en remplaçant X au sommet de la pile par α, l'automate passe de l'état q à l'état p. On utilise $\overset{*}{\underset{P}{\vdash}}$ ou plus simplement $\overset{*}{\vdash}$ pour représenter un ou plusieurs pas de l'automate.

Généralisation

$I \overset{*}{\vdash} I$

$I \overset{*}{\vdash} J$ s'il existe K tel que $I \overset{*}{\vdash} K$ et $K \vdash J$

8.5 Langage d'un automate à pile

Le langage d'un automate à pile peut être défini de deux façons selon le type d'acceptation de l'automate.

8.5.1 Acceptation par état final

L'automate reconnaît une chaîne s'il a consumé tous les symboles d'entrée et s'il se trouve dans un état final. La pile peut être vide ou non.

Définition 8.3 (Acceptation par état final) *Soit $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un AFP. Le langage accepté par P par état final est : $L(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \alpha)\}$ où q est un état final et α une suite quelconque de symboles.*

8.5.2 Acceptation par pile vide

L'automate reconnaît une chaîne s'il a consumé tous les symboles d'entrée et si la pile est vide.

Définition 8.4 (Acceptation par pile vide) *Soit $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un AFP. Le langage accepté par P par pile vide est :* $N(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\underset{P}{\vdash}} (q, \epsilon, \epsilon)\}$ *où q est un état quelconque.*

8.5.3 Théorèmes

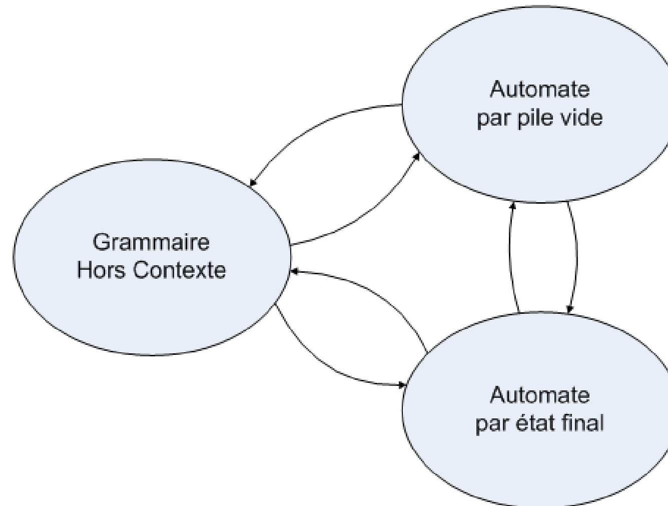


FIGURE 8.6 – Equivalence : grammaire - automate

La classe des langages qui sont $L(P)$ est la même que la classe des langages $N(P)$. C'est aussi la classe des langages hors contexte (cf. figure 8.6).

Théorème 8.1 *Si $L = N(P_N)$ pour un automate à pile $P_N = (Q, \Sigma, \Gamma, \delta_N, q_0, Z_0, F)$ alors il existe un automate à pile P_F tel que $L = L(P_F)$.*

Théorème 8.2 *Si $L = L(P_F)$ pour un automate à pile $P_F = (Q, \Sigma, \Gamma, \delta_F, q_0, Z_0, F)$ alors il existe un automate à pile P_N tel que $L = N(P_N)$.*

Théorème 8.3 *Etant donnée une grammaire hors contexte G, il est possible de construire un automate à pile P tel que $N(P) = L(G)$.*

Théorème 8.4 *Soit $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un AFP. Il existe une grammaire hors contexte G telle que $L(G) = N(P)$.*

Chapitre 9

Analyse Syntaxique Descendante

9.1 Introduction

Un analyseur syntaxique prend en entrée une chaîne d'unités lexicales fournie par l'analyseur lexical et détermine si celle-ci fait partie du langage en construisant l'arbre syntaxique correspondant.
Il doit donc écrire une dérivation, c'est-à dire la suite des règles de production à utiliser pour obtenir la chaîne en entrée.
Si G est une grammaire hors contexte pour un langage de programmation alors $L(G)$ est l'ensemble de tous les programmes que l'on peut écrire dans ce langage. Les symboles terminaux de la grammaire sont des unités lexicales qui ont une représentation sous forme de chaînes de caractères que l'on peut saisir au clavier et qui sont imprimables.

Pour l'analyse on ajoute généralement une variable S' qui devient le nouvel axiome de la grammaire, un terminal \$ désignant la fin de chaîne et une nouvelle règle de production $S' \longrightarrow S\$$.

9.2 Analyseur descendant LL(1)

9.2.1 Problématique

Etant donné un non terminal A, un symbole d'entrée a, à la position p de l'entrée, un analyseur descendant doit décider de la règle à appliquer pour que le sous-arbre étiqueté A soit le sous-arbre correct à la position p.
Une première idée consiste à essayer toutes les règles. La première règle qui fonctionne est la bonne. On peut mémoriser ainsi la bonne dérivation pour les autres occurrences de A.
Ce procédé ne permet cependant pas de reconnaître toutes les chaînes du langage.
Le plus simple est de créer au préalable une table donnant la règle à appliquer connaissant A et a. On construit alors ce que l'on appelle un analyseur prédictif.

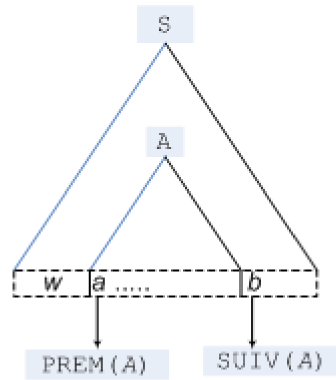


FIGURE 9.1 – Prédiction

Début de résolution

Une règle telle que $A \longrightarrow X_1X_2 \dots X_n$ permet de dire que les premiers symboles que peux générer A contiennent ceux que peut générer X_1. Mais si X_1 est nullifiable (pouvant générer ϵ), il faut considérer aussi ceux que peut générer X_2. Et ainsi de suite.
Si X_1 ne génère pas ϵ, $wA\beta \overset{*}{\Longrightarrow} wX_1X_2 \dots X_n\beta \overset{*}{\Longrightarrow} wa_1w_1X_2 \dots X_n\beta$.
Si X_1 génère ϵ et X_2 ne génère pas ϵ, on aura :
$wA\beta \overset{*}{\Longrightarrow} wX_1X_2 \dots X_n\beta \overset{*}{\Longrightarrow} wX_2 \dots X_n\beta \overset{*}{\Longrightarrow} wa_2w_2X_3 \dots X_n\beta$.

Si $A \overset{*}{\Longrightarrow} \epsilon$, $wA\beta \overset{*}{\Longrightarrow} w\beta$. Il faut ainsi prendre en compte les premiers symboles terminaux que β peut générer et qui sont en fait des symboles qui peuvent suivre A.
On est donc amené à prendre en compte et à calculer les “premiers” de chaque chaîne de symboles apparaissant dans la partie droite d'une règle de production ainsi que les “suivants” de chacune des variables.

Illustration

Les sections suivantes sont illustrées à partir de la grammaire définie par :
$T = \{+, -, *, /, (,), nb\}$; $V = \{E, E', T, T', F\}$; Axiome : E ;
$P = \{$

1. $E \longrightarrow TE'$
2. $E' \longrightarrow +TE'$
3. $E' \longrightarrow -TE'$
4. $E' \longrightarrow \epsilon$
5. $T \longrightarrow FT'$
6. $T' \longrightarrow *FT'$
7. $T' \longrightarrow /FT'$
8. $T' \longrightarrow \epsilon$
9. $F \longrightarrow (E)$

10. $F \longrightarrow nb$

}

9.2.2 Premiers

Définition 9.1 (Choix) *On appelle choix toute chaîne d'éléments de $V \cup T$ pouvant figurer dans la partie droite d'une règle de production.*

Définition 9.2 (Fin de choix) *Une fin de choix α est une chaîne de 0 ou plus éléments de $V \cup T$ telle qu'il existe $A \in V \cup T$ tel que $A\alpha$ soit un choix ou une fin de choix.*

Définition 9.3 (PREM) *Soit $X \in V \cup T$.*
PREM(X) est l'ensemble des terminaux (y compris ϵ) qui peuvent commencer une chaîne dérivée de X. $a \in PREM(X)$, si $X \stackrel{*}{\Rightarrow} a\beta$.
Soit α un choix ou une fin de choix. PREM(α) est l'ensemble des terminaux (y compris ϵ) qui peuvent commencer une chaîne dérivée de α. $a \in$ PREM(α), si $\alpha \stackrel{*}{\Rightarrow} a\beta$

Initialisation

. $\forall\ a \in T$, PREM(a) = $\{a\}$.
. $\forall\ A \in V$, PREM(A) = $\emptyset$.
. Pour tout choix ou fin de choix α non vide, PREM(α) = $\emptyset$.
. PREM(ϵ) = ϵ.

Algorithme de construction des PREM(α)

. Pour chaque règle de production $A \longrightarrow \alpha$, ajouter les éléments de PREM(α) à ceux de PREM(A) y compris ϵ.
. Pour chaque choix ou fin de choix de la forme $X\beta$, ajouter les éléments de PREM(X) à ceux de PREM($X\beta$), sauf ϵ.
  - Lorsque PREM(X) contient ϵ, ajouter les éléments de PREM(β) à ceux de PREM(α) y compris ϵ.

Répéter les phases précédentes tant que l'un des PREM(α) est modifié.

Choix commençant par un terminal

Soit un choix de la forme $a\alpha$, $a \in T$, le seul caractère pouvant commencer ce choix est a. $PREM(a\alpha) = PREM(a) = a$.

Choix dont les premiers sont immédiats

+TE'	$\{+\}$
-TE'	$\{-\}$
*FT'	$\{*\}$
/FT'	$\{/\}$
(E)	$\{(\}$

Construction des premiers des autres choix

	Init	1	2	3	4	PREM(α)
E	$\emptyset$				$\{nb, (\}$	$\{nb, (\}$
TE'	$\emptyset$			$\{nb, (\}$		$\{nb, (\}$
E'	$\emptyset$	$\{\epsilon\}$	$\{\epsilon, +, -\}$			$\{\epsilon, +, -\}$
T	$\emptyset$			$\{nb, (\}$		$\{nb, (\}$
FT'	$\emptyset$	$\{nb\}$	$\{nb, (\}$			$\{nb, (\}$
T'	$\emptyset$	$\{\epsilon\}$	$\{\epsilon, *, /, \}$			$\{\epsilon, *, /, \}$
F	$\emptyset$	$\{nb\}$	$\{nb, (\}$			$\{nb, (\}$
E)	$\emptyset$				$\{nb, (\}$	$\{nb, (\}$

9.2.3 Suivants

Les suivants sont définis pour tous les non terminaux d'une grammaire.

Définition 9.4 (SUIV) *Soit $X \in V$. SUIV(X) est l'ensemble des terminaux qui peuvent suivre immédiatement X dans une dérivation. $a \in$ SUIV(X) s'il existe un choix contenant Xa.*

Si un choix contient XYa et que Y est nullifiable alors a est un suivant de X.

Initialisation

. Pour toute variable A, SUIV(A) $= \emptyset$.
. Ajouter un marqueur de fin de chaîne à l'axiome ($ par exemple).
. Calculer les premiers pour tous les choix et fins de choix de la grammaire.

Algorithme

1. Pour chaque règle $A \longrightarrow \alpha B \beta$, ajouter les éléments de PREM(β) à ceux de SUIV(B), sauf ϵ le cas échéant.
2. Pour chaque règle $A \longrightarrow \alpha B$, ajouter les éléments de SUIV(A) aux éléments de SUIV(B).
3. Pour chaque règle $A \longrightarrow \alpha B \beta$, lorsque $\epsilon \in$ PREM(β), ajouter les éléments de SUIV(A) aux éléments de SUIV(B).

Répéter les phases 2 et 3 tant que l'un des SUIV(A) est modifié.

Illustration

	Init	1	2	3	SUIV(A)
E	$\emptyset$	$\{\$,)\}$			$\{), \$\}$
E'	$\emptyset$	$\{\$,)\}$			$\{\$,)\}$
T	$\emptyset$	$\{+, -, \$\}$	$\{+, -, \$,)\}$		$\{+, -, \$,)\}$
T'	$\emptyset$	$\{+, -, \$\}$		$\{+, -, \$,)\}$	$\{+, -, \$,)\}$
F	$\emptyset$	$\{*, /, +, -, \$\}$	$\{*, /, +, -, \$,)\}$		$\{*, /, +, -, \$,)\}$

9.2.4 Table d'analyse

Définition 9.5 (Table d'analyse) *Une table d'analyse est un tableau M à deux dimensions qui indique, pour chaque variable A et chaque terminal a (ainsi que $), la règle de production à appliquer.*

Algorithme de construction

Pour chaque production $A \longrightarrow \alpha$
pour tout $a \in$ *PREM*(α) *et* $a \neq \epsilon$
ajouter $A \longrightarrow \alpha$ *dans la case* $M[A, a]$
si $\epsilon \in$ *PREM*(α)*, alors*
pour chaque $b \in$ *SUIV*(A)*, ajouter* $A \longrightarrow \alpha$ *dans la case* $M[A, b]$

Une case vide qui devrait être utilisée lors de l'analyse correspond à une erreur de syntaxe.

Illustration

	+	-	*	/	(	)	nb	$
E					1		1	
E'	2	3				4		4
T					5		5	
T'	8	8	6	7		8		8
F					9		10	

9.2.5 Simulation par un automate à pile

On considère l'automate $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ défini par :

Q	=	$\{q\}$	l'ensemble des états.
Σ	=	T	l'ensemble des symboles d'entrée.
Γ	=	$T \cup V$	l'alphabet de pile.
δ			la fonction de transition.
q_0	=	q	l'état initial.
Z_0	=	$\$$	le symbole de départ de la pile.
F	=	$\emptyset$	acceptation par pile vide.

δ est défini par les transitions suivantes (auxquelles on ajoute un paramètre de sortie pour rendre compte des règles utilisées) :

Match : $(q, ax, a\gamma, y) \vdash (q, x, \gamma, y)$
Le symbole d'entrée est le symbole de sommet de la pile.
Il est consumé et le sommet de la pile est dépilé.

Production : $(q, x, A\gamma, y) \vdash (q, x, \alpha\gamma, yy_i)$
Le sommet de la pile est une variable A.
Il existe une règle de production : $A \longrightarrow \alpha$ donnée par M$[A, b]$ où b est le premier symbole de x. Dans ce cas, A est remplacé au sommet de la pile par le choix de la règle, en empilant à partir des symboles les plus à droite. En sortie, le rang y_i de la règle utilisée, est ajouté.

Acceptation : $(q, \$, \$, y) \vdash (q, \epsilon, \epsilon, y)$

Illustration

La table suivante illustre les transitions successives de l'automate à partir de la chaîne d'entrée : 3 + 4 * 5. L'analyseur lexical substitue *nb* à chaque nombre de la chaîne pour communiquer "*nb* + *nb* * *nb*", mais pour une meilleure lisibilité, la chaîne d'entrée est laissée intacte. Le fond de la pile est à droite. L'axiome est empilé.

Entrée	Pile	Transition	Sortie
3 + 4 * 5$	E$	Production	1
3 + 4 * 5$	TE'$	Production	1, 5
3 + 4 * 5$	FT'E'$	Production	1, 5, 10
3 + 4 * 5$	nb T'E'$	Match	1, 5, 10
+ 4 * 5$	T'E'$	Production	1, 5, 10, 8
+ 4 * 5$	E'$	Production	1, 5, 10, 8, 2
+ 4 * 5$	+TE'$	Match	1, 5, 10, 8, 2
4 * 5$	TE'$	Production	1, 5, 10, 8, 2, 5
4 * 5$	FT'E'$	Production	1, 5, 10, 8, 2, 5, 10
4 * 5$	nb T'E'$	Match	1, 5, 10, 8, 2, 5, 10
* 5$	T'E'$	Production	1, 5, 10, 8, 2, 5, 10, 6
* 5$	*FT'E'$	Match	1, 5, 10, 8, 2, 5, 10, 6
5$	FT'E'$	Production	1, 5, 10, 8, 2, 5, 10, 6, 10
5$	nb T'E'$	Match	1, 5, 10, 8, 2, 5, 10, 6, 10
$	T'E'$	Production	1, 5, 10, 8, 2, 5, 10, 6, 10, 8
$	E'$	Production	1, 5, 10, 8, 2, 5, 10, 6, 10, 8, 4
$	$	Accept	1, 5, 10, 8, 2, 5, 10, 6, 10, 8, 4

Il est simple de construire l'arbre d'analyse au fur et à mesure de la sortie de l'automate.

Une table plus simple est parfois suffisante. Elle ne fait apparaître en sortie que la règle de production utilisée :

Entrée	Pile	Sortie
3 + 4 * 5$	E$	E $\longrightarrow$ TE'
3 + 4 * 5$	TE'$	T $\longrightarrow$ FT'
3 + 4 * 5$	FT'E'$	F $\longrightarrow$ nb
3 + 4 * 5$	nb T'E'$	
+ 4 * 5$	T'E'$	T' $\longrightarrow \epsilon$
+ 4 * 5$	E'$	T' $\longrightarrow$ +TE'
+ 4 * 5$	+TE'$	
4 * 5$	TE'$	T $\longrightarrow$ FT'
4 * 5$	FT'E'$	F $\longrightarrow$ nb
4 * 5$	nb T'E'$	
* 5$	T'E'$	T' $\longrightarrow$ *FT'
* 5$	*FT'E'$	
5$	FT'E'$	F $\longrightarrow$ nb
5$	nb T'E'$	
$	T'E'$	T' $\longrightarrow \epsilon$
$	E'$	E' $\longrightarrow \epsilon$
$	$	Acceptation

9.3 Grammaire LL(1)

Toutes les grammaires ne permettent pas de construire une table d'analyse. Certaines font apparaître plusieurs règles pour un couple $(A\ ,\ a)$ de la table.

Définition 9.6 (Grammaire LL(1)) *Une grammaire est dite LL(1) lorsque sa table d'analyse ne comporte pas plusieurs règles de production pour une même case.*

Théorème 9.1 *Une grammaire ambiguë, récursive à gauche ou non factorisée à gauche n'est pas LL(1).*

Lorsqu'une grammaire n'est pas LL(1), il n'existe qu'une alternative : essayer de la rendre LL(1) ou choisir un analyseur syntaxique de nature différente.

9.3.1 Récursivité à gauche

Définition 9.7 (Grammaire propre) *Une grammaire est propre si elle ne contient aucune règle de la forme :* $A \longrightarrow \epsilon$.

Définition 9.8 (Grammaire immédiatement récursive à gauche) *Une grammaire est immédiatement récursive à gauche si elle contient une règle de la forme :* $A \longrightarrow A\alpha$.

Une telle grammaire possède une règle de la forme : $A \longrightarrow \beta$, sinon la règle précédente ne serait pas utilisable. Pour chaque premier(β), il existe au moins deux règles applicables. La grammaire n'est pas LL(1).

Définition 9.9 (Grammaire récursive à gauche) *Une grammaire est récursive à gauche si elle contient une variable* A *se dérivant en* $A\alpha$ ($A \overset{*}{\Longrightarrow} A\alpha$).

Une telle grammaire n'est pas LL(1) pour la même raison que pour la récursivité à gauche immédiate.
Une grammaire peut posséder une récursivité à gauche cachée. C'est le cas lorsque :
$A \overset{*}{\Longrightarrow} \alpha A$ et $\alpha \overset{*}{\Longrightarrow} \epsilon$

Elimination de la récursivité à gauche immédiate

On observe que $A \longrightarrow A\alpha$ et $A \longrightarrow \beta$, permettent de déduire que A produit des chaînes de la forme : $\beta\alpha*$.

