Université Amar Telidji Laghoaut
Faculté des sciences

Département d'Informatique

Cours de Compilation

3ème Licence Informatique

Tahar ALLAOUI
t.allaoui@lagh-univ.dz

T. ALLAOUI - UATL

Merci de me signaler les erreurs qui peuvent existter dans ce document

T. ALLAOUI - UATL

Programme du module Compilation

Objectif : L'objectif de ce module est de faciliter la compréhension des étapes de transformation d'un programme source vers un programme objet (cible).

Chapitre1

Introduction et rappels (5%)

Chapitre2

Analyse lexicale (10%)

Chapitre3

Analyse syntaxique (25%)

Analyse descendante

Analyse Ascendante

Chapitre4

Traduction dirigée par la syntaxe (5%)

Chapitre5

Contrôle du type (15%)

Chapitre6

Environnement d'exécution (10%)

Chapitre7

Génération du code et optimisation du code (30%)

T. ALLAOUI - UATL

Table des matières

Chapitre1 : Introduction générale 10
Objectifs de ce chapitre : 11
1. Historique : 12
2. La compilation 12
3. Les parties de compilation 13
3.1 L'analyse (Partie frontale) 13
3.2 La synthèse (Partie finale) 13
4. Structure d'un compilateur 13
4.1 Analyse lexicale 14
4.2 Analyse syntaxique 15
4.3 Analyse sémantique 15
4.4 Génération du code intermédiaire 15
4.5 Optimisation du code intermédiaire 15
4.6 Génération du code 15
La gestion de la table de symboles 15
La gestion des erreurs 15
5. Exemple 16
Chapitre2: Analyse lexicale 18
Objectifs de ce chapitre : 19
1. Introduction 20
Exemple 20
2. Définition 21
2.1 Unité lexicale 21
2.2 Modèle (Pattern) 21
2.3 Lexème 21
2.4 Attribut 21
Exemple 21
3. Rappels 21
3.1 Expressions régulières 22
Exemple 22
3.2 Automates d'états finis (AEF) 22
Remarques 22
Exemple 23

T. ALLAOUI - UATL

4. Mise en œuvre d'un analyseur lexical 23
Exemple 23
1. Les expressions régulières 23
2. Les automates 23
3. L'automate union 24
4. L'automate déterministe et minimal 24
5. Le programme qui simule le fonctionnement de l'automate : 25
5. Erreurs lexicales 25
6. Résoudre quelques problèmes d'analyse lexicale 26
6.1 Les mots clés et les identificateurs 26
6.2 Règles de priorité 26
7. Exercices 28
Chapitre3 : Analyse syntaxique 31
Objectifs de ce chapitre 32
1. Introduction 33
2. Rappels 33
2.1 Grammaires 33
2.2 Dérivation 33
2.3 Mot reconnu par une grammaire 34
2.4 L (G) 34
2.5 Arbre de dérivation 34
Exemple 34
2.6 Grammaires particulier 35
3. Mise en œuvre d'un analyseur syntaxique 36
4. Analyse descendante 36
4.1 Analyse prédictive LL 36
Calculer Premier(X) 37
Calculer Suivant(X) 37
Exemple 37
Construire la table d'analyse 38
4.2 Analyseur syntaxique prédictif 38
4.3 Grammaire LL(1) 40
Définition formelle d'une grammaire LL(1) 40
4.4 Analyse par la descente récursive 40
5. Analyse ascendante 43
T. ALLAOUI - UATL

5.1 Analyse LR(0) 43
Item LR(0) 43
Construction de l'automate 44
Fermeture d'un item (Closure) 44
Fonction Transition (Goto) 44
Les états de l'automate 44
Exemple 44
Construire la table d'analyse 45
Algorithme d'analyse 46
5.2 Analyse SLR(1) 48
Exemple 49
Calculer Suivant 49
La table d'analyse SLR(1) 49
5.3 Analyse LR(1) 51
Fonction Fermeture 52
Fonction Transition (Goto) 52
Construire la table d'analyse LR(1) 53
5.4 Analyse LALR(1) 56
Ensembles d'items LALR(1) 56
La table d'analyse 56
Remarques 57
5.5 Utilisation des grammaires ambiguës 58
Elimination des conflits 60
6. Erreurs syntaxiques 62
6.1 Récupération en mode panique 62
6.2 Récupération au niveau du syntagme 62
6.3 Récupération par production des erreurs 63
6.4 Récupération par correction globale 63
Exercices 64
Chapitre4 : Traduction dirigée par la syntaxe 68
Objectifs de ce chapitre : 69
1. Introduction 70
2. Définitions dirigées par la syntaxe 70
2.1 Définitions 70
Grammaire attribuée 70
T. ALLAOUI - UATL

Arbre syntaxique décoré 70
2.2 Notation 70
2.3 Types des attributs 71
Attributs synthétisés 71
Exemple 71
Attributs hérités 72
Exemple 73
3. Graphe de dépendances 74
4. Les arbres abstraits 75
5. Evaluation des attributs 76
5.1 Evaluation après l'analyse syntaxique 77
5.2 Evaluation pendant l'analyse syntaxique 77
Analyse ascendante et définition S-attribuées 77
Analyse descendante et définition L-attribuées 79
6. Schéma de traduction 79
6.1 Définition 79
6.2 Exemples 79
Exercices 81
Chapitre 5 : Contrôle de type 83
Objectifs de ce chapitre : 84
1. Introduction 85
2. Définitions 85
Expressions de type 85
Système de typage 85
3. Un contrôleur de type d'une grammaire simple 85
4. Conversion de type 87
5. Encore des définitions 87
Opérateurs surchargés 87
Fonctions polymorphes 88
Chapitre7 : Génération du code intermédiaire 89
Objectifs de ce chapitre : 90
1. Introduction 91
2. Différentes formes du code intermédiaire 91
2.1. Les arbres abstraits 91
2.2. Code postfixé (notation polonaise) 91

T. ALLAOUI - UATL

2.3. Code à trois adresses 92
Structures du code à trois adresses 94
1. Les quadruplets 94
2. Les triplets 94
3. Les triplets indirects 95
3. Production du code à 3 adresses 96
3.1. Expressions arithmétiques et Affectation 96
3.2. Expressions booléennes 96
3.3. Expressions conditionnelles 98
3.4. La boucle Tant que 99
Code Court-circuit 100
Exercices 103

T. ALLAOUI - UATL

Tables des figures

Figure 1 : le compilateur 13
Figure 2 : les parties de compilation 13
Figure 3 : Les étapes d’un compilateur 14
Figure 4 : l’arbre syntaxique 16
Figure 5 : l’automate de Identificateur 24
Figure 6 : l’automate de Nombre 24
Figure 7 : automate de Opérateur 24
Figure 8 : Automate de Affectation........ 24
Figure 9 : l’automate union indéterministe 24
Figure 10 : Représentation graphique d’automate 25
Figure 11 : Arbre de dérivation de la chaine : Nbr * id + Nbr 35

T. ALLAOUI - UATL

Chapitre 1: Introduction générale

T. ALLAOUI - UATL

Chapitre1 : Introduction générale

Objectifs de ce chapitre :

- *Répondre à la question : Pourquoi la compilation ?*
- *Présenter les différentes étapes d'un compilateur*

T. ALLAOUI - UATL

1. Historique :

L'utilisateur peut réaliser toutes ses opérations en utilisant l'ordinateur, pour cela, il doit exprimer ses idées sous forme d'un programme.

Les 1er programmes étaient en langage machine, les instructions sont encodées par des chaînes de bits : 0010110… de longueur 8, 16, 32, et ses multiples.

Ces programmes sont incompréhensibles par l'humain, et ils sont difficiles à corriger en cas d'erreur. En plus, les instructions sont différentes selon l'architecture de la machine.