Pour éliminer la récursivité à gauche immédiate, il suffit d'ajouter une variable A' et de remplacer les règles telles que :
. $A \longrightarrow A\alpha$
. $A \longrightarrow \beta$
par les règles :
. $A \longrightarrow \beta A'$
. $A' \longrightarrow \alpha A' \mid \epsilon$

Elimination de la récursivité à gauche

Lorsque la grammaire est propre, l'algorithme suivant permet d'éliminer la récursivité à gauche :
Ordonner les non terminaux en A_1, A_2, ... A_n.
Pour i de 1 à n
Pour j de 1 à i - 1
Remplacer les productions $A_i \longrightarrow A_j\alpha$ *où* $A_j \longrightarrow \beta_1|\beta_2|\ldots|\beta_p$
par $A_i \longrightarrow \beta_1\alpha|\beta_2\alpha|\ldots|\beta_p\alpha$
Eliminer la récursivité à gauche immédiate des productions concernant A_j

9.3.2 Factorisation à gauche

Définition 9.10 (Grammaire non factorisée à gauche) *Une grammaire n'est pas factorisée à gauche lorsque deux choix d'une même variable commencent par le même symbole.*

$A \longrightarrow X\alpha$ et $A \longrightarrow X\beta$ sont deux règles applicables pour chaque prem(X). La grammaire n'est pas LL(1).

L'algorithme suivant permet de factoriser à gauche une grammaire :
Pour chaque variable A
Trouver α *le plus long préfixe commun à deux ou plusieurs de ses choix*
si $\alpha \neq \epsilon$
Remplacer $A \longrightarrow \alpha\beta_1|\ldots|\alpha\beta_n|\gamma_1|\ldots|\gamma_p$ *(les* γ_i *ne commencent pas par* α*)*
par :
$A \longrightarrow \alpha A'|\gamma_1|\ldots|\gamma_p$
$A' \longrightarrow \beta_1|\ldots|\beta_n$
Recommencer tant que deux productions commencent par le même symbole

9.4 Conflits

9.4.1 Conflit Premier-Premier

Exemple 1
La grammaire définie par les règles suivantes présente un conflit.

1. $S \longrightarrow aAb$
2. $A \longrightarrow cd$
3. $A \longrightarrow ce$

Le tableau suivant donne les premiers et suivants des variables de la grammaire :

PREM(S) = { a }	SUIV(S) = {$}
PREM(A) = { c }	SUIV(A) = {b}

La table d'analyse est donnée par :

	a	b	c	d	$
S	1				
A			2 ; 3		

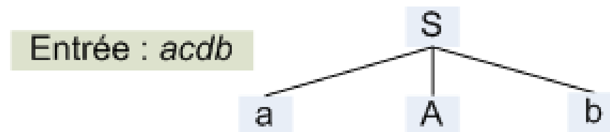


FIGURE 9.2 – Conflit : premier-premier

La table présente un conflit premier-premier à cause de la non-factorisation à gauche de la grammaire. La figure 9.2 montre le conflit lors de l'analyse de la chaîne acdb.
Exemple 2

La grammaire définie par les règles suivantes présente également un conflit premier-premier.

1. $S \longrightarrow aTd$
2. $T \longrightarrow Tb$
3. $T \longrightarrow c$

Le tableau suivant donne les premiers et suivants des variables de la grammaire :

PREM(S) = { a }	SUIV(S) = {$}
PREM(T) = { c }	SUIV(T) = {b, d}

La table d'analyse est donnée par :

	a	b	c	d	$
S	1				
T		2 ; 3			

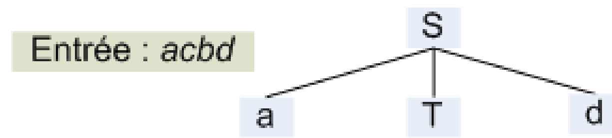


Figure 9.3 – Conflit : premier-premier - Récursivité

La table présente un conflit prem-prem à cause de la récursivité à gauche de la grammaire. La figure 9.3 montre le conflit lors de l'analyse de la chaîne acbd.

Lookahead(k)
Pour lever un conflit premier-premier, on peut indiquer un lookahead > pour un non terminal particulier. Lorsque deux règles entrent en conflit, on considère alors les k prochains caractères de la chaîne d'entrée.

9.4.2 Conflit Premier-Suivant

Exemple 3
La grammaire définie par les règles suivantes présente un conflit premier-suivant.

1. $S \longrightarrow aTb$
2. $T \longrightarrow bT$
3. $T \longrightarrow \epsilon$

Le tableau suivant donne les premiers et suivants des variables de la grammaire :

PREM(S) = { a }	SUIV(S) = {$}
PREM(T) = { b, ϵ }	SUIV(T) = {b}

La table d'analyse est donnée par :

	a	b	$
S	1		
T		2 ; 3	

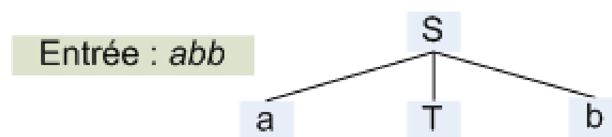


Figure 9.4 – Conflit : premier-suivant

La table présente un conflit premier-suivant à cause de la variable T. T est nullifiable et PREM(T) $\cap$ SUIV(T) $\neq \emptyset$. La figure 9.4 montre le conflit lors de l'analyse de la chaîne abb.

Exemple 4

La grammaire définie par les règles suivantes ne présente pas de conflit bien qu'une variable soit nullifiable.

1. $S \longrightarrow aTb$
2. $T \longrightarrow cT$
3. $T \longrightarrow \epsilon$

Le tableau suivant donne les premiers et suivants des variables de la grammaire :

PREM(S) = { a }	SUIV(S) = {$}
PREM(T) = { c, ϵ }	SUIV(T) = {b}

La table d'analyse est donnée par :

	a	b	c	$
S	1			
T		3	2	

La table ne présente pas de conflit bien que T soit nullifiable.

Choix arbitraire

Lorsque deux règles entrent en conflit, il est possible également de choisir arbitrairement la règle applicable. On choisira par exemple, la première figurant dans la grammaire.

9.4.3 Conclusion

Une grammaire est LL(1) lorsque :

1. Il n'y a pas de conflit Premier-Premier. Pour chaque non terminal, les premiers des choix doivent être tous distincts.
2. Il n'y a pas de conflit Premier-Suivant. Pour chaque non terminal ayant un choix nullifiable, les Premier et Suivant doivent être distincts.
3. Il n'y a pas de choix nullifiables multiples.

Chapitre 10

Analyse Ascendante LR(0)

10.1 Principe d'analyse ascendante

L'analyse ascendante ou par décalage réduction a pour but de construire un arbre d'analyse pour une chaîne d'entrée donnée w à partir de ses feuilles. La racine est construite après que tous les nœuds inférieurs ont été construits. A chaque étape de réduction une sous-chaîne correspondant à la partie droite d'une règle de production est remplacée par la variable de la partie gauche.

10.1.1 Exemple d'analyse

Soit la grammaire G suivante :
$T = \{a, b, c, d\}$ $V = \{S, T\}$
$P = \{$

1. $S \longrightarrow aSbT$
2. $S \longrightarrow cT$
3. $S \longrightarrow d$
4. $T \longrightarrow aT$
5. $T \longrightarrow bS$
6. $T \longrightarrow c$

$\}$
La chaîne $w = accbbadbc$ peut être réduite vers S par les étapes suivantes :

$accbbadbc$
$acTbbadbc$
$aSbbadbc$
$aSbbaSbc$
$aSbbaSbT$
$aSbbS$
$aSbT$
S

10.1.2 Décalage - Réduction

Sur cette chaîne $w = accbbadbc$, les décalages - réductions sont explicités par la table :

$\underline{a}ccbbadbc$	décalage	$aSbbad\underline{b}c$	réduction par $S \longrightarrow d$
$a\underline{c}cbbadbc$	décalage	$aSbbaS\underline{b}c$	décalage
$ac\underline{c}bbadbc$	décalage	$aSbbaSb\underline{c}$	décalage
$acc\underline{b}badbc$	réduction par $T \longrightarrow c$	$aSbbaSbc$	réduction par $T \longrightarrow c$
$acT\underline{b}badbc$	réduction par $S \longrightarrow cT$	$aSbbaSbT$	réduction par $S \longrightarrow aSbT$
$aS\underline{b}badbc$	décalage	$aSbbS$	réduction par $T \longrightarrow bS$
$aSb\underline{b}adbc$	décalage	$aSbT$	réduction par $S \longrightarrow aSbT$
$aSbb\underline{a}dbc$	décalage	S	
$aSbba\underline{d}bc$	décalage		

Ce qui donne la dérivation droite suivante :

$$S \Rightarrow aSbT \Rightarrow aSbbS \Rightarrow aSbbaSbT \Rightarrow aSbbaSbc \Rightarrow aSbbadbc \Rightarrow acTbbadbc \Rightarrow accbbadbc$$

10.1.3 Méthodes d'analyse

Il est à peu près impossible d'écrire un analyseur ascendant à la main. Il faut pour cela utiliser un générateur d'analyseur.
Il existe plusieurs algorithmes d'analyse ascendante, différents principalement dans leur première partie ; la fin de l'analyse étant ensuite la même pour chaque algorithme. Les principales méthodes sont :

– LR(0) : théoriquement importante mais trop faible

– SLR(1) : version renforcée de LR(0) mais assez faible

– LR(1) : très puissante mais nécessite une mémoire importante

– LALR(1) : version affaiblie de LR(1) puissante et exécutable

10.1.4 Manche

La tâche principale d'un analyseur ascendant est de trouver le nœud le plus à gauche qui n'a pas encore été construit mais dont tous les fils sont déjà construits. Cette suite de nœuds est appelé un manche (pivot). Les algorithmes d'analyse ascendante diffèrent uniquement dans la manière de déterminer les manches successifs.

Définition 10.1 (Manche) *Un manche d'un syntagme $\gamma = \alpha\beta w$ est une sous-chaîne β qui correspond à un choix (la partie droite) d'une règle de production $A \longrightarrow \beta$ et une posi-*

tion dans la chaîne γ où la chaîne β peut être trouvée et remplacée par A pour déterminer le syntagme précédent dans une dérivation droite de γ.

Si $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$ alors
$A \longrightarrow \beta$ ou plus simplement β est un manche de $\alpha\beta w$
pour la position suivant α.

Dans une grammaire non ambiguë, à chaque dérivation droite ne correspond qu'un seul manche.

Exemples :

1. $T \longrightarrow c$ est le manche de $accbbadbc$ en position 3.

2. $S \longrightarrow cT$ est le manche de $acTbbadbc$ en position 2.

3. $S \longrightarrow aSbT$ est le manche de $aSbbaSbT$ en position 5.

Lors de l'analyse ascendante d'une chaîne, on est amené à reconnaître une forme $\alpha\beta w$ dans des syntagmes, la partie β correspondant à un manche au moment où le pointeur d'entrée est sur le premier symbole de w. La chaîne w est une chaîne de symboles terminaux non encore explorée.

Définition 10.2 (Préfixe viable) *Les préfixes viables sont les préfixes des syntagmes qui ne s'étendent pas au-delà de l'extrémité droite du manche le plus à droite. Ces préfixes sont ceux que l'on peut trouver sur la pile d'un analyseur.*

Lors de l'analyse de l'exemple précédent, le syntagme $aSbb\underline{a}dbc$ est considéré lorsque le buffer d'entrée ne contient plus que $adbc$. le prochain manche est $S \longrightarrow d$. Les symboles a, S, b, b constituent un préfixe viable et sont sur la pile de l'analyseur.

10.2 Item LR(0)

10.2.1 Définition

Les items LR(0) ont la forme $A \longrightarrow \alpha_1 \cdot \alpha_2$ où $A \longrightarrow \alpha_1\alpha_2$ est une règle de production. Ils n'indiquent que des possibilités d'analyse.
La forme $A \longrightarrow \alpha_1 \cdot \alpha_2$ représente l'hypothèse que $\alpha_1\alpha_2$ est un pivot possible et qu'il sera réduit à A. Le point représente l'avancement de l'analyseur quand il lit la règle de gauche à droite : α_1 a déjà été reconnu et α_2 doit encore être analysé lorsque la production est utilisée.
La forme $A \longrightarrow \alpha_1\alpha_2\cdot$ signifie que le pivot est atteint puisque α_1 et α_2 ont été reconnus.
$S \longrightarrow aSbT$ fournit 5 items :

$S \longrightarrow \cdot\, aSbT$
$S \longrightarrow a \cdot SbT$
$S \longrightarrow aS \cdot bT$
$S \longrightarrow aSb \cdot T$

$S \longrightarrow aSbT\cdot$

Une production telle que $A \longrightarrow \epsilon$ ne donne qu'un seul item : $A \longrightarrow\cdot$.

10.2.2 Fermeture d'un ensemble d'items

Si I est un ensemble d'items pour une grammaire G, Fermeture(I) est l'ensemble des items construit par l'algorithme suivant :

Placer chaque item de I dans Fermeture(I)
Si $A \longrightarrow \alpha_1\cdot B\alpha_2 \in$ Fermeture(I) et $B \longrightarrow \gamma$ est une production de G
ajouter $B \longrightarrow\cdot \gamma$ dans Fermeture(I)
Recommencer tant qu'un nouvel item est ajouté

Exemples

Si $I = \{S \longrightarrow a\cdot SbT\}$
Fermeture(I) $= \{S \longrightarrow a\cdot SbT, S \longrightarrow\cdot cT, S \longrightarrow\cdot d\}$

Si $I = \{S \longrightarrow aSb\cdot T\}$
Fermeture(I) $= \{S \longrightarrow aSb\cdot T, T \longrightarrow\cdot aT, T \longrightarrow\cdot bS, T \longrightarrow\cdot c\}$

10.3 Automate caractéristique canonique

10.3.1 Définition

On construit un automate caractéristique canonique LR(0). Il est défini sur $V \cup T$. Cet automate fini déterministe reflète les décisions que prend l'analyseur lorsqu'il choisit une règle à appliquer. Analyser B dans $A \longrightarrow \alpha_1\cdot B\alpha_2$ se traduit par l'analyse d'une règle issue de B.
Chaque fermeture d'ensemble d'items forme un état de l'automate. Les items de I, tels que $S \longrightarrow aSb\cdot T$ forment le noyau de *Fermeture(I)*.

10.3.2 Diagramme de transition

La figure 10.1 représente le diagramme de transition de l'automate caractéristique canonique LR(0) issu de la table construite à la sectionn 10.3.5.

10.3.3 Transitions

La fonction de transition détermine l'état successeur d'un état donné après l'analyse d'un symbole terminal ou non.

Calcul des transitions

Transition(I,X), I est un ensemble d'items et $X \in T \cup V$.

Pour chaque item $A \longrightarrow \alpha_1\cdot X\alpha_2$ de I on détermine l'item $A \longrightarrow \alpha_1 X\cdot \alpha_2$.
La fermeture de l'ensemble de ces derniers donne Transition(I,X).

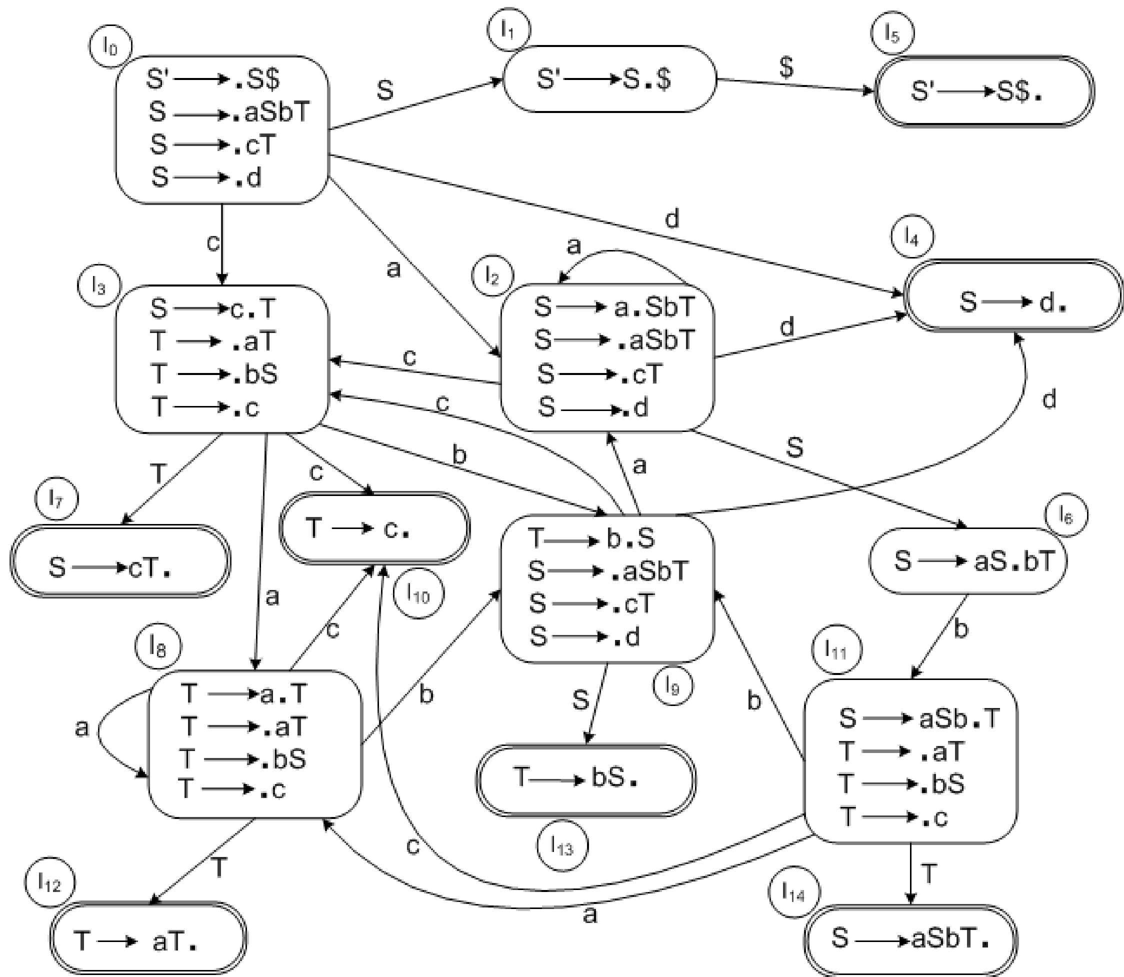


FIGURE 10.1 – Diagramme de transition

10.3.4 Collection des items d'une grammaire

Ajouter l'axiome S' *et la production* $S' \longrightarrow S\$$
$I_0 \longleftarrow$ *Fermeture(*$\{S' \longrightarrow \cdot S\$\}$*)*
$C \longleftarrow \{I_0\}$
Pour chaque $I \in C$
  Pour chaque $X \in V \cup T$ *tel que Transition(*I,X*)* $\neq \emptyset$
    Ajouter Transition(I,X*) à* C
Recommencer tant que C *n'est pas stable*

10.3.5 Construction des items

Nous utilisons la grammaire G définie dans la section 10.1.1 :
$T = \{a, b, c, d\}$ $V = \{S, T\}$
$P = \{$