Cette difficulté nous amenait à donner une représentation textuelle symbolique du langage machine : 0010110 → Mov AX, 5 par exemple, c'est l'apparition du langage d'assemblage ou l'assembleur.

Ce langage est un peu plus lisible, mais il a des inconvénients :

- Des programmes très longs pour faire des choses simples.
- L'utilisateur doit connaître les caractéristiques de la machine : nombre de registres, taille de registres et le jeu d'instruction.

On a pensé donc à inventer des langages plus puissants, des langages qui soient facile et efficace, ce qui permet d'inventer les langages de haut niveau.

Ces langages ont plusieurs avantages par rapport au langage d'assemblage :

- Langage compréhensible par l'humain.
- Langage indépendant de la machine : le même programme peut être exécuté sur différentes machines.
- Erreur facile à détecter
- Utilisation des procédures, foncions, structures de données variables,…

On a donc des programmes écrits en langage évolué (Langage de programmation) qui doivent être exécutés par l'ordinateur (µp), le problème c'est que ce dernier ne comprend que le langage machine, il faut donc traduire le programme en langage machine, autrement dit, on doit **compiler** le programme.

T. ALLAOUI - UATL

2. La compilation

La compilation est la traduction d'un programme appelé programme source, écrit en langage de programmation (langage évolué) en un autre programme objet équivalent en langage cible, ce langage cible peut être le langage machine ou l'assembleur.

Ce passage est réalisé à l'aide d'un programme particulier appelé **Compilateur.** L'entrée de ce programme est le programme source (en langage évolué), la sortie est un programme cible (en langage cible).

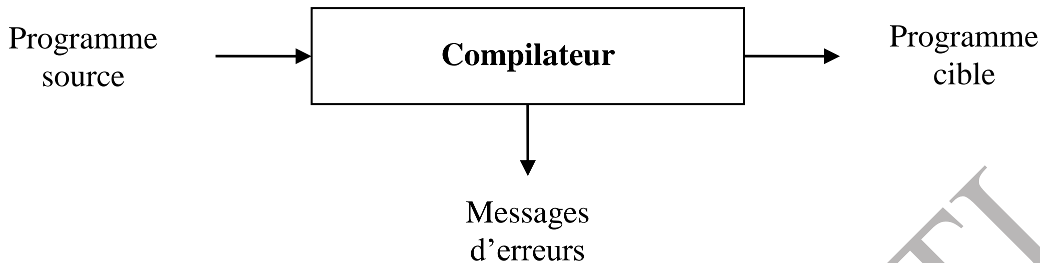


Figure 1 : le compilateur

3. Les parties de compilation

Il y a deux grandes parties de compilation : l'analyse et la synthèse

3.1 L'analyse (Partie frontale)

Cette partie permet de partitionner le programme source en unités , et de créer une forme intermédiaire, cette partie est indépendante du code cible.

3.2 La synthèse (Partie finale)

Permet de créer un programme exécutable à partir de la représentation intermédiaire de programme source, cette partie est indépendante du langage source.

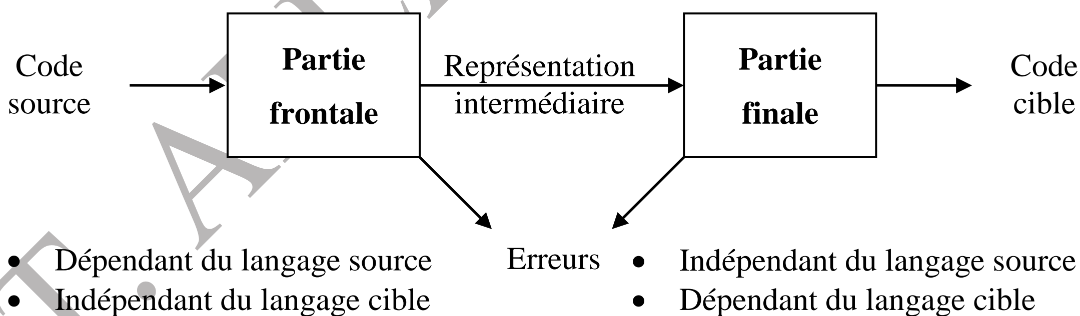


Figure 2 : les parties de compilation

4. Structure générale d'un compilateur

Un compilateur est décomposé en étapes, chaque étape transforme le programme source d'une forme à une autre, la nouvelle représentation sera utilisée par l'étape suivante

T. ALLAOUI - UATL

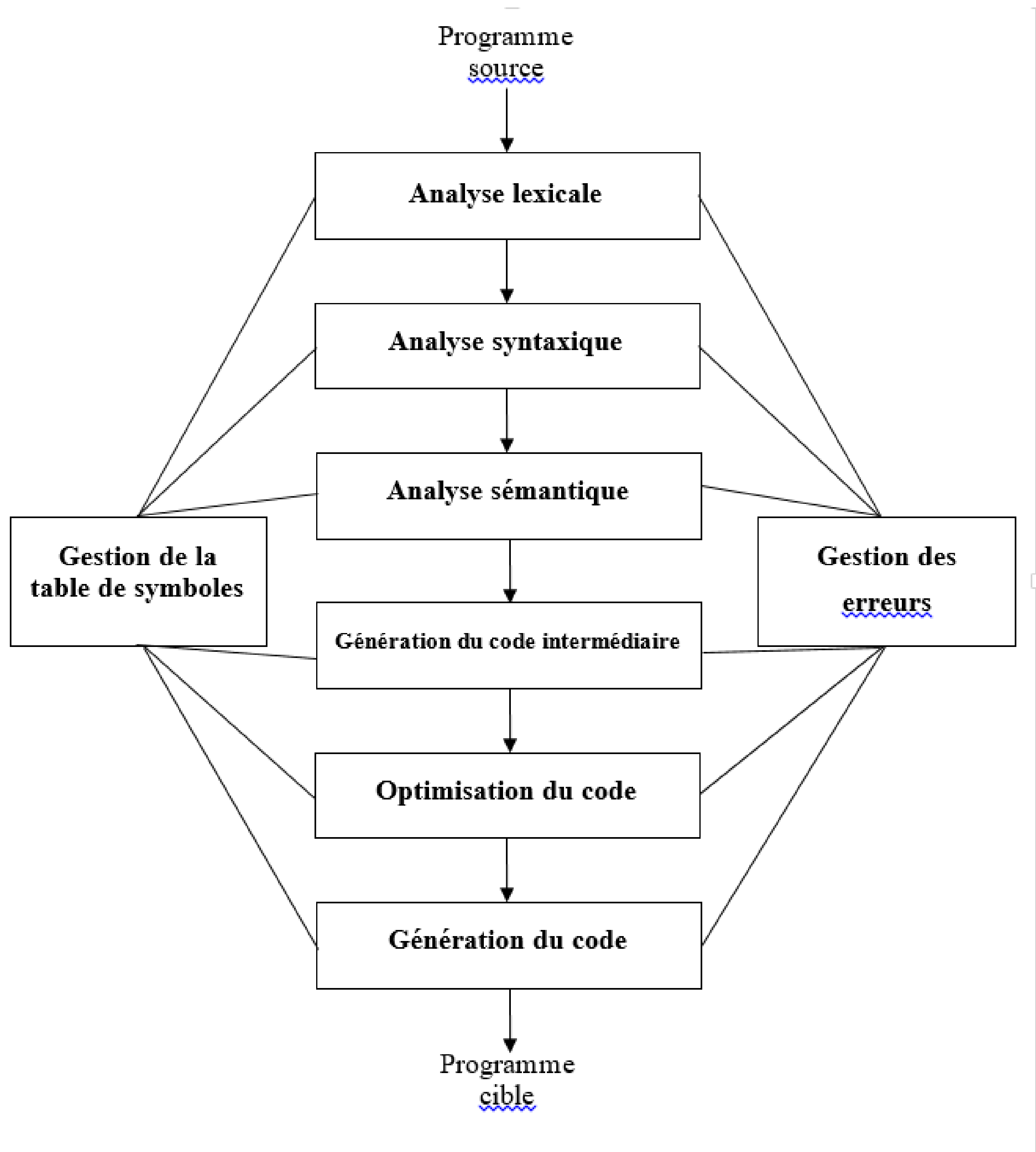


Figure 3 : Les étapes d'un compilateur

4.1 L'analyse lexicale

L'analyseur lexical (*Scanner*) lit le programme source caractère par caractère, pour reconnaître les mots du programme et remplacer chaque mot par une **unité lexicale** (Entité lexicale, Token).

Une unité lexicale prend plusieurs formes, elle peut être un identificateur, un mot clé, un constant, un opérateur ou un séparateur.

L'analyseur lexical doit remplir la table de symboles par les tokens, en associant à chaque token un numéro, et un type. Cette table sera utilisée par les autres étapes de compilateur.

4.2 L'analyse syntaxique

La fonction de l'analyseur syntaxique (*Parser*) est de vérifier que l'ordre des unités lexicales correspond à l'ordre défini par le langage, en d'autre terme, cette étape vérifie la syntaxe du langage à partir de sa grammaire.

L'analyseur syntaxique crée un arbre syntaxique où les feuilles sont les tokens, et les nœuds correspondent à des règles de la grammaire, les feuilles font références aux éléments de la table de symboles.

4.3 Analyse sémantique

Le rôle de l'analyseur sémantique est de vérifier le sens des phrases données par l'analyseur syntaxique.

Dans cette étape, on vérifie que les variables ont un type correct en parcourant l'arbre syntaxique et en vérifiant à chaque niveau que les opérations sont correctes.

4.4 Génération du code intermédiaire

Cette étape consiste à créer un programme équivalent au code source en utilisant un langage plus proche du langage machine.

4.5 Optimisation du code intermédiaire