1. $S \longrightarrow aSbT$
2. $S \longrightarrow cT$
3. $S \longrightarrow d$
4. $T \longrightarrow aT$
5. $T \longrightarrow bS$
6. $T \longrightarrow c$ $\}$

La table suivante montre la construction des items :

$I_0 =$	$\{S' \longrightarrow \cdot S\$, S \longrightarrow \cdot aSbT, S \longrightarrow \cdot cT, S \longrightarrow \cdot d\}$		
$I_1 =$	$\delta(I_0, S) =$	$\{S' \longrightarrow S \cdot \$\}$	
$I_2 =$	$\delta(I_0, a) =$	$\{S \longrightarrow a \cdot SbT, S \longrightarrow \cdot aSbT, S \longrightarrow \cdot cT, S \longrightarrow \cdot d\}$	
$I_3 =$	$\delta(I_0, c) =$	$\{S \longrightarrow c \cdot T, T \longrightarrow \cdot aT, T \longrightarrow \cdot bS, T \longrightarrow \cdot c\}$	
$I_4 =$	$\delta(I_0, d) =$	$\{S \longrightarrow d \cdot\}$	
$I_5 =$	$\delta(I_1, \$) =$	$\{S' \longrightarrow S\$ \cdot\}$	
$I_6 =$	$\delta(I_2, S) =$	$\{S \longrightarrow aS \cdot bT\}$	
	$\delta(I_2, a) =$	$\{S \longrightarrow a \cdot SbT, S \longrightarrow \cdot aSbT, S \longrightarrow \cdot cT, S \longrightarrow \cdot d\}$	$= I_2$
	$\delta(I_2, c) =$	$\{S \longrightarrow c \cdot T, T \longrightarrow \cdot aT, T \longrightarrow \cdot bS, T \longrightarrow \cdot c\}$	$= I_3$
	$\delta(I_2, d) =$	$\{S \longrightarrow d \cdot\}$	$= I_4$
$I_7 =$	$\delta(I_3, T) =$	$\{S \longrightarrow cT \cdot\}$	
$I_8 =$	$\delta(I_3, a) =$	$\{T \longrightarrow a \cdot T, T \longrightarrow \cdot aT, T \longrightarrow \cdot bS, T \longrightarrow \cdot c\}$	
$I_9 =$	$\delta(I_3, b) =$	$\{T \longrightarrow b \cdot S, S \longrightarrow \cdot aSbT, S \longrightarrow \cdot cT, S \longrightarrow \cdot d\}$	
$I_{10} =$	$\delta(I_3, c) =$	$\{T \longrightarrow c \cdot\}$	
$I_{11} =$	$\delta(I_6, b) =$	$\{S \longrightarrow aSb \cdot T, T \longrightarrow \cdot aT, T \longrightarrow \cdot bS, T \longrightarrow \cdot c\}$	

$$\begin{array}{llll}
I_{12} = & \delta(I_8, T) = & \{T \longrightarrow aT\cdot\} & \\
& \delta(I_8, a) = & \{T \longrightarrow a\cdot T, T \longrightarrow\cdot aT, T \longrightarrow\cdot bS, T \longrightarrow\cdot c\} & = I_8 \\
& \delta(I_8, b) = & \{T \longrightarrow b\cdot S, S \longrightarrow\cdot aSbT, S \longrightarrow\cdot cT, S \longrightarrow\cdot d\} & = I_9 \\
& \delta(I_8, c) = & \{T \longrightarrow c\cdot\} & = I_{10}
\end{array}$$

$$\begin{array}{llll}
I_{13} = & \delta(I_9, S) = & \{T \longrightarrow bS\cdot\} & \\
& \delta(I_9, a) = & \{S \longrightarrow a\cdot SbT, S \longrightarrow\cdot aSbT, S \longrightarrow\cdot cT, S \longrightarrow\cdot d\} & = I_2 \\
& \delta(I_9, c) = & \{S \longrightarrow c\cdot T, T \longrightarrow\cdot aT, T \longrightarrow\cdot bS, T \longrightarrow\cdot c\} & = I_3 \\
& \delta(I_9, d) = & \{S \longrightarrow d\cdot\} & = I_4
\end{array}$$

$$\begin{array}{llll}
I_{14} = & \delta(I_{11}, T) = & \{S \longrightarrow aSbT\cdot\} & \\
& \delta(I_{11}, a) = & \{T \longrightarrow a\cdot T, T \longrightarrow\cdot aT, T \longrightarrow\cdot bS, T \longrightarrow\cdot c\} & = I_8 \\
& \delta(I_{11}, b) = & \{T \longrightarrow b\cdot S, S \longrightarrow\cdot aSbT, S \longrightarrow\cdot cT, S \longrightarrow\cdot d\} & = I_9 \\
& \delta(I_{11}, c) = & \{T \longrightarrow c\cdot\} & = I_{10}
\end{array}$$

10.4 Analyse LR(0)

10.4.1 Table des actions

La table des Actions de l'analyseur est remplie selon l'algorithme :

Pour tout état s
si s contient un item tel que $A \longrightarrow \alpha\cdot a\beta$ alors
Action[s] $\leftarrow$ Décalage
sinon
si s contient $A \longrightarrow \alpha\cdot$ (règle i) et $A \neq S'$ alors
Action[s] $\leftarrow$ Réduction i
si s contient $S' \longrightarrow S\$\cdot$ alors
Action[s] $\leftarrow$ Acceptation

Définition 10.3 (Grammaire LR(0)) *Une grammaire est LR(0) si aucune entrée de la table des actions ne contient plus d'une action.*

Table des successeurs

La table des successeurs n'est qu'une simple table de transition d'un automate. La définition des transitions entre les états permet de remplir cette table.

10.4.2 Illustration

Etats	Action	Successeurs						
		a	b	c	d	S	T	$
0	Décalage	2		3	4	1		
1	Décalage							5
2	Décalage	2		3	4	6		
3	Décalage	8	9	10			7	
4	Réduction 3							
5	Acceptation							
6	Décalage		11					
7	Réduction 2							
8	Décalage	8	9	10			12	
9	Décalage	2		3	4	13		
10	Réduction 6							
11	Décalage	8	9	10			14	
12	Réduction 4							
13	Réduction 5							
14	Réduction 1							

10.4.3 Algorithme d'analyse

L'analyse est réalisée grâce à un automate à pile. Son fonctionnement est le suivant :
Syntaxe : *(état, entrée, pile, sortie)*. Le fond de la pile est situé à droite.
Soit s l'état courant

Si Action[s] = Décalage
$(s, ax, s\gamma, y) \vdash (s', x, s'as\gamma, y)$ avec s' = successeur[s,a]
L'automate :
- consume le caractère d'entrée (a),
- empile le caractère et le successeur de l'état actuel,
- et se positionne dans l'état successeur.

Si Action[s] = Réduction j ; (par la règle j : $A \longrightarrow \alpha$)
$\alpha = X_1 X_2 \cdots X_n$
$(s, x, sX_n \cdots s_2 X_2 s_1 X_1 t\gamma, y) \vdash (s', x, s'At\gamma, yj)$ avec s' = successeur[t,A]

L'automate :
- dépile les symboles correspondant à la partie droite de la règle j, ainsi que les sommets intermédiaires,
- empile le symbole gauche de la règle j et le successeur de l'état apparu sur la pile,
- et se positionne dans cet état.

Si Action[s] = Acceptation
chaîne syntaxiquement correcte.

Si Successeur[s] $= \emptyset$ (le successeur recherché n'apparaît pas dans la table)
erreur

10.4.4 Simulation de l'automate

Soit à analyser la chaîne : accbbadbc. L'état initial est empilé.
La table suivante illustre le fonctionnement de l'automate (Acceptation par état final).

Entrée	Pile	Action	Sortie
accbbadbc$	0⊣	Décalage	
ccbbadbc$	2a0⊣	Décalage	
cbbadbc$	3c2a0⊣	Décalage	
bbadbc$	10c3c2a0⊣	Réduction 6	6
bbadbc$	7T3c2a0⊣	Réduction 2	6 2
bbadbc$	6S2a0⊣	Décalage	6 2
badbc$	11b6S2a0⊣	Décalage	6 2
adbc$	9b11b6S2a0⊣	Décalage	6 2
dbc$	2a9b11b6S2a0⊣	Décalage	6 2
bc$	4d2a9b11b6S2a0⊣	Réduction 3	6 2 3
bc$	6S2a9b11b6S2a0⊣	Décalage	6 2 3
c$	11b6S2a9b11b6S2a0⊣	Décalage	6 2 3
$	10c11b6S2a9b11b6S2a0⊣	Réduction 6	6 2 3 6
$	14T11b6S2a9b11b6S2a0⊣	Réduction 1	6 2 3 6 1
$	13S9b11b6S2a0⊣	Réduction 5	6 2 3 6 1 5
$	14T11b6S2a0⊣	Réduction 1	6 2 3 6 1 5 1
$	1S0⊣	Décalage	6 2 3 6 1 5 1
ϵ	5$1S0⊣	Acceptation	6 2 3 6 1 5 1

10.4.5 Conflits

Conflit Décaler - Réduire

Certains états peuvent contenir un item de la forme $A \longrightarrow \alpha \cdot a\beta$ et un item de la forme $A \longrightarrow \alpha\cdot$. Ceci engendre un conflit Décaler - Réduire.

Conflit Réduire - Réduire

Certains états peuvent contenir deux items de la forme $A \longrightarrow \alpha\cdot$, représentant deux situations de réduction par deux productions différentes. Ceci engendre un conflit Réduire - Réduire.

Chapitre 11

Analyse Ascendante SLR(1)

11.1 Introduction

Très peu de grammaires sont LR(0). Par exemple, une grammaire avec une production vide ne peut être LR(0). Si nous considérons une production telle que $T \longrightarrow \epsilon$, un ensemble d'items contenant un item de la forme $A \longrightarrow \alpha \cdot T\beta$ contient aussi l'item $T \longrightarrow \cdot$ d'où un conflit Décaler - Réduire.
La méthode d'analyse SLR(1) (ou Simple LR(1)) est un peu plus puissante que la méthode LR(0), mais encore insuffisante en pratique.

11.2 Algorithme de construction

La méthode LR(0) du chapitre 10 ne considère aucun caractère de prévision. La méthode SLR(1) quant à elle prend en compte les suivants globaux de chaque variable pour construire les tables d'actions et de successeurs. La table des actions est à deux dimensions.
La méthode SLR(1) s'appuie sur la remarque qu'un pivot ne doit pas être réduit à un non terminal A si le caractère de prévision n'appartient pas à *SUIV(A)*.
Le diagramme de transition de l'automate SLR(1) est le même que celui de l'automate LR(0), le nombre d'états est le même, la table des successeurs est la même, mais la table d'actions est différente. L'algorithme de construction des tables est le suivant :

1. Construire la collections des ensembles d'items LR(0) : $\{I_0, I_1, \cdots I_n\}$
   Chaque item I_p permet de construire l'état p de l'analyseur.

2. Pour chaque $\delta(I_i, a) = I_j$, $a \in T$
   mettre Décalage dans la case $M_A[i, a]$ de la table des actions.
   mettre j dans la case $M_S[i, a]$ de la table des successeurs.

3. Pour chaque $\delta(I_i, A) = I_j$, $A \in V$

   mettre j dans la case $M_S[i, A]$ de la table des successeurs.

4. Pour chaque règle k, $A \longrightarrow \alpha$, telle que $A \longrightarrow \alpha\cdot \in I_p$ (sauf pour $A = S'$),
   mettre Réduction k dans chaque case $M_A[p, a]$ où $a \in$ SUIV(A).

5. Si $S' \longrightarrow S\$\cdot \in I_p$,
   mettre Acceptation dans la ligne $M_A[p]$ de la table des actions.

11.3 Tables SLR(1)

Reprenons la grammaire de la section 10.3.5. Les premiers et suivants sont précisés dans les tables suivantes :

Premiers et suivants

	Premiers
S	$\{a, c, d\}$
SbT	$\{a, c, d\}$
T	$\{a, b, c\}$

	Suivants
S	$\{\$, b\}$
T	$\{\$, b\}$

Etats	Action					Successeurs						
	a	b	c	d	$	a	b	c	d	$	S	T
0	D		D	D		2		3	4		1	
1					D					5		
2	D		D	D		2		3	4		6	
3	D	D	D			8	9	10				7
4		R 3			R 3							
5	Acceptation											
6		D					11					
7		R 2			R 2							
8	D	D	D			8	9	10				12
9	D		D	D		2		3	4		13	
10		R 6			R 6							
11	D	D	D			8	9	10				14
12		R 4			R 4							
13		R 5			R 5							
14		R 1			R 1							

Il est possible de fondre ces deux tables. Il suffit de faire suivre l'action de décalage du numéro de l'état successeur lorsqu'il s'agit d'un symbole terminal.

Tables d'analyse

Etats	Action					Succ	
	a	b	c	d	$	S	T
0	D 2		D 3	D 4		1	
1					D 5		
2	D 2		D 3	D 4		6	
3	D 8	D 9	D 10				7
4		R 3			R 3		
5	Acceptation						
6		D 11					
7		R 2			R 2		
8	D 8	D 9	D 10				12
9	D 2		D 3	D 4		13	
10		R 6			R 6		
11	D 8	D 9	D 10				14
12		R 4			R 4		
13		R 5			R 5		
14		R 1			R 1		

11.4 Grammaire SLR(1), non LR(0)

Soit G la grammaire suivante :
$T = \{id, +, (,), [,]\}$ $V = \{E, T\}$
$P = \{$

1. $E \longrightarrow T$
2. $E \longrightarrow E + T$
3. $T \longrightarrow id$
4. $T \longrightarrow (E)$
5. $T \longrightarrow id[E]$

$\}$
Cette grammaire illustre la création d'une expression. Les productions 1 à 4 ne posent pas de problème. L'introduction de la règle 5 signifiant qu'un terme peut être un élément de tableau produit un conflit Décalage - Réduction en analyse LR(0). En effet, après la rencontre du symbole *id*, on ne peut décider si l'on doit réduire en appliquant la règle 3 ou si l'on doit décaler pour compléter le choix de la règle 5.
L'analyse SLR(1), prenant en compte les caractères suivant globalement chacune des variables permet de lever ce conflit. Le terminal [n'apparaissant pas dans les suivants possibles de la variable T indique un décalage obligatoire lorsqu'a été reconnue la séquence *id* [. Ceci est illustré par l'analyse suivante.

Etats

$I_0 = \{S' \longrightarrow \cdot E\$, E \longrightarrow \cdot T, E \longrightarrow \cdot E + T, T \longrightarrow \cdot id, T \longrightarrow \cdot (E), T \longrightarrow \cdot id[E]\}$

$I_1 = \delta(I_0, E) = \{S' \longrightarrow E \cdot \$, E \longrightarrow E \cdot +T\}$
$I_2 = \delta(I_0, T) = \{E \longrightarrow T \cdot\}$
$I_3 = \delta(I_0, id) = \{T \longrightarrow id \cdot, T \longrightarrow id \cdot [E]\}$
$I_4 = \delta(I_0, () = \{T \longrightarrow (\cdot E), E \longrightarrow \cdot T, E \longrightarrow \cdot E + T, T \longrightarrow \cdot id, T \longrightarrow \cdot (E), T \longrightarrow \cdot id[E]\}$

$I_5 = \delta(I_1, \$) = \{S' \longrightarrow E\$ \cdot\}$
$I_6 = \delta(I_1, +) = \{E \longrightarrow E + \cdot T, T \longrightarrow \cdot id, T \longrightarrow \cdot (E), T \longrightarrow \cdot id[E]\}$

$I_7 = \delta(I_3, [) = \{T \longrightarrow id[\cdot E], E \longrightarrow \cdot T, E \longrightarrow \cdot E + T, T \longrightarrow \cdot id, T \longrightarrow \cdot (E),$
$T \longrightarrow \cdot id[E]\}$

$I_8 = \delta(I_4, E) = \{T \longrightarrow (E \cdot), E \longrightarrow E \cdot +T\}$
$\delta(I_4, T) = \{E \longrightarrow T \cdot\} = I_2$
$\delta(I_4, id) = \{T \longrightarrow id \cdot, T \longrightarrow id \cdot [E]\} = I_3$
$\delta(I_4, () = \{T \longrightarrow (\cdot E), E \longrightarrow \cdot T, E \longrightarrow \cdot E + T, T \longrightarrow \cdot id,$
$T \longrightarrow \cdot (E), T \longrightarrow \cdot id[E]\} = I_4$

$I_9 = \delta(I_6, T) = \{E \longrightarrow E + T \cdot\}$
$\delta(I_6, id) = \{T \longrightarrow id \cdot, T \longrightarrow id \cdot [E]\} = I_3$
$\delta(I_6, () = \{T \longrightarrow (\cdot E), E \longrightarrow \cdot T, E \longrightarrow \cdot E + T, T \longrightarrow \cdot id,$
$T \longrightarrow \cdot (E), T \longrightarrow \cdot id[E]\} = I_4$

$I_{10} = \delta(I_7, E) = \{T \longrightarrow id[E \cdot], E \longrightarrow E \cdot +T\}$
$\delta(I_7, T) = \{E \longrightarrow T \cdot\} = I_2$
$\delta(I_7, id) = \{T \longrightarrow id \cdot, T \longrightarrow id \cdot [E]\} = I_3$
$\delta(I_7, () = \{T \longrightarrow (\cdot E), E \longrightarrow \cdot T, E \longrightarrow \cdot E + T, T \longrightarrow \cdot id,$
$T \longrightarrow \cdot (E), T \longrightarrow \cdot id[E]\} = I_4$

$I_{11} = \delta(I_8,)) = \{T \longrightarrow (E) \cdot\}$
$\delta(I_8, +) = \{E \longrightarrow E + \cdot T, T \longrightarrow \cdot id, T \longrightarrow \cdot (E), T \longrightarrow \cdot id[E]\} = I_6$

$I_{12} = \delta(I_{10},]) = \{T \longrightarrow id[E] \cdot\}$
$\delta(I_{10}, +) = \{E \longrightarrow E + \cdot T, T \longrightarrow \cdot id, T \longrightarrow \cdot (E), T \longrightarrow \cdot id[E]\} = I_6$

11.4.1 Premiers - Suivants

	Premiers
E	$\{id, (\}$
T	$\{id, (\}$
E + T	$\{id, (\}$

	Suivants
E	$\{\$, +,),]\}$
T	$\{\$, +,),]\}$

11.4.2 Tables d'analyse

Etats	Action							Succ	
	id	+	(	)	[	]	$	E	T
0	D 3		D 4					1	2
1		D 6					D 5		
2		R 1		R 1		R 1	R 1		
3		R 3		R 3	D 7	R 3	R 3		
4	D 3		D 4					8	2
5	Acceptation								
6	D 3		D 4						9
7	D 3		D 4					10	2
8		D 6		D 11					
9		R 2		R 2		R 2	R 2		
10		D 6				D 12			
11		R 4		R 4		R 4	R 4		
12		R 5		R 5		R 5	R 5		

11.4.3 Simulation de l'automate

Soit à analyser la chaîne : tab[i] + p
L'analyse lexicale transforme l'entrée en : id[id] + id. La table suivante illustre le fonctionnement de l'automate (Acceptation par état final).

Entrée	Pile	Action	Sortie
id[id] + id$	0⊣	Décalage	
[id] + id$	3 id 0⊣	Décalage	
id] + id$	7 [3 id 0⊣	Décalage	
] + id$	3 id 7 [3 id 0⊣	Réduction	3
] + id$	2 T 7 [3 id 0⊣	Réduction	3 1
] + id$	10 E 7 [3 id 0⊣	Décalage	3 1
+ id$	12] 10 E 7 [3 id 0⊣	Réduction	3 1 5
+ id$	2 T 0⊣	Réduction	3 1 5 1
+ id$	1 E 0⊣	Décalage	3 1 5 1
id$	6 + 1 E 0⊣	Décalage	3 1 5 1
$	3 id 6 + 1 E 0⊣	Réduction	3 1 5 1 3
$	9 T 6 + 1 E 0⊣	Réduction	3 1 5 1 3 2
$	1 E 0⊣	Décalage	3 1 5 1 3 2
ϵ	5 $ 1 E 0⊣	Acceptation	3 1 5 1 3 2

11.5 Conflits

Les mêmes types de conflits que dans le cas des grammaires LR(0) subsistent.

11.5.1 Conflit Décaler - Réduire

Certains états peuvent contenir un item de la forme $A \longrightarrow \alpha \cdot a\beta$ et un item de la forme $A \longrightarrow \alpha \cdot$. Ceci engendre un conflit Décaler - Réduire.

11.5.2 Conflit Réduire - Réduire

Certains états peuvent contenir deux items de la forme $A \longrightarrow \alpha \cdot$, représentant deux situations de réduction par deux productions différentes. Ceci engendre un conflit Réduire - Réduire.

Chapitre 12

Analyse LR(1)

12.1 Exemple

Considérons la grammaire G :
T = $\{=, *, id\}$ V = $\{S, G, D\}$
P = {
$S \longrightarrow G = D$
$S \longrightarrow D$
$G \longrightarrow *D$
$G \longrightarrow id$
$D \longrightarrow G$
}
Le symbole = appartient à $SUIV(D)$. En effet on peut dériver :
$S \Rightarrow G = D \Rightarrow *D = D$
$I_0 = \{S' \longrightarrow \cdot S\$, S \longrightarrow \cdot G = D, S \longrightarrow \cdot D, D \longrightarrow \cdot G, G \longrightarrow \cdot *D, G \longrightarrow \cdot id\}$
I_1 = Transition(I_0, G)= $\{S \longrightarrow G \cdot = D, D \longrightarrow G \cdot \}$.
En analyse SLR(1), cette grammaire possède un état LR(0), qui indique de réduire par la production $D \longrightarrow G$ en présence de tout suivant de D et en particulier =. Un syntagme tel que $G = **id$ pourrait être réduit en $D = **id$ or un tel syntagme commençant par $D =$ ne peut exister. Cette grammaire n'est donc pas SLR(1).

12.2 Introduction

Si un ensemble d'items I_i contient $A \longrightarrow \alpha \cdot$ et si $a \in SUIV(A)$ alors la table des actions SLR indiquera de réduire. Cependant cette décision peut être illicite. En effet, il peut exister des préfixes $\beta\alpha$ ne pouvant pas être suivis de a.
Des conflits Décaler-Réduire peuvent provenir du fait que la méthode SLR n'est pas suffisamment puissante pour se rappeler assez de contexte gauche pour décider de l'action à effectuer par l'analyseur.
Il faut attacher davantage d'information aux items LR(0), ce qui permet d'éviter certaines réductions invalides. Pour cela, il faut que chaque état d'un analyseur LR indique exactement les symboles d'entrée qui peuvent suivre un manche $A \longrightarrow \alpha$ pour lesquels une

réduction vers A est possible.

12.3 Items LR(1)

12.3.1 Définition

Définition 12.1 (Item LR(1)) *Un item LR(1) est une forme telle que* $[A \longrightarrow \alpha \cdot \beta, a]$ *où* $A \longrightarrow \alpha\beta$ *est une production de la grammaire et* a *est un symbole terminal,* ϵ *ou la marque de fin d'entrée $.*

Un item $[A \longrightarrow \alpha \cdot, a]$ signifie Réduire par la production $A \longrightarrow \alpha$ uniquement si le prochain symbole d'entrée est a. Il ne sera pas ainsi obligatoire de réduire pour tout suivant de A. Un item $[A \longrightarrow \alpha \cdot b\beta, a]$ impliquera un décalage.

Définition 12.2 (Item LR(1) valide) *Un item LR(1)* $[A \longrightarrow \alpha \cdot \beta, a]$ *est valide pour un préfixe viable* $\delta\alpha$*, s'il existe une dérivation droite telle que :* $S \underset{rm}{\overset{*}{\Longrightarrow}} \delta A a w \underset{rm}{\Longrightarrow} \delta\alpha\beta a w$

12.3.2 Fermeture d'un ensemble d'items LR(1)

Considérons un item $[A \longrightarrow \alpha \cdot B\beta, a]$, valide pour un préfixe $\delta\alpha$. Il existe une dérivation : $S \underset{rm}{\overset{*}{\Longrightarrow}} \delta A a w \underset{rm}{\Longrightarrow} \delta\alpha B\beta a w$. Pour une production $B \longrightarrow \eta$, la dérivation devient : $S \underset{rm}{\overset{*}{\Longrightarrow}} \delta A a w \underset{rm}{\Longrightarrow} \delta\alpha B\beta a w \underset{rm}{\Longrightarrow} \delta\alpha\eta\beta a w$.
Les items valides pour le même préfixe $\delta\alpha$ sont les items $[B \longrightarrow \cdot \eta, b]$ tels que $b \in PREM(\beta a w)$.

Remarque :
$PREM(\beta a w) = PREM(\beta a)$
Si β est nullifiable alors $a \in PREM(\beta a)$
Si β n'est pas nullifiable alors $PREM(\beta a) = PREM(\beta)$

Algorithme de fermeture

Soit I un ensemble d'items d'une grammaire G.

Placer chaque item de I *dans Fermeture(I).*
Si l'item $[A \longrightarrow \alpha \cdot B\beta, a] \in$ *Fermeture(I)*
et si $B \longrightarrow \gamma$ *est une production de G*
Pour chaque terminal $b \in$ *PREM(*βa*)*
Ajouter l'item $[B \longrightarrow \cdot \gamma, b]$ *à Fermeture(I).*
Recommencer tant que Fermeture(I) n'est pas stable.

12.3.3 Transition

Détermination de $Transition(I, X)$, où I est un ensemble d'items d'une grammaire G et X un élément de $T \cup V$.

Considérer les items de la forme $[A \longrightarrow \alpha_1 \cdot X\alpha_2, a] \in I$

Construire J l'ensemble des items de la forme $[A \longrightarrow \alpha_1 X \cdot \alpha_2, a]$
Transition(I,X) = Fermeture(J)

12.3.4 Calcul de l'ensemble des items

Soit à calculer l'ensemble des items LR(1) d'une grammaire.
Ajouter l'axiome S' *et la production* $S' \longrightarrow S\$$
$I_0 \longleftarrow$ *Fermeture(*$\{[S' \longrightarrow \cdot S\$, \epsilon]\}$*)*
$C \longleftarrow \{I_0\}$
Pour chaque $I \in C$
Pour chaque $X \in V \cup T$ *tel que Transition(*I,X*)* $\neq \emptyset$
Ajouter Transition(I,X*) à* C
Recommencer tant que C *n'est pas stable*

12.3.5 Tables d'analyse LR(1)

1. Construire C = $\{I_0, I_1, \ldots I_n\}$ l'ensemble des items LR(1). Chaque I_i permet de construire l'état i de l'analyseur.
2. Si $[A \longrightarrow \alpha \cdot a\beta, b] \in I_i$
   mettre Décalage dans $M_A[i, a]$ de la table des actions.
   On ne se préoccupe pas de b.
   mettre j dans $M_S[i, a]$ de la table des successeurs où $I_j = \delta(I_i, a)$
3. Si $[A \longrightarrow \alpha \cdot, a] \in I_i$ et $A \neq S'$
   mettre Réduction p (où p est le rang de la production $A \longrightarrow \alpha$)
   dans $M_A[i, a]$ de la table des actions.
4. Si $[S' \longrightarrow S\$ \cdot, \epsilon] \in I_i$
   mettre Acceptation dans $M_A[i]$ de la table des actions.
5. Si $\delta(I_i, A) = I_j$
   mettre j dans $M_S[i, A]$ de la table des successeurs

Définition 12.3 (Grammaire LR(1)) *Une grammaire est LR(1) si aucune case de la table des actions ne contient plus d'une action.*

12.3.6 Illustration

Soit G la grammaire :
T = $\{c, d\}$, V = $\{S, C\}$
P = { 1. $S \longrightarrow CC$
2. $C \longrightarrow cC$ PREM(C) = $\{c, d\}$.
3. $C \longrightarrow d$ }

Ensemble des Items LR(1)

$$I_0 = \{[S' \longrightarrow \cdot S\$, \epsilon], [S \longrightarrow \cdot CC, \$], [C \longrightarrow \cdot cC, c], [C \longrightarrow \cdot cC, d], [C \longrightarrow \cdot d, c], [C \longrightarrow \cdot d, d]\}$$

$$I_1 = \delta(I_0, S) = \{[S' \longrightarrow S \cdot \$, \epsilon]\}$$
$$I_2 = \delta(I_0, C) = \{[S \longrightarrow C \cdot C, \$], [C \longrightarrow \cdot cC, \$], [C \longrightarrow \cdot d, \$]\}$$
$$I_3 = \delta(I_0, c) = \{[C \longrightarrow c \cdot C, c], [C \longrightarrow c \cdot C, d], [C \longrightarrow \cdot cC, c], [C \longrightarrow \cdot cC, d], [C \longrightarrow \cdot d, c], [C \longrightarrow \cdot d, d]\}$$
$$I_4 = \delta(I_0, d) = \{[C \longrightarrow d \cdot, c][C \longrightarrow d \cdot, d]\}$$

$$I_5 = \delta(I_1, \$) = \{[S' \longrightarrow S\$ \cdot, \epsilon]\}$$

$$I_6 = \delta(I_2, C) = \{[S \longrightarrow CC \cdot, \$]\}$$
$$I_7 = \delta(I_2, c) = \{[C \longrightarrow c \cdot C, \$], [C \longrightarrow \cdot cC, \$], [C \longrightarrow \cdot d, \$]\}$$
$$I_8 = \delta(I_2, d) = \{[C \longrightarrow d \cdot, \$]\}$$

$$I_9 = \delta(I_3, C) = \{[C \longrightarrow cC \cdot, c], [C \longrightarrow cC \cdot, d]\}$$
$$\delta(I_3, c) = \{[C \longrightarrow c \cdot C, c], [C \longrightarrow c \cdot C, d], [C \longrightarrow \cdot cC, c], [C \longrightarrow \cdot cC, d], [C \longrightarrow \cdot d, c], [C \longrightarrow \cdot d, d]\} = I_3$$
$$\delta(I_3, d) = \{[C \longrightarrow d \cdot, c][C \longrightarrow d \cdot, d]\} = I_4$$

$$I_{10} = \delta(I_7, C) = \{[C \longrightarrow cC \cdot, \$]\}$$
$$\delta(I_7, c) = \{[C \longrightarrow c \cdot C, \$], [C \longrightarrow \cdot cC, \$], [C \longrightarrow \cdot d, \$]\} = I_7$$
$$\delta(I_7, d) = \{[C \longrightarrow d \cdot, \$]\} = I_8$$

Tables d'analyse

Etats	Action			Succ	
	c	d	\$	S	C
0	D 3	D 4		1	2
1			D 5		
2	D 7	D 8			6
3	D 3	D 4			9
4	R 3	R 3			
5	Acceptation				
6			R 1		
7	D 7	D 8			10
8			R 3		
9	R 2	R 2			
10			R 2		

Analyse de la chaîne ccdcd

Entrée	Pile	Action	Sortie
ccdcd$	0⊣	Décalage	
cdcd$	3 c 0⊣	Décalage	
dcd$	3 c 3 c 0⊣	Décalage	
cd$	4 d 3 c 3 c 0⊣	Réduction	3
cd$	9 C 3 c 3 c 0⊣	Réduction	3 2
cd$	9 C 3 c 0⊣	Réduction	3 2 2
cd$	2 C 0⊣	Décalage	3 2 2
d$	7 c 2 C 0⊣	Décalage	3 2 2
$	8 d 7 c 2 C 0⊣	Réduction	3 2 2 3
$	10 C 7 c 2 C 0⊣	Réduction	3 2 2 3 2
$	6 C 2 C 0⊣	Réduction	3 2 2 3 2 1
$	1 S 0⊣	Décalage	3 2 2 3 2 1
ϵ	5 $ 1 S 0⊣	Acceptation	

12.3.7 Exemples de conflits LR(1)

Décaler - Réduire

Lorsqu'un état contient des items tels que : $[A \longrightarrow \alpha\cdot, a]$ et $[A \longrightarrow \beta\cdot a\gamma, b]$, il dénote un conflit Décaler - Réduire à la rencontre du caractère a.

Réduire - Réduire

Lorsqu'un état contient des items tels que : $[A \longrightarrow \alpha\cdot, a]$ et $[B \longrightarrow \alpha\cdot, a]$, il dénote un conflit Réduire - Réduire à la rencontre du caractère a commun aux suivants de A et de B.

12.4 Analyse LALR(1)

La méthode LALR(1) (LookAheadLR) est presque aussi puissante que la méthode LR(1) et aussi compacte que la méthode SLR(1). C'est certainement la plus utilisée par les générateurs d'analyseur syntaxique. Les tables SLR et LALR ont le même nombre d'états. Nous construirons les états LALR à partir des états LR bien qu'en pratique un algorithme permette de les générer à partir des items LR(0).

12.4.1 Cœur d'un ensemble d'items

$$
\begin{aligned}
I_3 = \{\ & [C \longrightarrow c\cdot C, c], [C \longrightarrow c\cdot C, d] \\
& [C \longrightarrow\cdot cC, c], [C \longrightarrow\cdot cC, d] \\
& [C \longrightarrow\cdot d, c], [C \longrightarrow\cdot d, d]\ \}
\end{aligned}
$$

I_3 peut être réécrit en compactant les items bâtis sur le même item LR(0) :

$$
\begin{aligned}
I_3 = \{\ & [C \longrightarrow c\cdot C, c/d] \\
& [C \longrightarrow\cdot cC, c/d]
\end{aligned}
$$

$[C \longrightarrow \cdot\, d, c/d]$ }

Le noyau de l'état est l'ensemble des items placés initialement dans celui-ci et permettant de générer les autres par fermeture. Sauf pour l'état initial, le noyau ne contient pas d'items dont le point est situé en première position.
Le cœur de l'état est l'ensemble des items LR(0) obtenus à partir des items LR(1), par restriction sur les symboles de prévision.
La figure 12.1 indique les noyau et cœur de l'état 3.

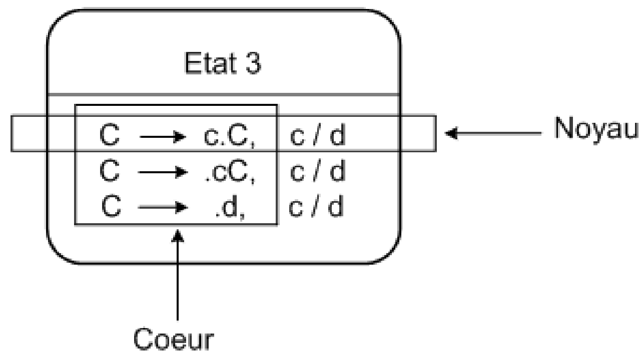


FIGURE 12.1 – Noyau et cœur de l'état 3

Dans l'exemple de la section 12.3.6 les items I_3 et I_7, I_4 et I_8, I_9 et I_{10} ont respectivement le même cœur.

12.4.2 Fusion de deux états

Il est possible de fusionner deux états ayant le même cœur. Il suffit de réunir les listes de symboles de prévision pour chaque item LR(0) correspondant.
$I_3 \cup I_7 = J_{37}$
$J_{37} = \{ \ [C \longrightarrow c\cdot C, c/d/\$]$
$[C \longrightarrow \cdot\, cC, c/d/\$]$
$[C \longrightarrow \cdot\, d, c/d/\$] \ \}$
$I_4 \cup I_8 = J_{48}$
$J_{48} = \{ \ [C \longrightarrow d\cdot, c/d/\$] \ \}$

La fusion de deux ou plusieurs états est consistante par transition. En effet : Si I et J ont le même cœur alors Transition(I,X) et Transition(J,X) auront le même cœur puisqu'ils ne diffèreront que sur les symboles de prévision (voir illustration en figure 12.2).

12.4.3 Tables d'analyse LALR

1. Construire C = $\{I_0, I_1, \ldots I_n\}$ l'ensemble des items LR(1) pour la grammaire augmentée.
2. Pour chaque cœur, trouver les états ayant ce même cœur et les remplacer par leur fusion. Soit C' = $\{J_0, J_1, \ldots J_m\}$ l'ensemble des items LALR(1) qui en résultent

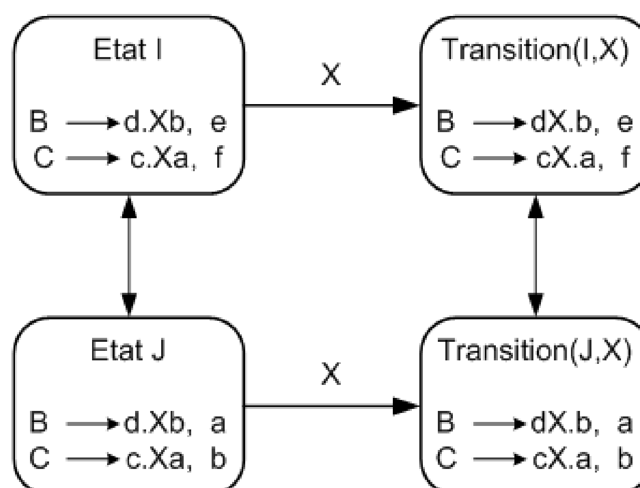


FIGURE 12.2 – Consistance de la fusion et de la fonction de transition

$(m \leq n)$. Chaque J_i permet de construire l'état i de l'analyseur.