Cette étape tante à améliorer le code intermédiaire pour obtenir un code machine plus rapide. Quelques optimisations sont triviales telles que l'élimination des variables intermédiaire, d'autres plus complexes telles que la définition des tâches parallèles.

4.6 Génération du code

C'est l'étape finale de compilation, elle produit un code équivalent au programme du départ, soit en assembleur, soit directement en langage machine.

La gestion de la table de symboles

La table de symboles regroupe toutes les unités définis dans le programme, elle conserve les informations importantes de ces unités, par exemple, pour une variable, la table va contenir le type, la valeur, l'emplacement mémoire avec un numéro de référence.

Cette table est utilisée pendant tout le processus de la compilation, depuis son remplissage par l'analyseur lexical jusqu'à la génération du code.

La gestion des erreurs

T. ALLAOUI - UATL

La gestion des erreurs est une partie indéspensable d'un compilateur. Une gestion des erreurs efficace permet de détecter les d'erreurs tout en déterminant leurs causes.

Chaque étape détecte type différents d'erreurs :

- L'analyseur lexical détecte les caractères interdits et les mots non reconnus
- L'analyseur syntaxique peut détecter les erreurs de construction du programme, tel que un « ; » manquant à la fin d'une instruction Pascal par exemple.
- L'analyseur sémantique détecte les erreurs de type, les variables non déclarées,…

5. Exemple

On suppose qu'on a tiré cette instruction à partir d'un programme permettant de calculer la surface d'un triangle :

Surface :=(base/ 2) * hauteur

- L'analyse lexicale produit la suite des unités lexicales suivantes :
  id1 Egal id2 Mul id3 Div Nbr
- L'analyse syntaxique regroupe les tokens sous forme d'un arbre syntaxique en respectant la grammaire