3. Soit δ et Δ les fonctions de transition sur les états LR(1) et LALR(1).
   - Si I_k n'a pas fusionné, il est renommé J_k et
     $\Delta(J_k, X) = \delta(I_k, X)$ pour tout X.
   - Si $\delta(I_k, X) = I_p$ et si I_k a fusionné en J_l alors I_p a fusionné en J_q et
     $\Delta(J_l, X) = J_q$
4. Si $[A \longrightarrow \alpha \cdot a\beta, b] \in J_i$
   mettre Décalage dans $M_A[i, a]$ de la table des actions.
   On ne se préoccupe pas de b.
   mettre j dans $M_S[i, a]$ de la table des successeurs où $J_j = \Delta(J_i, a)$
5. Si $[A \longrightarrow \alpha \cdot, a] \in J_i$ et $A \neq S'$
   mettre Réduction p (où p est le rang de la production $A \longrightarrow \alpha$)
   dans $M_A[i, a]$ de la table des actions.
6. Si $[S' \longrightarrow S\$ \cdot, \epsilon] \in J_i$
   mettre Acceptation dans $M_A[i]$ de la table des actions.
7. Si $\Delta(J_i, A) = J_j$
   mettre j dans $M_S[i, A]$ de la table des successeurs

Définition 12.4 (Grammaire LALR(1)) *Une grammaire est LALR(1) si aucune case de la table des actions ne contient plus d'une action.*

12.4.4 Exemple

Reprenons l'exemple de la section 12.3.6. Les états pouvant fusionner sont les états I_3 et I_7, I_4 et I_8, I_9 et I_{10}. On obtient les nouveaux états J_{37}, J_{48} et J_{910}.

$$J_{37} = \{ \; [C \longrightarrow c \cdot C, c/d/\$] \\ \quad [C \longrightarrow \cdot cC, c/d/\$]$$

$[C \longrightarrow \cdot\, d, c/d/\$]$ }

$J_{48} = \{\ [C \longrightarrow d\cdot\,, c/d/\$]\ \}$

$J_{910} = \{\ [C \longrightarrow cC\cdot\,, c/d/\$]\ \}$

Tables d'analyse

Analyse LR					
Etats	Action			Succ	
	c	d	$	S	C
0	D 3	D 4		1	2
1			D 5		
2	D 7	D 8			6
3	D 3	D 4			9
4	R 3	R 3			
5	Acceptation				
6			R 1		
7	D 7	D 8			10
8			R 3		
9	R 2	R 2			
10			R 2		

Analyse LALR					
Etats	Action			Succ	
	c	d	$	S	C
0	D 37	D 48		1	2
1			D 5		
2	D 37	D 48			6
37	D 37	D 48			910
48	R 3	R 3	R 3		
5	Acceptation				
6			R 1		
910	R 2	R 2	R 2		

12.4.5 Conflits

La fusion de deux états ne peut entraîner de nouveaux conflits Décaler - Réduire qui n'auraient pas figuré dans les états LR(1). En effet :
Soit J_{kl} un état LALR présentant un conflit Décaler - Réduire. J_{kl} est de la forme :
$J_{kl} = \{[A \longrightarrow \alpha\cdot\,, a], [B \longrightarrow \beta\cdot a\gamma, b], \cdots\}$
Soit I_k l'état LR contenant $[A \longrightarrow \alpha\cdot\,, a]$. Puisque I_k et I_l ont le même cœur, il contient également un item tel que $[B \longrightarrow \beta\cdot a\gamma, c]$.
I_k présente donc déjà un conflit Décaler - Réduire.

La fusion de deux états peut entraîner de nouveaux conflits Réduire - Réduire qui n'auraient pas figuré dans les états LR(1). Ceci explique l'existence de grammaire LR(1) non LALR(1).

12.4.6 Grammaire LR(1) non LALR(1)

Soit G la grammaire :
T = $\{a, b, c, d, e\}$, V = $\{S, A, B\}$
P = {
1. $S \longrightarrow aAd/bBd/aBe/bAe$
2. $A \longrightarrow c$
3. $B \longrightarrow c$ }
$I_0 = \{[S' \longrightarrow \cdot\, S\$, \epsilon],$

$[S \longrightarrow \cdot\, aAd, \$]$
$[S \longrightarrow \cdot\, bBd, \$]$
$[S \longrightarrow \cdot\, aBe, \$]$
$[S \longrightarrow \cdot\, bAe, \$]\}$

$I_2 = \delta(I_0, a) = \{[S \longrightarrow a\cdot\, Ad, \$], [S \longrightarrow a\cdot\, Be, \$], [A \longrightarrow \cdot\, c, d], [B \longrightarrow \cdot\, c, e]\}$
$I_3 = \delta(I_0, b) = \{[S \longrightarrow b\cdot\, Bd, \$], [S \longrightarrow b\cdot\, Ae, \$], [B \longrightarrow \cdot\, c, d], [A \longrightarrow \cdot\, c, e]\}$
$I_k = \delta(I_2, c) = \{[A \longrightarrow c\cdot\,, d], [B \longrightarrow c\cdot\,, e]\}$
$I_l = \delta(I_3, c) = \{[A \longrightarrow c\cdot\,, e], [B \longrightarrow c\cdot\,, d]\}$
I_k et I_l ne présentent pas de conflit Réduire - Réduire.
Le préfixe *ac* est viable et les chaînes *acd* et *ace* sont reconnues :
$S \underset{rm}{\Longrightarrow} aAd \underset{rm}{\Longrightarrow} acd$
$S \underset{rm}{\Longrightarrow} aBe \underset{rm}{\Longrightarrow} ace$

Après fusion :
$J_{kl} = \{[A \longrightarrow c\cdot\,, d/e], [B \longrightarrow c\cdot\,, e/d]\}$
J_{kl} présente un conflit Réduire - Réduire à la rencontre des symboles d et e.

Chapitre 13

Analyse Sémantique

13.1 Introduction

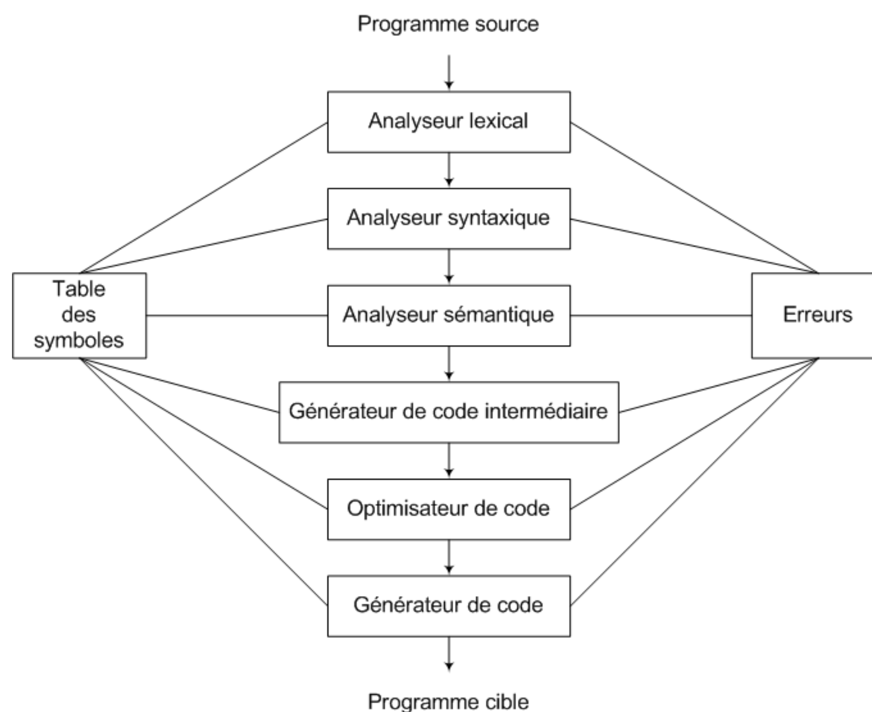


FIGURE 13.1 – Les différentes étapes de la compilation

La figure 13.1 représente de façon théorique les phases d'un compilateur. L'analyse syntaxique produit une représentation sous forme d'arbre du texte d'entrée. L'arbre syntaxique (ou concret) n'est cependant pas la représentation la plus usuelle de la structure du programme source. On lui préfère l'arbre abstrait plus compacte dont les nœuds internes sont les mots-clés du langage ou les opérateurs et leurs fils les opérandes. La construction de cet arbre est basée sur les mêmes concepts que ceux utilisés par l'analyse sémantique.

Celle-ci peut d'ailleurs être effectuée en parallèle de l'analyse syntaxique.
La phase d'analyse sémantique rassemble des informations qui serviront dans la production de code et contrôle un certain nombre de cohérences sémantiques. Par exemple, c'est au cours de cette phase que sont déterminés les instructions, les expressions et les identificateurs. Sont également contrôlés les type dans les affectations et les passages de paramètres. Les types d'indices utilisés pour les tableaux peuvent aussi être vérifiés.
Les grammaires hors contexte ne contiennent pas tous les éléments dont l'analyse sémantique a besoin. Ceux-ci dépendent du contexte dans lequel est conçu un programme. On pourrait faire appel à des grammaires contextuelles pour représenter de tels programmes, mais elles sont cependant plus délicates à manipuler et les processus évoqués lors de l'analyse sémantique sont difficilement automatisables. Ils dépendent de la catégorie de programmes et sont en général implémentés à la main. Nous regroupons sous le terme de *«définition dirigée par la syntaxe»* les concepts requis pendant l'analyse sémantique et par *«traduction dirigée par la syntaxe»* l'implémentation des processus qui les mettent en œuvre.

13.2 Grammaire attribuée

On utilise un formalisme qui permet d'associer des actions aux règles de production d'une grammaire. Ainsi :

- Chaque symbole de la grammaire possède des attributs. On note $X.a$ l'attribut a du symbole X. Si plusieurs symboles X figurent dans une règle, on note X le symbole lorsqu'il est en partie gauche, X_1, X_2, ..., les symboles X en partie droite, à partir du plus à gauche.
- Chaque règle de production de la grammaire possède un ensemble de règles sémantiques ou actions permettant de calculer les valeurs des attributs.
- Une règle sémantique est une suite d'instructions algorithmiques.

Définition 13.1 (Définition dirigée par la syntaxe) *On appelle Définition dirigée par la syntaxe (DDS) la donnée d'une grammaire et de règles sémantiques.*

13.2.1 Exemple

Soit la grammaire G définie par les règles suivantes :
$S \longrightarrow aSb/aS/\epsilon$
Il s'agit de déterminer le nombre de a.

Production	Règle sémantique
$S \longrightarrow aSb$	S.nba := S_1.nba + 1
$S \longrightarrow aS$	S.nba := S_1.nba + 1
$S \longrightarrow \epsilon$	S.nba := 0
$S' \longrightarrow S\$$	// S.nba contient la valeur

Dans cet exemple, on s'aperçoit que l'attribut en partie gauche est calculé en fonction de l'attribut en partie droite.

13.3 Attributs

13.3.1 Arbre décoré

Définition 13.2 (Arbre décoré) *Un arbre décoré est un arbre syntaxique sur les nœuds duquel on fait figurer à gauche et à droite du symbole ses attributs.*

La figure 13.2 représente l'arbre décoré lors de la validation de la chaîne : *aaab*. On peut calculer la valeur de l'attribut d'un nœud dès que l'attribut des fils a été calculé.

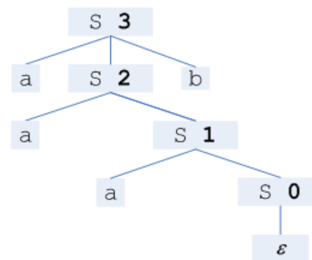


FIGURE 13.2 – Arbre décoré

L'écriture des règles sémantiques impose un ordre d'évaluation des attributs. On distingue deux sortes d'attributs : les attributs synthétisés et les attributs hérités. Dans l'arbre décoré on place à gauche du nœud les attributs hérités et à droite les attributs synthétisés.

Définition 13.3 (Attribut synthétisé) *Un attribut synthétisé est attaché au non terminal en partie gauche et se calcule en fonction des attributs des symboles de la partie droite.*

Dans l'arbre décoré, un attribut synthétisé est attaché à un nœud et se calcule en fonction des attributs de ses fils.

Définition 13.4 (Attribut hérité) *Un attribut est hérité lorsqu'il est calculé à partir des attributs du non terminal de la partie gauche et des attributs des autres symboles de la partie droite.*

Dans l'arbre décoré, un attribut hérité dépend des attributs du nœud père et des attributs des nœuds frères.

Définition 13.5 (Définition S-attribuée) *Une définition dirigée par la syntaxe n'ayant que des attributs synthétisés est appelée définition S-attribuée.*

Définition 13.6 (Définition L-attribuée) *Une définition dirigée par la syntaxe est L-attribuée si tout attribut hérité d'un symbole de la partie droite d'une production ne dépend que :*

- *. des attributs hérités du symbole en partie gauche et*
- *. des attributs des symboles le précédant dans la production.*

13.3.2 Définition formelle

Dans une définition dirigée par la syntaxe, chaque production $A \longrightarrow \alpha$ de la grammaire possède un ensemble de règles sémantiques de la forme : $b := f(c_1, c_2, \ldots c_k)$, où :

. f est une fonction,

. b est soit un attribut synthétisé de A, soit un attribut hérité d'un des symboles figurant en partie droite de la production

. les $c_{i,1\leq i\leq k}$ sont des attributs quelconques des symboles.

On considère qu'un terminal ne peut avoir que des attributs synthétisés définis à partir des caractères qui composent son lexème. Parfois, il est nécessaire que certaines règles sémantiques produisent des effets de bord. On les considère alors comme des fonctions associées à des attributs factices.

13.3.3 Ordre d'évaluation des attributs

Une DDS peut utiliser à la fois des attributs hérités et des attributs synthétisés. Il est donc nécessaire d'établir un ordre d'évaluation des règles sémantiques.

Définition 13.7 (Graphe de dépendances) *Un graphe de dépendances est un graphe orienté représentant l'ordre d'évaluation des attributs d'une grammaire attribuée. Chaque attribut constitue un sommet du graphe. Il y a un arc de a vers b si le calcul de b dépend de celui de a.*

Lorsque le graphe de dépendance contient un cycle, l'évaluation est impossible.

13.3.4 Exemple

Déclaration des variables en C :

Production	Règle sémantique
$D \longrightarrow TL$	L.type = T.val
$L \longrightarrow L, id$	$L_1.type = L.type$ setType(id.ref,L.type)
$L \longrightarrow id$	setType(id.ref,L.type)
$T \longrightarrow$ int	T.val = integer
$T \longrightarrow$ real	T.val = real

Déclaration : **real** x, y, z

13.4 Traduction dirigée par la syntaxe

Définition 13.8 (Schéma de traduction) *Un schéma de traduction est une définition dans laquelle l'ordre d'exécution des actions sémantiques est imposé. La production* $A \longrightarrow \alpha X\{action\}Y\beta$ *signifie que l'action est exécutée après que le sous-arbre issu de X a été construit et parcouru et avant que celui issu de Y ne le soit.*

On peut évaluer les attributs en même tant que l'on effectue l'analyse syntaxique. Dans ce cas, on utilise souvent une pile pour conserver les valeurs des attributs, cette pile pouvant

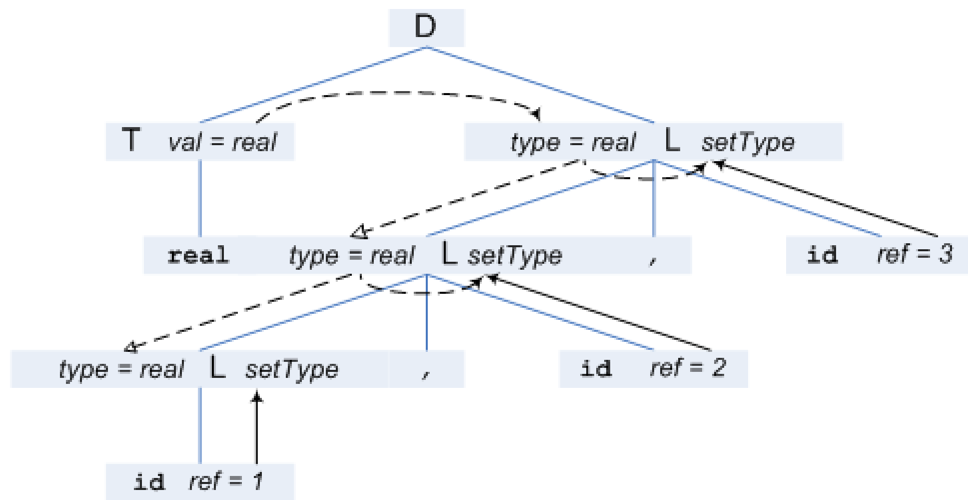


FIGURE 13.3 – Graphe de dépendances

être la même que celle de l'analyseur syntaxique, ou une autre. L'ordre d'évaluation des attributs est tributaire de l'ordre dans lequel les noeuds de l'arbre syntaxique sont créés (soit par une méthode ascendante, soit par une méthode descendante). Cependant il ne faut pas perdre de vue que l'arbre entier n'est pas réellement construit lors de l'analyse syntaxique.

13.4.1 Analyse ascendante

L'analyse ascendante se prête bien à la traduction de définitions S-attribuées puisque l'arbre syntaxique est créé des feuilles vers la racine. On empile et dépile les attributs synthétisés.
Exemple : évaluation d'une expression.

Production	Règle sémantique	Traduction
$S \longrightarrow E\$$	S.val := E.val	écrire p.depiler()
$E \longrightarrow E + T$	E.val := E_1.val+T.val	tmpT = p.depiler() tmpE = p.depiler() p.empiler(tmpE + tmpT)
$E \longrightarrow E - T$	E.val := E_1.val-T.val	tmpT = p.depiler() tmpE = p.depiler() p.empiler(tmpE - tmpT)
$E \longrightarrow T$	E.val := T.val	
$T \longrightarrow T * F$	T.val := T_1.val*F.val	tmpF = p.depiler() tmpT = p.depiler() p.empiler(tmpT x tmpF)
$T \longrightarrow F$	T.val := F.val	
$F \longrightarrow (E)$	F.val := E.val	
$F \longrightarrow nb$	F.val := nb.val	p.empiler(nb)

Analyse de 9 - 5 + 2

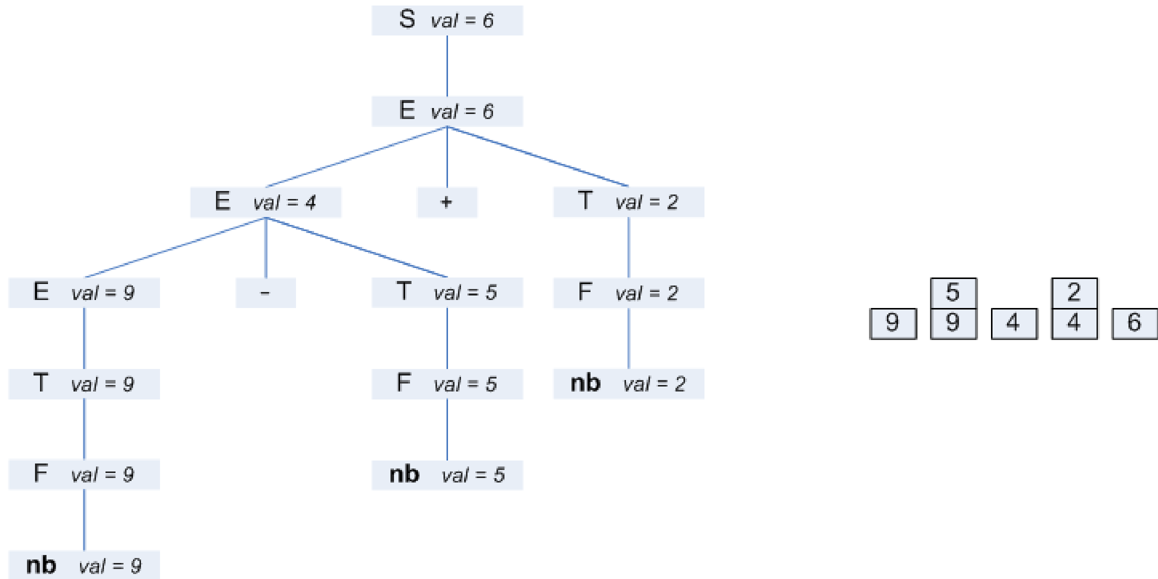


FIGURE 13.4 – Analyse ascendante

13.4.2 Analyse descendante

L'analyse descendante se prête bien à la traduction de définitions L-attribuées n'ayant que des attributs hérités puisque l'arbre syntaxique est créé de la racine vers les feuilles. On empile et dépile les attributs hérités.
Exemple : calcul du niveau d'imbrication de parenthèses.