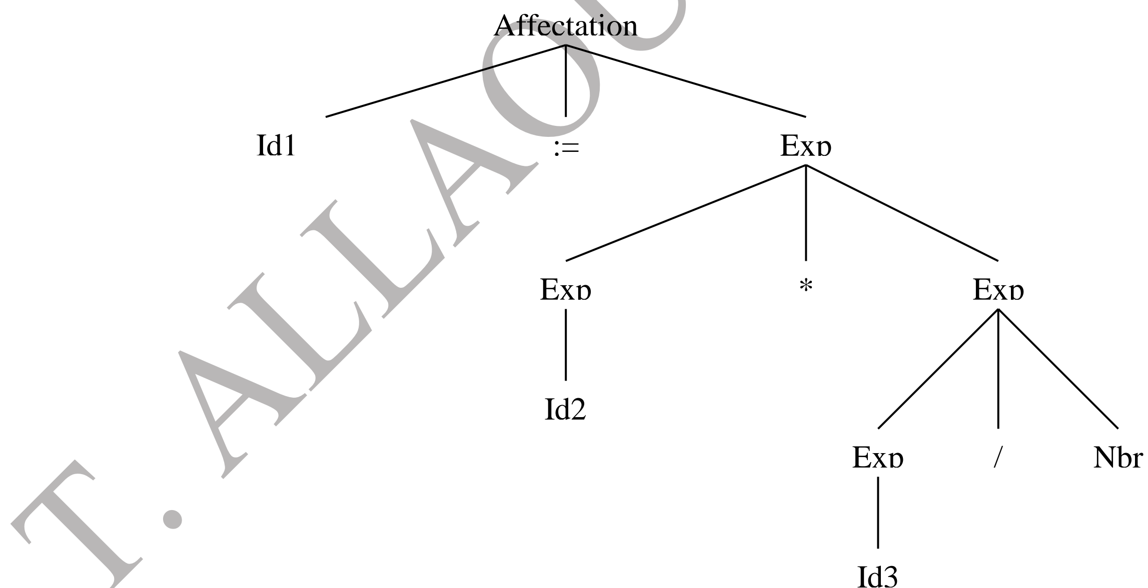


Figure 4 : l'arbre syntaxique

- L'analyse sémantique va contrôler le type : Surface est réel, 2 est un entier, donc on doit convertir 2 en réel.

T. ALLAOUI - UATL

- Le code intermédiaire sera ensuite généré :

```
temp1 :=Entier Vers Réel (2)
temp2 :=ident3 / temp1
temp3 := ident2 * temp2
ident1 := temp3
```

- Ce code peut être optimisé dans l'étape suivante :

```
temp1 := ident3 / 2.0
ident1 := ident2 * temp1
```

- Enfin, le code optimisé sera converti en code cible (assembleur dans ce cas) :

```
Mov  ident3, R1
Div  2.0, R1
Mov  ident2, R2
Mul  R1, R2
Mov  R2, ident1
```

T. ALLAOUI - UATL

Chapitre 2: Analyse lexicale

T. ALLAOUI-UATL

Chapitre2: Analyse lexicale

Objectifs de ce chapitre :

- *Présenter l'étape d'analyse lexicale.*
- *Créer un analyseur lexical.*
- *Décrire la nature des erreurs lexicales.*

T. ALLAOUI - UATL

1. Introduction

L'analyse lexicale est la 1ère étape de compilation, elle opère directement sur le programme source.

La fonction principale de l'analyseur lexical est de scanner (lire) le programme source caractère par caractère, afin de déterminer la suite des caractères comprenant une unité lexicale et d'identifier cette unité. Le résultat est donc une suite d'unités lexicales qui sera traitée par l'analyseur syntaxique.

En plus, l'analyseur lexical doit remplir la table de symboles par des informations concernant les unités lexicales rencontrées dans le programme source.

L'analyseur lexical peut également réaliser des tâches supplémentaires telles que :

- L'élimination des caractères inutiles : les blancs, les tabulations, et les commentaires.
- La gestion des numéros des lignes dans le programme source afin d'associer à chaque erreur la ligne dans laquelle elle figure.