Production	Règle sémantique	Traduction
$S' \longrightarrow S\$$	S.nb := 0	p.empiler(0)
$S \longrightarrow (S)S$	S_1.nb := S.nb + 1 S_2.nb := S.nb	tmp = p.depiler() p.empiler(tmp) p.empiler(tmp + 1)
$S \longrightarrow \epsilon$	écrire S.nb	écrire p.depiler()

Analyse de (( )) ( )

13.4.3 Principe

On insère les actions dans les règles de production comme suit :

- Un attribut hérité attaché à une variable en partie droite de règle est calculé par une action intervenant avant cette variable.
- Un attribut synthétisé attaché à la variable en partie gauche est calculé après que tous les arguments dont il dépend ont été calculés. L'action est en général placée en fin de règle.

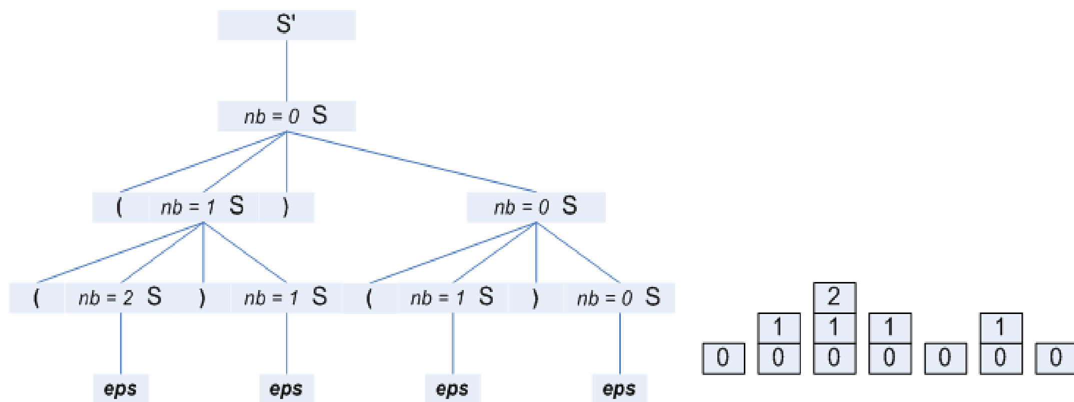


FIGURE 13.5 – Analyse descendante

- Dans les définitions L-attribuées une action ne peut faire référence à un attribut d'un symbole placé à sa droite.

13.4.4 Définition L-attribuée

L'exemple suivant représente l'évaluation d'une expression somme définie par une grammaire LL. E signifie *expression* et T signifie *terme*.

Production	Règle sémantique	Traduction
$E \longrightarrow T$	R.he := T.val	rhe = t
R	E.val := R.s	e = rs
$R \longrightarrow +T$	R_1.he := R.he + T.val	rhe = rhe + t
R	R.s := R_1.s	
$R \longrightarrow -T$	R_1.he := R.he - T.val	rhe = rhe - t
R	R.s := R_1.s	
$R \longrightarrow \epsilon$	R.s := R.he	rs = rhe
$T \longrightarrow nb$	T.val := nb.val	t = n

Production	Schéma de Traduction
$E \longrightarrow TR$	$E \longrightarrow T$ {rhe = t ;} R {e = rs ; }
$R \longrightarrow +TR$	$R \longrightarrow +T$ {rhe = rhe + t ; } R
$R \longrightarrow -TR$	$R \longrightarrow -T$ {rhe = rhe - t ;} R
$R \longrightarrow \epsilon$	$R \longrightarrow \epsilon$ {rs = rhe ;}
$T \longrightarrow nb$	$T \longrightarrow nb$ {t = n ;}

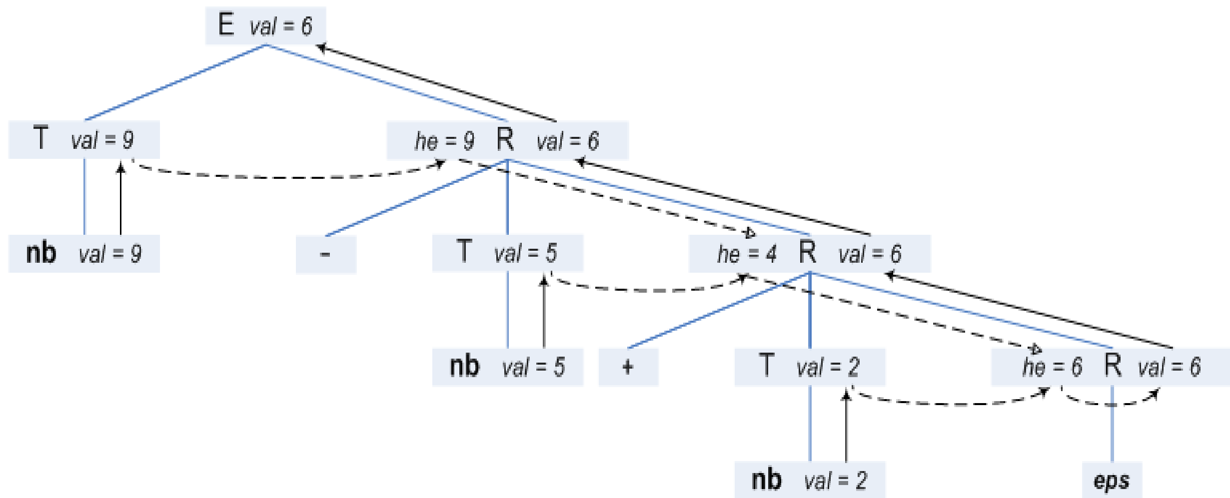


FIGURE 13.6 – Analyse L-attribuée

Analyse de 9 - 5 + 2

Ordre des actions

Action	Valeur
t = 9	
rhe = t	9
t = 5	
rhe = rhe - t	4
t = 2	
rhe = rhe + t	6
rs = rhe	6
e = rs	6

13.5 Arbre syntaxique abstrait - AntLR (v 3)

Une définition dirigée par la syntaxe peut être utilisée pour construire un arbre syntaxique abstrait représentant un programme satisfaisant une grammaire. Dans sa version 3, le générateur AntLR [1] peut construire cet arbre lors de l'analyse syntaxique. Il faut pour cela ajouter des symboles dans la grammaire.

Le symbole $\wedge$ placé derrière une catégorie lexicale indique que ce token est considéré comme racine intermédiaire d'une arborescence de l'arbre abstrait. Un nœud est construit pour ce token et devient la racine de toute portion de l'arbre déjà construite.

Les feuilles de l'arbre syntaxique sont automatiquement insérées dans l'arbre abstrait sauf si elles sont suivies du symbole !. Si A, B et C représentent des catégories lexicales :

A B^ C   donne l'arbre ^(B A C).

1. Terence Parr, Université de San Francisco, http ://www.antlr.org/

B est la racine, A et C sont les fils.
 A B^ C^   donne l'arbre ^(C ^(B A)).

C est la racine et a un seul fils dont la racine est B et qui a pour fils A.

13.5.1 Exemple : expression arithmétique

La grammaire suivante permet de décrire des expressions arithmétiques sur des entiers. Les symboles ^ permettent de construire l'arbre abstrait d'une expression.

```
grammar Expr;
options {
  output = AST;
}
tokens{
  PLUS = '+';
  MOINS = '-';
  MULT = '*';
  DIV = '/';
  EXP = '^';
}
expr : sumExpr ;
sumExpr  : prodExpr ((PLUS^|MINUS^) prodExpr)* ;
prodExpr : powExpr ((MUL^|DIV^) powExpr)* ;
powExpr  : atom (EXP^ atom)? ;
atom     : INT ;

INT :   '0'..'9'+ ;
```

13.5.2 Arbre abstrait

A partir de l'expression `10 - 5 + 4 * 20 / 2`, l'environnement AntLR permet de construire l'arbre abstrait suivant :

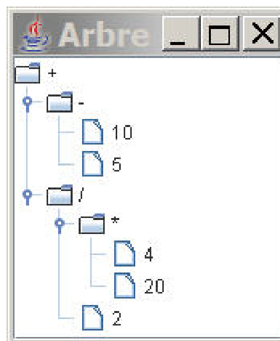


Figure 13.7 – Arbre abstrait

13.5.3 Evaluation récursive

AntLR version 3 permet de définir des grammaires sur des arbres abstraits. La grammaire suivante permet d'évaluer récursivement l'expression de la section précédente. Le résultat en est 45.0.

```
tree grammar ExprEval;
options {
 tokenVocab = Expr;
 ASTLabelType=CommonTree;
}

exp returns[double v] :
   ^(PLUS x=exp y=exp)  {$v = $x.v + $y.v;}
 | ^(MOINS x=exp y=exp) {$v = $x.v - $y.v;}
 | ^(MULT x=exp y=exp)  {$v = $x.v * $y.v;}
 | ^(DIV x=exp y=exp)   {$v = $x.v / $y.v;}
 | ^(EXP x=exp y=exp)   {$v = Math.exp($y.v * Math.log($x.v));}
 | INT {$v = Double.parseDouble($INT.text);}
;
```

13.6 Traduction de langage

Traduction de texte écrits en langage csv (comma separated values) en langage xhtml.

Règles de production :

```
grammar CSV;
COMMA   :  ',' ;
RECORD  :  ('A'..'Z' | 'a'..'z' |'0'..'9')+
      ;
NEWLINE :  '\r'? '\n' ;
WS      :  (' '|'\t')+ {skip();} ;
file    : line (NEWLINE line)* NEWLINE? EOF
line    : record+ ;
record  : RECORD COMMA? ;
```

Affichage de la trace d'analyse :

Les règles sont complétées par des actions écrites dans le langage de programmation désiré. Elles indiquent ici les règles appelées et les records trouvés.

```
grammar CSV;
...
file
: {System.out.println("file called");}
  line (NEWLINE line)* NEWLINE? EOF
```

```
  {System.out.println("file matched");}
;
line :
  {System.out.println("line called");}
  record+
  {System.out.println("line matched");}
;
record  :
  {System.out.println("record called");}
  r=RECORD COMMA?
  {System.out.println("record = " + $r.getText());
   System.out.println("record matched");}
;
```

Affichage xHTML :

En substituant les actions, il est possible d'afficher une table xHTML.

```
grammar CSV;
...
file
: {System.out.println("<table align=\"center\" border=\"1\">");}
  line (NEWLINE line)* NEWLINE? EOF
  {System.out.println("</table>");}
;
line :
  {System.out.println(" <tr>");}
  record+
  {System.out.println(" </tr>");}
;
record  :
  {System.out.print(" <td>");}
  r=RECORD COMMA?
  {System.out.print($r.getText());
   System.out.println("</td>");}
;
```

Méthode retournant une chaîne :

Il est possible de modifier les règles pour que les méthodes correspondantes construisent une chaîne réutilisable.

```
grammar CSV;
@members{
  StringBuilder sb = new StringBuilder();
}
...
```

```
file returns [String s] :
  {sb.append("<table align=\"center\" border=\"1\"> \n");}
  line (NEWLINE line)* NEWLINE? EOF
  {sb.append("</table>\n"); s = sb.toString();}
;
line :
  {sb.append(" <tr>\n");}
  record+
  {sb.append(" </tr>\n");}
;
record  :
  {sb.append(" <td>");}
  r=RECORD COMMA?
  {sb.append($r.getText());
   sb.append("</td>\n");}
;
```

13.7 AntLR version 4

13.7.1 Evolution d'AntLR

L'avantage de la version 3 d'AntLR est la construction automatique d'un arbre syntaxique abstrait (AST) où ne figurent que des tokens. Il faut alors écrire une grammaire d'arbre qui permet d'exploiter cet arbre. Un inconvénient est qu'il faut insérer dans cette grammaire des actions dans un langage cible, ce qui dénature la grammaire. De plus, ces actions doivent tenir compte de la représentation interne par AntLR de cette structure d'arbre.
Dans la version 4 d'AntLR, le parti pris d'AntLR est de plus construire d'arbre syntaxique abstrait mais de se contenter de l'arbre syntaxique concret, c'est à dire de l'arbre de dérivation. AntLR permet cependant de générer des classes implémentant un visiteur d'arbre. Les actions permettant d'exploiter la structure d'arbre sont ainsi externalisées et la gammaire reste intacte. Un visiteur d'abre externe permet beaucoup plus facilement d'implémenter des possibilités de visite comme le fait de ne pas visiter une branche ou de visiter plusieurs fois une branche.

13.7.2 Grammaire d'expression

```
grammar Expr;
@header {
  package ...;
}
INT :   '0'..'9'+ ;
WS  :   (' '|'\t')+ {skip();} ;
expr:    sumExpr ;
multExpr:
    powExpr (('*'|'/') powExpr)* ;
```

```
powExpr:
    atom ('^' atom)* ;
atom:
    INT
 |  '(' expr ')'
 ;
```

A partir de la grammaire d'expression précédente AntLR génére la structure d'arbre suivante à partir de l'expression : 5 * (10 - 2)

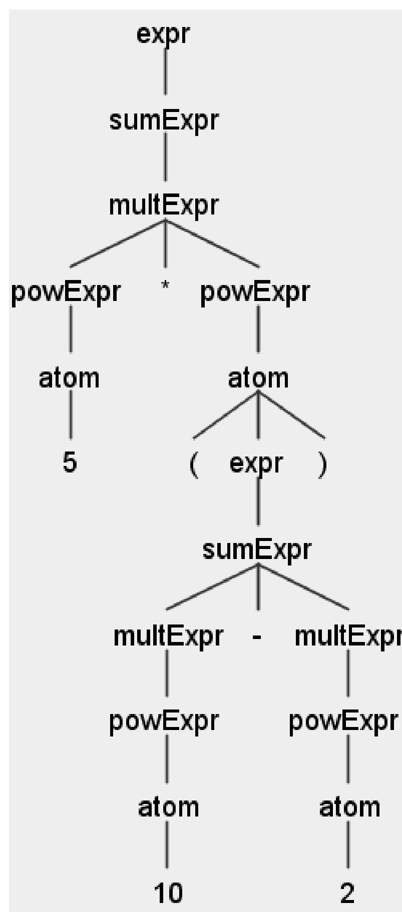


FIGURE 13.8 – Arbre de dérivation

13.7.3 Nouvelles caractéristiques d'AntLR

Bien que AntLR soit un analyseur descendant, les apports théoriques récents permettent de s'affranchir des inconvénient majeurs de ces analyseurs :

- Il est possible d'écrire des grammaires non factorisées à gauche ;
- Il est possible d'écrire des règles directement récursives à gauche ;

– Il est possible d'écrire des grammaires non ambigües en ordonnant les alternatives d'une règle.

Il est possible de ré-écrire de la façon suivante la grammaire d'expression :

```
grammar Expr;
@header {
  package ...;
}
INT :   '0'..'9'+ ;
WS  :   (' '|'\r'|'\t'|'\n')+ {skip();} ;
prg:
  expr
  ;
expr:
     expr '^' expr
  |  expr ('*' | '/' ) expr
  |  expr ('+' | '-' ) expr
  |  INT
  |  '(' expr ')'
;
```

AntLR considère l'ordre des règles pour établir les priorités entre les opérateurs. Ainsi une expression telle que 2 + 5 * 10 donnera l'arbre de dérivation qui permettra de l'évaluer à 52.

Bibliographie

[1] A. Aho, R. Sethi, and J. D. Ullman. *Compilateurs Principes, techniques et outils.* InterEditions, 1991.

[2] A. Aho and J. D. Ullman. *Concepts fondamentaux de l'Informatique.* Dunod, 1993.

[3] A. W. Appel. *Modern compiler implementation in java.* Cambridge, 2002.

[4] Jeffrey E.F. Friedl. *Maîtrise des Expressions Régulières.* O'Reilly, 2003.

[5] D. Grune, H. E. Bal, C. J. H. Jacobs, and K. G. Langendoen. *Compilateurs, Cours et exercices corrigés.* Dunod, 2002.

[6] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 2001.

[7] Terence Parr. *The Definitive ANTLR Reference.* The Pragmatic Programmers, 2007.

[8] Terence Parr. *Language Implementation Patterns.* The Pragmatic Programmers, 2010.

[9] Terence Parr. *The Definitive ANTLR 4 Reference.* The Pragmatic Programmers, 2012.

Annexe A

Générateurs d'analyseur

A.1 Commandes unix

lex (Lexical Analyzer Generator)
flex : version gnu de lex
yacc (Yet Another Compiler-Compiler)

Exemple de spécifications flex

Principe : {spécifications flex} $\longrightarrow$ flex $\longrightarrow$ programme C

```
%{
int nbVoyelles, nbConsonnes, nbPonct;
%}
consonne      [b-df-hj-np-xy]
ponctuation   [,;:?!\.]
%%
[aeiouy]      nbVoyelles++;
{consonne}    nbConsonnes++;
{ponctuation} nbPonct++;
\n            // ne rien faire
%%
main() {
  nbVoyelles = nbConsonnes = nbPonct = 0;
  yylex();
  printf(``Nb voyelles : %d - Nb consonnes : %d - Ponctuation : %d``,
          nbVoyelles, nbConsonnes, nbPonct);
}
```

A.2 Générateurs en langage Java

- JLex : Princeton University
  - http ://www.cs.princeton.edu/∼appel/modern/java/JLex/
- javaCC : https ://javacc.dev.java.net/
- AntLR : http ://www.antlr.org/

Exemple javacc

Principe : {spécifications javacc} ⟶ javacc ⟶ fichiers .java

Exemple de lexer reconnaissant des identificateurs ou des entiers

```
PARSER_BEGIN(Parser)
package spec;
   class Parser{
     public static void main(String args[]) throws ParseException {
      java.util.Scanner s = new java.util.Scanner(System.in);
      System.out.println("Write a string");
      String s = scanner.nextLine();
      Parser p = new Parser(new StringReader(s));
      p.Input();
     }
   }
PARSER_END(Parser)
TOKEN : {
     < #CAR: ["a"-"z","A"-"Z"] >
   | < #DIGIT: ["0"-"9"] >
   | < ID: <CAR>(<CAR>|<DIGIT>)* >
   | < #INTPLUS: (["0"-"9"])+ >
   | < NUM: (<INTPLUS>"."<INTPLUS>)  |
          (<INTPLUS>".") >
}
void Input() :
{Token t = null;}
{t = <ID>
   {System.out.println("Token: " + t.image + "type: " + t.kind);}
 | t = <NUM>
   {System.out.println("Token: " + t.image + "type: " + t.kind);}
}
```

Annexe B

Le générateur AntLR

Cette annexe décrit l'utilisation de la version 3 et de la version 4 du générateur de parser AntLR [1].

B.1 AntLR version 3

B.1.1 Exemple simple

Introduction

Pour réaliser les analyses lexicale et syntaxique, il faut spécifier une grammaire (Lexer, Parser) qui décrit la façon de regrouper le flux de caractères en un flux de tokens puis d'analyser le flux de tokens sortant. Il suffit de créer un fichier d'extension .g définissant les unités lexicales et syntaxiques de la grammaire. AntLR créera de lui-même un fichier annexe contenant la grammaire du lexer. On peut ajouter à la grammaire des actions regroupant des lignes de code du même langage que celui utilisé pour le code généré pour les lexer et parser. Les unités lexicales sont écrites en majuscules. Une règle du parser est utilisée comme axiome de la grammaire. AntLR génère les classes correspondant au lexer et au parser.

Grammaire

```
grammar SimpleGrammar;
ALPHA
    : ('a'..'z'|'A'..'Z')+
        {System.out.println("Found alpha: "+getText());}
;
NUMERIC
    : '0'..'9'+
         {System.out.println("Found numeric: "+getText());}
;
EXIT : '.' {System.exit(0);};
prog  : ALPHA NUMERIC EXIT ;
```

1. Site AntLR : http ://www.antlr.org

B.1.2 Mise en œuvre

Génération des classes en ligne de commande

Il faut exécuter AntLR sur le fichier SimpleGrammar.g qui contient la description des unités lexicales. Le nom du fichier doit être identique au nom de la grammaire. A partir de l'exemple précédent, les classes créées porteront les noms : `SimpleGrammarLexer` et `SimpleGrammarParser`.

Après avoir mis à jour la variable `PATH` (pour accéder à Java - version 1.4 ou supérieure) et la variable `CLASSPATH` (les librairies utiles sont disponibles sur le site AntLR : `antlr-3.0.jar`, `stringtemplate-3.0.jar` et `antlr-2.7.7.jar`), il suffit d'exécuter une ligne de commande telle que :

```
java org.antlr.Tool SimpleGrammar.g
```

Exemple de fichier batch sous Windows

On suppose Java, installé sur la machine hôte. Dans un répertoire on copie le fichier de grammaire (SimpleGrammar.g). On crée un répertoire lib contenant les librairies java requises pour le fonctionnement de AntLR (antlr-3.0.jar, stringtemplate-3.0.jar, antlr-2.7.7.jar). On crée au même niveau le fichier .bat suivant :

```
set SAVECLASSPATH=%CLASSPATH%
set CLASSPATH=../lib/antlr-3.0.jar;../lib/stringtemplate-3.0.jar;
                                   ../lib/antlr-2.7.7.jar
java org.antlr.Tool SimpleGrammar.g
pause
set CLASSPATH=%SAVECLASSPATH%
```

Outil ANTLRWorks

Il est recommandé d'écrire la grammaire avec l'outil ANTLRWorks [2]. Cet éditeur possède de nombreuses fonctionnalités et en particulier génère les classes correspondant à la grammaire. Il est possible de configurer cet outil de façon à ce que les classes générées soient accessibles à partir d'un environnement de développement tel qu'Eclipse [3], où l'on peut visualiser les erreurs de compilation dues au code inséré dans les actions. ANTLRWorks permet d'indiquer l'endroit où les classes sont générées et le package qu'elles respectent. La figure B.1 montre l'interface de l'outil ANTLRworks à partir de la grammaire de la section B.1.3. Elle montre également l'arbre de dérivation de l'expression : $ab = 5 \times 4 + 1$.

Exécution

Programme principal. On peut ajouter les lignes suivantes à la grammaire juste après la déclaration du nom de la grammaire pour définir les packages des classes créées :

```
@header {
  package annexeb;
}
@lexer::header {package annexeb;}
```

2. Site ANTLRWorks : http ://www.antlr.org/works/
3. Site Eclipse : http ://www.eclipse.org

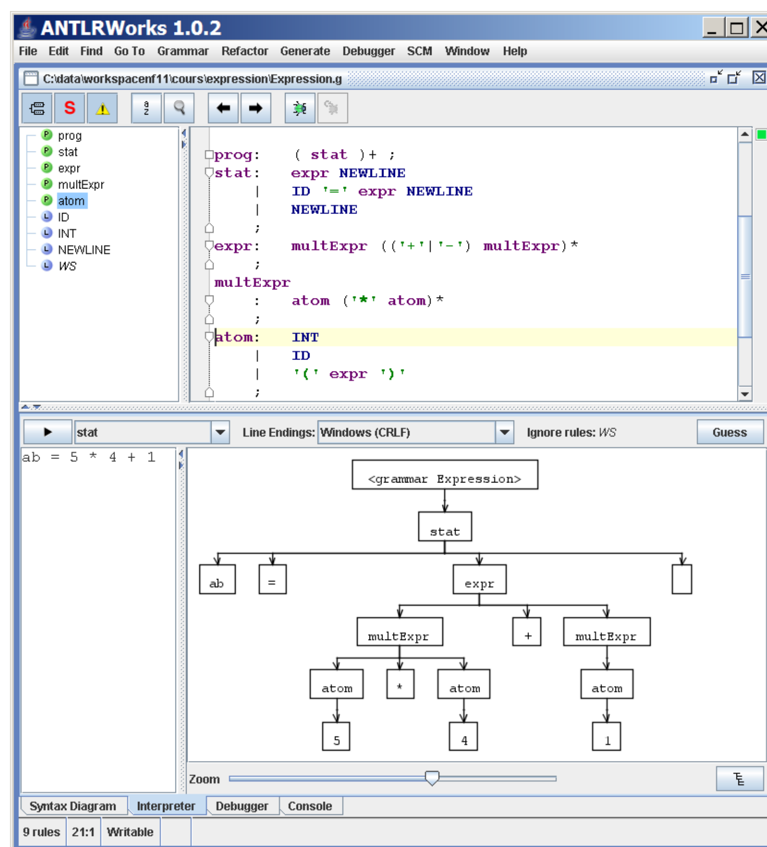


FIGURE B.1 – Interface de ANTLRWorks

Pour utiliser les classes créées, il faut les instancier. On peut par exemple dans le même package créer une classe contant une méthode main telle que :

```
package annexeb;
import java.io.ByteArrayInputStream;
import java.util.Scanner;
import org.antlr.runtime.ANTLRInputStream;
import org.antlr.runtime.CommonTokenStream;
public class SimpleGrammarMain {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Texte : ");
    String s = scanner.nextLine();
    ByteArrayInputStream bais = new ByteArrayInputStream(s.getBytes());
    try {
      ANTLRInputStream input = new ANTLRInputStream(bais);
      SimpleGrammarLexer lexer = new SimpleGrammarLexer(input);
      CommonTokenStream tokens = new CommonTokenStream(lexer);
      SimpleGrammarParser parser = new SimpleGrammarParser(tokens);
      parser.prog();
    }
    catch(Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

La saisie d'un exemple se fait au clavier après l'invite Texte. La méthode crée un flux de caractères qui est associé au lexer. Un flux de tokens est créé à partir du lexer qui est associé au parser. Il suffit alors d'appeler la méthode correspondant à l'axiome de la grammaire. Selon celle-ci, il suffit de taper deux fois une suite de lettres et une suite de chiffres puis un point. Le point fait sortir du programme. Le programme affiche les lettres et les chiffres tapés.

Fonctionnement. AntLr a créé une classe lexer et une classe parser dans le package spécifié (dans l'exemple appelées `SimpleGrammarLexer` et `SimpleGrammarParser`). Les unités lexicales permettent d'ajouter des méthodes de la classe lexer. Ainsi l'unité `ALPHA` sera associée à la méthode `mALPHA()` permettant d'analyser une suite de lettres. Les règles du parser sont associées à des méthodes de la classe parser : ainsi la règle `prog` donne une méthode `prog()`.

Méta-langage AntLr.

( )	Règle
( )*	0 ou plus
( )+	1 ou plus
( ) ?	0 ou 1
|	Alternative
..	Intervalle
~	Négation
.	caractère quelconque

B.1.3 Autre exemple

La grammaire suivante permet d'écrire des expressions numériques. Les règles du parser sont en minuscule ; celles du lexer en majuscule. Lexer et parser ne sont pas séparés.

Grammaire

```
grammar Expression;

@lexer::header {
  package expression;
}

@header {
  package expression;
}

prog:   stat+ ;

stat:   expr NEWLINE
    |   ID '=' expr NEWLINE
    |   NEWLINE
    ;

expr:   multExpr (('+'|'-') multExpr)*
    ;

multExpr
    :   atom ('*' atom)*
    ;

atom:   INT
    |   ID
    |   '(' expr ')'
    ;

ID  :   ('a'..'z'|'A'..'Z')+ ;
INT :   '0'..'9'+ ;
NEWLINE:'\r'? '\n' ;
WS  :   (' '|'\t')+ {skip();} ;
```

`expression.ExpressionLexer` et `expression.ExpressionParser` sont les deux classes créées par AntLR. La méthode associée à l'axiome de la grammaire est `prog`.

Exécution

L'exécution est réalisée à partir d'un fichier contenant des expressions par une méthode main telle que :

```
public static void main(String[] args) {
  try {
    FileInputStream fis =
```

```
        new FileInputStream("expression/expression.prg");
    ANTLRInputStream input = new ANTLRInputStream(fis);
    ExpressionLexer lexer = new ExpressionLexer(input);
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    ExpressionParser parser = new ExpressionParser(tokens);
    parser.prog();
  }
  catch(Exception ex) {
    ex.printStackTrace();
  }
}
```

A ce stade, le parser n'a pas de sortie et n'indique pas les expressions mal construites. Il serait nécessaire de compléter la grammaire en vue d'une utilisation particulière du parser.

B.1.4 Création de l'arbre AST

Création par défault

AntLR donne la possibilité de créer automatiquement, semi-automatiquement ou manuellement l'arbre AST, structure intermédiaire d'un programme compatible avec la grammaire. Il faut tout d'abord ajouter une section `options` indiquant au parser de créer l'arbre :

```
grammar Expression;
options {
  language = Java;
  output = AST;
}
```

Les méthodes du parser associées aux règles de la grammaire ne sont alors plus de type `void` mais d'un type structuré dont le nom reprend celui de la méthode. Ainsi la méthode `stat` sera de type `stat_return` sous-classe de `ParserRuleReturnScope`.

Pour chaque règle, AntLR crée une racine d'arborescence. Pour chaque token présent dans une règle, il crée un noeud (Tree) et l'ajoute comme fils du noeud courant. Par défaut les noeuds forment une liste et antlr attache les noeuds de la liste à un noeud racine de type `nil`.

Création automatique

Pour ne pas créer un noeud pour un token particulier, il suffit de le suffixer avec un `!`. Pour créer un noeud racine à partir d'un token particulier, il suffit de le suffixer avec un `^`. A partir des règles du parser suivantes :

```
prog:   stat+ ;
stat:   expr NEWLINE!
    |   ID '='^ expr NEWLINE!
    |   NEWLINE!
    ;
expr:   multExpr (('+'^|'-'^) multExpr)*
```

```
    ;
multExpr
    :   atom ('*'^ atom)*
    ;
atom:   n=INT {System.out.println("Entier : " + $n.text);}
    |   ID
    |   '('! expr ')'!
    ;
```

et avec le programme suivant :

```
25 + 3 * 4
ab = 10 * (5 - 2)
```

On obtient l'arbre suivant qui est une liste :

```
(+ 25 (* 3 4)) (= ab (* 10 (- 5 2)))
```

Code

```
public static void mainAST() {
  try {
    FileInputStream fis =
        new FileInputStream("expression/expression.prg");
    ANTLRInputStream input = new ANTLRInputStream(fis);
    ExpressionLexer lexer = new ExpressionLexer(input);
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    ExpressionParser parser = new ExpressionParser(tokens);
    prog_return r = parser.prog();
    CommonTree t = (CommonTree) r.getTree();
    System.out.println(t.toStringTree());
  }
  catch(Exception ex) {
    ex.printStackTrace();
  }
}
```

Affichage de l'arbre

En utilisant les classes `ASTFrame` et `ASTtoTreeModelAdapter` écrites par J. Werner, il est possible de créer une fenêtre présentant l'arbre : Il suffit d'ajouter les deux lignes suivantes au code précédent :

```
ASTFrame af = new ASTFrame("Arbre", t);
af.setVisible(true);
```

Création manuelle de l'arbre

Il est parfois plus simple d'indiquer directement l'arborescence qui doit être construite à partir d'une règle. La génération de l'arbre peut être décrite dans la grammaire en utilisant `->` :

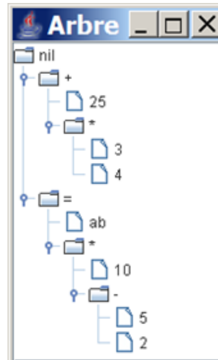


FIGURE B.2 – Arbre AST

```
prog:   stat+ ;
stat:   expr NEWLINE          -> expr
    |   ID '=' expr NEWLINE -> ^('=' ID expr)
    |   NEWLINE               ->
    ;
expr:   multExpr (('+'^|'-'^) multExpr)*
    ;
multExpr
    :   atom ('*'^ atom)*
    ;
atom:   n=INT {System.out.println("Entier : " + $n.text);}
    |   ID
    |   '(' expr ')' -> expr
    ;
```

On peut insérer des noeuds imaginaires facilitant l'analyse ultérieure :

```
prog:   stat+ -> ^(PROG stat+)
```

`PROG` est simplement déclaré comme un token non défini par une expression régulière.

B.1.5 Actions AntLR

Il est possible d'insérer des actions écrites dans le langage cible à tout endroit de la grammaire.

Actions du parser

Il est possible de réutiliser les tokens issus du lexer.

```
atom:   n=INT {System.out.println("Entier : " + $n.text);}
    |   ID
    |   '(' expr ')'
    ;
```

Dans la règle précédente `n` est un objet de type Token. On le réutilise en écrivant `$n` dans le code de l'action. `text` est un attribut de l'objet Token.

Variables et méthodes globales

La section `@members` permet de définir des méthodes et des variables d'instance de l'objet parser créé. Elles sont appelées dans le code sans ajouter de symbole particulier. La communication d'information entre les règles appelées lors d'une analyse est ainsi simplifiée. Une règle peut avoir besoin de variables locales. Elles sont alors déclarées et initialisées dans la section `@init` de la règle. L'exemple suivant montre l'usage de ces divers éléments.

```
grammar CSVString;
@header{
  package csv;
}
@lexer::header{
  package csv;
}
@members{
  StringBuilder sb = new StringBuilder();
}
COMMA  :  ',' ;
RECORD  :  ('A'..'Z' | 'a'..'z' |'0'..'9')+
;
NEWLINE :  '\r'? '\n' ;
WS      :  (' '|'\t')+ {skip();} ;

file returns [String s] :
  {sb.append("<table align=\"center\" border=\"1\"> \n");}
  line (NEWLINE line)* NEWLINE? EOF
    {sb.append("</table>\n"); s = sb.toString();}
;
line
@init {String tagin = " <tr>\n", tagout = " </tr>\n";}:
  {sb.append(tagin);}
  record+
  {sb.append(tagout);}
;
record  :
  {sb.append(" <td>");}
  r=RECORD COMMA?
  {sb.append($r.getText());
   sb.append("</td>\n");}
;
```

B.2 AntLR version 4

B.2.1 Exemple : grammaire d'expressions arithmétiques

Introduction

Pour réaliser les analyses lexicale et syntaxique, il faut spécifier une grammaire (Lexer, Parser) qui décrit la façon de regrouper le flux de caractères en un flux de tokens puis d'analyser le flux de tokens sortant. Il suffit de créer un fichier d'extension .g4 définissant les unités lexicales et syntaxiques de la grammaire.
Dans sa génération en langage Java, AntLR créée un ensemble de classes permettant l'utilisation de l'arbre créé par le parser. Ainsi à partir d'une grammaire appelée Expr, AntLR créée :

1. le Lexer : `ExprLexer.java`
2. le Parser : `ExprParser.java`
3. les Listes de tokens : `Expr.tokens ; ExprLexer.tokens`
4. les Listeners :
   `ExprListener.java` (interface)
   `ExprBaseListener.java` (classe implémentant à vide l'interface)
5. les Visitors :
   `ExprVisitor.java` (interface)
   `ExprBaseVisitor.java` (classe implémentant à vide l'interface)

Grammaire

La grammaire suivante est reconnue par AntLR bien que non conforme au modèle LL grâce à des algorithmes nouveaux issus de la recherche.

```
grammar Expr;
@header {
  package expression;
}
INT :   '0'..'9'+ ;
WS  :   (' '|'\r'|'\t'|'\n')+ {skip();} ;
prg:
  expr
  ;
expr:
     expr '^' expr           #exp
  |  expr ('*' | '/' ) expr  #mult
  |  expr ('+' | '-' ) expr  #sum
  |  INT                     #int
  |  '(' expr ')'            #parent
;
```

Cette grammaire permet d'analyser des expressions. La priorité des opérateurs et l'associativité à gauche sont préservées grâce à l'ordre des alternatives dans la règle `expr`.

L'utilisation des compléments tels que `sum` est expliquée plus loin. A partir de la grammaire d'expression précédente AntLR génére la structure d'arbre suivante à partir de l'expression : 5 * 3 / 2 * (15 - 2)

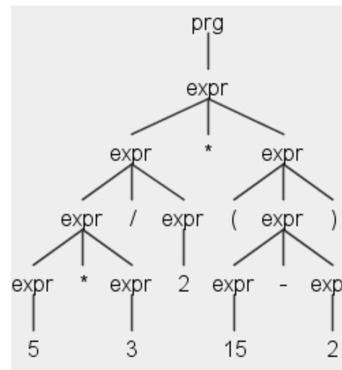


FIGURE B.3 – Arbre de dérivation

B.2.2 Mise en œuvre

Génération des classes en ligne de commande

Il faut exécuter AntLR sur le fichier Expr.g4 qui contient la grammaire. Le nom du fichier doit être identique au nom de la grammaire. Avec la commande java accessible il suffit de créer la commande suivante dans un fichier batch (version Windows) pour obtenir l'ensemble des fichiers (listeners et visitors) :

```
@echo off
set SAVECLASSPATH=%CLASSPATH%
set CLASSPATH=lib/antlr-4.0-rc-1-complete.jar;
java org.antlr.v4.Tool -visitor -o src/expression
                           src/expression/Expr.g4
pause
set CLASSPATH=%SAVECLASSPATH%
```

Les fichiers créés se retrouveront dans le répertoire suivant l'option `-o` et la grammaire utilisée correspond au dernier paramètre.
Une fois les classes compilées, il est possible de visualiser l'arbre syntaxique d'une expression grâce à la commande suivante :

```
@echo off
set SAVECLASSPATH=%CLASSPATH%
set CLASSPATH=lib/antlr-4.0-rc-1-complete.jar;./bin
java org.antlr.v4.runtime.misc.TestRig expression.Expr prg -gui
                                                programs/expr2.txt
pause
set CLASSPATH=%SAVECLASSPATH%
```

Le premier paramètre est la grammaire, le second l'axiome de la grammaire, le troisième (`-gui`) indique la création de la fenêtre et le dernier indique le fichier où se trouve l'expression à analyser.

Programme minimal pour listener

```
public static void run_listener() {
 try {
  // lexer
  InputStream is = new FileInputStream(<program>);
  ANTLRInputStream stream = new ANTLRInputStream(is);
  ExprLexer lexer = new ExprLexer(stream);
  // parser
  CommonTokenStream tokens = new CommonTokenStream(lexer);
  ExprParser parser = new ExprParser(tokens);

  ParseTree tree = parser.prg();
  System.out.println("Arbre syntaxique :");
  System.out.println(tree.toStringTree(parser));
  ParseTreeWalker  walker  =  new  ParseTreeWalker();
  walker.walk(new  ExprEvalListener(),  tree);
 } catch(Exception ex) {
    ex.printStackTrace();
 }
}
```

La classe `ExprEvalListener` est une classe qu'il faut écrire pour répondre au parcours de l'arbre syntaxique créé (`tree`). Elle doit hériter de la classe `ExprBaseListener` générée par AntLR.

Programme minimal avec visitor

```
public static void run_antlr_visitor() {
 try {
  // lexer
  InputStream is = new FileInputStream(<programme>);
  ANTLRInputStream stream = new ANTLRInputStream(is);
  ExprLexer lexer = new Expr3Lexer(stream);
  // parser
  CommonTokenStream tokens = new CommonTokenStream(lexer);
  ExprParser parser = new ExprParser(tokens);

  ParseTree tree = parser.prg();
  System.out.println(tree.toStringTree(parser));

  ExprEvalVisitor visitor = new  ExprEvalVisitor();
  Double result = visitor.visit(tree);
```

```
  System.out.println("Result = " + result);
 } catch(Exception ex) {
    ex.printStackTrace();
 }
}
```

La classe `ExprEvalVisitor` est une classe qu'il faut écrire pour visiter l'arbre syntaxique créé (`tree`) et effectuer le traitement désiré. Elle doit hériter de la classe de base générée par AntLR, `ExprBaseVisitor`. Dans le cas de l'exemple, la méthode `visit` retourne un élément de type `Double`.

B.2.3 Principe du visitor

La méthode `visit` d'un visitor déclenche la visite sur chacun des fils de l'arbre. AntLR créée une méthode de visite spécifique pour chaque nœud de l'arbre de dérivation. Il suffit d'indiquer ce que doit faire chacune de ces méthodes. Un nœud de l'arbre correspond à l'application d'une règle de la grammaire. Cependant dans le cas d'une règle possédant l'alternative de plusieurs possibilités, il faut nommer chaque cas (ex : `#sum`, `#mult`) et AntLR génére une méthode du visitor pour chacun de ces cas. Chaque méthode prend en paramètre un contexte local contenant toutes les informations nécessaires à la visite d'un nœud.

Ainsi la méthode suivante relance la visite sur le nœud fils :

```
public Double visitPrg(PrgContext ctx) {
  return visit(ctx.expr());
}
```

La méthode suivante est appelée lorsque la visite d'une arborescence rencontre un nœud somme de deux expressions :

```
public Double visitSum(SumContext ctx) {
 Double left = visit(ctx.expr(0));
 Double right = visit(ctx.expr(1));
 Double result =  ctx.getChild(1).getText().equals("+") ?
            left + right : left - right;
 return result;
}
```

Le contexte permet de retrouver les deux nœuds fils `ctx.expr(0)` et `ctx.expr(1)`. L'option a été ici de retourner la somme des deux sous-expressions comme valeur de retour de la méthode. D'autres façons de faire sont évidemment possibles.

Annexe C

Introduction à Java 5.0

C.1 Introduction

C.1.1 Objectifs

L'objectif de cette annexe est d'introduire les principales notions de la programmation objet et les principaux éléments de la syntaxe Java. La version de Java utilisée est la version 5.0 Tiger. Il ne s'agit pas d'une liste exhaustive des caractéristiques de java. Les exemples sont volontairement choisis en direction de la théorie des langages. Introduction de façon linéaire

C.1.2 Environnements de développement

Il y a deux façons d'écrire des programmes Java et de les exécuter :

- Sans utiliser d'environnement de développement
  - Ecrire le code dans un simple éditeur de texte
  - Compiler et exécuter le code en ligne de commande
- En utilisant un environnement de développement
  - Les environnements les plus utilisés sont :
    - Eclipse : http ://www.eclipse.org/
    - Netbeans : http ://www.netbeans.com
    - Borland JBuilder : http ://www.borland.com/jbuilder
  - Avantages
    - Compilation, exécution, debugging dans un seul environnement
    - Complétion du code source
    - Possibilité d'augmenter l'environnement

Dans cette initiation, tous les exemples sont supposés créés dans l'environnement Eclipse, version 3.2. Certaines actions décrites comme obligatoires sont en réalité parfois simplement conseillées, mais une bonne programmation orientée objet les requiert comme telles.

C.2 Programmation orientée objet

C.2.1 Classes et Objets

L'unité de base est l'objet. Une application est un ensemble d'objets communiquant entre eux. Un objets contient notamment des données (appelées attributs) et stockés dans les variables d'instance. Un objet peut effectuer une série d'actions (traditionnellement nommée méthodes) utilisant les données internes de l'objets et des paramètres.
A un moment donné de l'exécution d'un programme, chaque attribut a une valeur. L'ensemble des valeurs des attributs de l'objet détermine l'état courant de l'objet. Cet état ne peut changer que si une méthode de l'objet est invoquée.
Il faut commencer par définir les catégories des objets (leurs classes) en décrivant leurs attributs et leurs méthodes. Le concept permettant cela est celui de classe. Une classe décrit une méthode particulière destinée à construire un objet correspondant, une instance de cette classe.
Une classe doit être également considérée comme une entité faisant partie d'un programme. Elle peut contenir des attributs, stockés dans des variables de classe et des méthodes de classe. A cet égard, il faut concevoir une classe comme une instance de la classe Class. Pour différencier les champs des objets de ceux de la classe ces derniers sont précédés du modificateur static. L'écriture d'une classe comporte donc :

- la déclaration des variables des instances de cette classe
- la déclaration des méthodes des instances de cette classe
- la déclaration des variables de cette classe
- la déclaration des méthodes de cette classe
- la déclaration des méthodes constructeurs des instances de cette classe

C.2.2 API Java

Java définit un ensemble de classes prêtes à être utilisées. Elles constituent l'API java (Application Programming Interface) c'est-à-dire l'ensemble des ressources utilisables par toute application. Il est intéressant de consulter régulièrement la description de cette API (http ://java.sun.com/j2se/1.5.0/docs/api/index.html). Les classes sont regroupées en packages ayant une finalité particulière. Par exemple les packages suivant sont fréquemment utilisés :

- java.io : Classes pour le système d'entrée/sortie (ex : File)
- java.lang : Classes fondamentales pour la conception (ex : Math, String, System)
- java.util : Classes collections de structures, classes utilitaires diverses (ex : Vector, Scanner)
- javax.swing : Interfaces graphiques (ex : Frame)

Le nom complet d'une classe est composé du nom du package auquel elle appartient suivi du nom local. Une classe est encore appelée le type de ses instances.

C.2.3 Entrées-Sorties sur la console

L'entrée standard est un objet de type `InputStream` et la sortie standard est un objet de type `PrintStream`. Ce sont les variables `in` et `out` de la classe `System`. Les premières applications que l'on crée font généralement intervenir ces deux objets prédéfinis.

Exemple d'instruction pour écrire une chaîne sur la console :

```
System.out.println("Chaîne");
```

L'objet `in` est moins aisé à utiliser et Java 5.0 définit les objets `Scanner` qui permettent des saisies clavier plus simple :

```
Scanner scanner = new Scanner(System.in);
String s = scanner.nextLine();
```

C.2.4 Une application en Java

Pour créer une application, il suffit d'implémenter une méthode particulière, la méthode `main` d'une classe quelconque. Cette méthode a la signature suivante :

```
public static void main(String[] args) {...}
```

C'est une méthode de type `void`, c'est-à-dire ne retournant pas de valeur et prenant en paramètre un tableau de chaînes contenant les paramètres de la ligne de commande. Pour lancer une application il suffit d'exécuter cette méthode.

Exemple 1

Le premier exemple consiste à considérer une chaîne structurée de la façon suivante : une chaîne est constituée d'un ou de plusieurs caractères.

```
// Nom du package de la classe
package intro.java;
// Classes à importer
import java.util.Scanner;
// Nom de la classe
public class Exemple {
// Méthode main exécutable
  public static void main(String[] args) {
// Appel d'une méthode de la classe Exemple
    exp1();
  }
// Déclaration de la méthode exp1
  public static void exp1() {
// Instanciation d'un objet Scanner permettant les entréées console
    Scanner scanner = new Scanner(System.in);
// Lecture d'une chaîne sur la console
    String s = scanner.nextLine();
// Affichage de chaque caractère de la chaîne entrée sur la console
    for (int i = 0; i < s.length() ; i++) {
      System.out.println(s.charAt(i));
    }
  }
}
```

Dans l'exemple précédent `scanner`, `s`, `i` sont des variables locales à la méthode exp1. Ce ne sont ni des variables d'instances ni des variables de classes.

C.3 Éléments de base

C.3.1 Types primitifs

Les types de données primitifs regroupent les valeurs numériques entières ou décimales, les caractères simples et Unicode ainsi que les booléens true et false. Le langage propose un grand nombre de types numériques. On peut classer ces types en deux catégories : les types entiers et les types décimaux. Ce qui différencie les types d'une même catégorie, c'est la taille de mémoire utilisée pour contenir une valeur.

byte	nombres entiers binaires 8 bits.
short	nombres entiers courts sur 16 bits.
int	nombres entiers sur 32 bits.
long	nombres entiers longs sur 64 bits.
float	nombres à virgule flottante sur 32 bits.
double	nombres à virgule flottante double sur 64 bits.
char	caractères Unicode sur 16 bits.
boolean	true ; false ; booléen sur 1 bit.

C.3.2 Types des variables

Si une variable est déclarée de type primitif (caractère, nombre, booléen), elle contient physiquement une valeur. Lors des appels de méthodes les paramètres de type primitif sont passés par valeur.
Si une variable contient une instance d'une classe, elle contient en fait une référence, c'est-à-dire l'adresse mémoire où l'information relative à l'objet stockée. Lors des appels de méthodes les paramètres de type non primitif sont passés par référence. Il peut exister plusieurs références sur le même objet. Une méthode peut construire un objet aussi complexe soit-il. Seule, une référence sera retournée.
La classe `String` n'est pas un type primitif. La comparaison de chaînes doit se faire par la méthode `equals`.

C.3.3 Structures de contrôle des méthodes

Structure conditionnelle

Alternative simple :

```
if (expression booléenne) blocvrai; else blocfaux;
```

Alternative multiple :

```
switch (variable){
 case cas1: //
   instructions cas1
```

```
    break ; //si ce break est omis, on exécute aussi instruc21 et instruc22
  case cas2 :
    instructions cas2
    break;
  ...
  default :
    instructions par defaut
 }
```

Dans une structure alternative multiple, la variable doit être d'un type scalaire, c'est-à-dire complètement ordonné.

Opérateur ternaire :

```
variable = condition ?
      expression si condition Vraie :
      expression si condition Fausse
```

Structure répétitive

Boucle FOR

```
for (initialisation; condition; mise à jour){
   instructions
}
```

- Initialisation : suite d'instructions à exécuter par le programme avant de rentrer pour la première fois dans la boucle.
- Condition : suite de conditions exécutées en fin de boucle. Elles permettent de continuer la répétition.
- Mise à jour : suite d'instructions exécutées chaque fois que la boucle est terminée avant le test de continuité.

Boucle WHILE

```
while (test logique) {
   instructions
}
```

Le test est effectué avant les instructions de la boucle. Tant que le test renvoie la valeur `true` les instructions de la boucle sont exécutées. La boucle n'est pas exécutée si le test renvoie `false` à sa première évaluation.
On utilise plutôt une boucle for lorsque la condition porte également sur un compteur de boucle.

Exemple 2

Nous écrivons simplement une méthode `exp2` qui pourrait être substituée à la méthode `exp1` de l'exemple C.2.4.
Considérons une chaîne qui contient une série de lettres suivie par une série de chiffres telle que : abcdef1234. On peut penser à la méthode suivante pour analyser la chaîne.

```
public static void exp2() {
  Scanner scanner = new Scanner(System.in);
  String s = scanner.nextLine();
  StringBuilder letters = new StringBuilder();
  StringBuilder digits = new StringBuilder();
  int i = 0;
  boolean exist = true;
  char c = s.charAt(i);
  while (exist && Character.isLetter(c)) {
    letters.append(c);
    i++;
    exist = i < s.length();
    if (exist)
      c = s.charAt(i);
  }
  while (exist &&  Character.isDigit(c) ) {
    digits.append(c);
    i++;
    exist = i < s.length();
    if (exist)
      c = s.charAt(i);
  }
  System.out.println(letters.toString());
  System.out.println(digits.toString());
}
```

Exemple 3

L'exemple C.3.3 peut être récrit en :

```
public static void exp3() {
  Scanner scanner = new Scanner(System.in);
  String s = scanner.nextLine();
  StringBuilder letters = new StringBuilder();
  StringBuilder digits = new StringBuilder();
  int i;
  for (i = 0 ; i < s.length() && Character.isLetter(s.charAt(i)); i++) {
    letters.append(s.charAt(i));
  }
  for (; i < s.length() && Character.isDigit(s.charAt(i)); i++ ) {
    digits.append(s.charAt(i));
  }
  System.out.println(letters.toString());
  System.out.println(digits.toString());
}
```

La reconnaissance de la structure entraîne la confrontation des caractères entrés avec une série de contraintes dictées par la structure. Après avoir analysé la chaîne d'entrée, le

programme fournit les groupes de caractères associés avec les éléments de la structure tels que les lettres et les chiffres.

On peut considérer la chaîne d'entrée à deux niveaux : le niveau caractère et le niveau de la structure de la chaîne. On peut voir la structure de caractères contenus dans la chaîne comme une suite de lettres et une suite de chiffres. On peut aussi interpréter la suite de chiffres comme un identificateur et la suite de chiffres comme un entier. La structure de la chaîne devient alors : un identificateur suivi par un entier.
Au deux niveaux, on parle de langages. Les symboles du langage au niveau caractère sont les lettres et les chiffres. Les symboles du langage au niveau de la structure sont <*identificateur*> et <*entier*>.

Exemple 4

Comment construire un analyseur d'affectation ? x=25 ;
On considère qu'il existe une variable d'instance, c par exemple, qui contient toujours le prochain caractère à analyser. On écrit :
- une méthode qui analyse le niveau le plus élevé du langage.
- les méthodes appelées par la première et qui décrivent une structure intermédiaire.
- ainsi de suite les méthodes jusqu'au niveau le plus élémentaire : le niveau caractère

On pourrait obtenir dans cette approche top-down :
- 1. affectation
- 2. identificateur ; entier
- 3. caractère_courant ; consume

C.4 Construction d'une classe

Supposons que nous voulions construire un analyseur permettant de reconnaître une chaîne pouvant contenir plusieurs affectations. Pour simplifier l'exercice nous considérons que la chaîne doit être terminé par le symbole *. La saisie de la chaîne est dissociée de l'analyse elle-même.

C.4.1 Portée - Visibilité

Les variables ne sont accessibles que dans le bloc de code où elles sont déclarées.

Private

Les variables d'instances permettent de définir l'état d'un objet. L'ensemble des valeurs des variables d'instances définit l'état de l'objet à un instant de l'exécution. Les variables d'instances devraient toutes être privées, c'est-à-dire accompagnées du modificateur d'accès `private`. On ne peut accéder à ces variables que dans les méthodes de la classe. Une instance d'une classe peut cependant voir les attributs privés d'une autre instance de la même classe.

```
private String input;
```

Les méthodes d'instance accessoires doivent être également déclarées `private`.

Public

Aucune variable d'instance ne devrait être déclarée publique. Il faut construire les accesseurs à ce type de variables. Les variables déclarées `public` sont accessibles à partir de toute classe.

```
public void parse() { ... }
```

Protected

Une méthode déclarée `protected` dans une classe est accessible par les seules sous-classes.

```
protected void parse() { ... }
```

C.4.2 Constructeurs

Un constructeur est une méthode spéciale d'une classe, permettant d'initialiser ses variables d'instances et d'exécuter différentes opérations d'initialisation.

```
ParserSimpleImpl(String input) {
  this.input = input;
}
```

Le mot clé `this` permet de faire référence à l'objet en cours au moment de l'exécution.

```
this.input = input;
```

La variable d'instance `input` est initialisée avec le paramètre `input`.
la création d'un objet instance d'une classe utilise l'opérateur `new` appelant un constructeur.

```
Scanner scanner = new Scanner(System.in);
String s = scanner.nextLine();
Parsersimple parser = new ParserSimpleImpl(s);
```

C.4.3 Interface

Une interface décrit le comportement d'une classe et ne contient que les signatures des méthodes publiques de la classe. Une

```
public interface ParserSimple {
  public void parse();
}
```

Une interface est destinée à être implémentée par une ou différentes classes qui définissent le corps des méthodes.

```
public class ParserSimpleImpl implements ParserSimple { ... }
```

C.4.4 Application simple

Un parser est destiné à analyser une chaîne. L'unique méthode du parser intéressant un programme utilisateur est `parse` et est déclarée dans l'interface (Section C.4.3) et devra être implémentée. Une application cliente devra également savoir comment construire un objet parser (Section C.4.2).
Une classe implémentant l'interface requiert deux variables d'instance pour contenir la chaîne à analyser et le rang du caractère courant lors de l'analyse.

```
public class ParserSimpleImpl implements Parsersimple {
  private String input;
  private int rang;
  ...
}
```

Elle doit implémenter la méthode `parse`. Le corps de la méthode ne fait apparaître que le niveau le plus élevé d'analyse (C.3.3, niveau 1) : méthode `affectation`.

```
public void parse() {
  rang = 0;
  while (currentChar() != '*')  { //le caractère situé à rang
  affectation();
  consume(); // consume le ;
  }
}
```

Cette méthode `affectation` fait elle-même apparaître le niveau suivant (C.3.3, niveau 2) : méthodes `identificateur` et `entier`.

```
private void affectation() {
  String id = identificateur();
  System.out.println(id);
  consume(); // consume le =
  String n = entier();
  System.out.println(n);
}
```

Les méthodes `identificateur` et `entier` sont similaires. Elles essaient de reconnaître les caractères respectivement comme des lettres ou des chiffres.

```
private String identificateur() {
  StringBuilder s = new StringBuilder();
  while ( Character.isLetter(currentChar()) ) {
    s.append(currentChar());
    consume();
  }
  return s.toString();
}
```

C.5 Conteneur Générique

C.5.1 Déclaration

Considérons maintenant un parser chargé de mémoriser les tokens, c'est-à-dire les groupements de caractères reconnus comme identificateur ou entier. Les tokens sont ici des chaînes. Ils sont ajoutés au fur et à mesure dans une structure conteneur : une liste de chaînes. Un conteneur générique est une structure contenant des objets et connaissant le type des objets quelle contient. Exemple de déclaration :

```
private ArrayList<String> tokens = null;
```

Exemple d'initialisation :

```
tokens = new ArrayList<String>();
```

Le parser devra déclarer une méthode publique permettant à une application client d'utiliser les tokens reconnus :

```
public ArrayList<String> getTokens() {
  return tokens;
}
```

C.5.2 Itération

Java 5.0 offre une façon simple d'itérer sur les objets d'un conteneur générique. Dans l'exemple précédent, les objets contenu dans la structure sont de type `String`. L'itération se fait pour chaque objet de la structure :

```
for (String t : parser.getTokens()) {
  System.out.println(t);
}
```

C.6 Héritage de Classes

L'héritage est un concept fondamental de la programmation orientée objet. Il permet de spécialiser le comportement d'une classe (ses méthodes) en lui créant une ou plusieurs sous-classes.

```
public class ParserIdLCImpl extends ParserAvecGenerique {
 ...
}
```

La sous-classe peut ajouter de nouvelles fonctionnalités à celles existantes dans la super-classes, mais aussi de redéfinir des méthodes héritées. Celles-ci doivent être décarées `public` ou `protected` pour pouvoir être redéfinies. Ainsi, la méthode `affectation` doit être déclarée `protected` dans `ParserAvecGenerique`, la super-classe, et dans `ParserIdLCImpl`, la classe dérivée.

Nous la modifions pour que les identificateurs puissent être composés d'un groupe de lettres suivi éventuellement par un groupe de chiffres. Les autres méthodes utilisées dans la sous-classes doivent être elles aussi déclarées `protected`. Elles ne sont pas "surchargées" dans la sous-classe.

```
protected void affectation() { // overall structure method
  getTokens().add(identificateur() + entier());
  consume(); // consume le =
  getTokens().add(entier());
}
```

C.7 Sites Utiles

Français

http ://www.laltruiste.com/coursjava/sommaire.html
http ://java.developpez.com/cours/

Anglais

http ://java.sun.com/docs/books/tutorial/index.html