Exemple

Soit le fragment du programme suivant :

```
Begin
   Valeur1 := Valeur0 + 5 ;
End
```

L'analyseur lexical va produire la suites des tokens suivantes :

Begin : mot clé

Valeur1 : identificateur

:= : affectation

Valeur0 : identificateur

+ : addition

5 : constante

; : séparateur

End : mot clé

La table de symboles sera donc

N° de symbole	Unité lexicale	Type
5	Begin	Mot clé
6	End	Mot clé
…	…	…
12	+	Op. addition

T. ALLAOUI - UATL

13	:=	Op. affectation
…	…	…
21	;	Séparateur
…	…	…
35	Valeur0	Identificateur
36	Valeur1	Identificateur
…	….	…
40	5	constante

La suite des unités lexicales qui sera générée est un ensemble de références à la table :
[5, 36, 13, 35, 12, 40, 21, 6]

2. Définition

2.1 Unité lexicale

C’est une suite de caractères ayant une signification commune, en d’autre terme, l’unité lexicale est le plus petit ensemble de caractères qui ait un sens.

2.2 Modèle (Pattern)

C’est une règle associée à une unité lexicale décrivant l’ensemble des mots du programme qui correspondent à cette unité.

2.3 Lexème

Tout mot du programme qui respecte le modèle d’une unité lexicale.

2.4 Attribut

Une information additionnelle sur les tokens qui sera utilisée par les étapes suivantes.

Exemple

Pour le programme précédent :

Unité lexicale	Modèle	Lexème
Identificateur	Toute chaîne de caractère qui commence par une lettre, qui peut être suivie par des lettres ou des chiffres	Valeur0 Valeur1

Pour que l’analyseur lexical puisse réaliser sa tâche, on doit :

1. Décrire le langage du programme source : donner la structure des mots de ce langage.
2. Avoir un moyen permettant de reconnaître les mots du langage.

3. Rappels

T. ALLAOUI - UATL

3.1 Expressions régulières

Les expressions régulières sont des opérateurs permettant de décrire des langages définis sur le même alphabet. Le langage décrit par des expressions régulières est dit langage régulier.

- L'expression régulière ε représente {ε}.
- Si a est un symbole de l'alphabet, alors a est une expression régulière qui représente {a}.
- Si r et s sont des expressions régulières qui représentent L(r) et L(s) alors :
  - r/s est une expression régulière qui représente $L(r) \cup L(s)$.
  - rs est une expression régulière qui représente L(r) L(s).

Pour décrire les unités lexicales d'un langage, on peut utiliser les expressions régulières.

Exemple

Lettre = [A-Z a-z]

Chiffre = [0-9]

Identificateur = *Lettre* (*Lettre*/*Chiffre*)*

Nombre = *Chiffre* (*Chiffre*)* ou $(Chiffre)^+$

Après la description du langage, le problème qui se pose c'est de vérifier qu'un mot appartient à un langage donné ou non.

Pour cela, on utilise les automates d'états finis qui sont des reconnaisseurs des langages réguliers.

3.2 Automates d'états finis (AEF)

Un automate d'états finis (AEF) est défini par :

- Un ensemble S d'états.
- Un état $S_0 \in S$ dit initial
- Un ensemble d'états $\in S$ dits finaux
- Un alphabet des symboles d'entrée
- Une fonction de transition δ

Remarques

- Un mot reconnu par l'automate est une chaîne qui permet de passer de l'état initial à un état final.

T. ALLAOUI - UATL

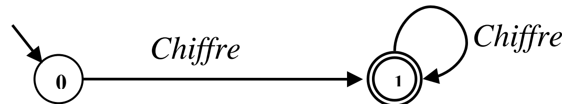
- Le langage reconnu par un automate est l'ensemble de tous les mots reconnus par cet automate.
- Toute expression régulière peut être présentée par un AEF.

Exemple

Pour l'expression régulière Chiffre :

Nombre = Chiffre (*Chiffre*)*

L'automate qui reconnaît cette expression est :



Généralement, le 1er automate obtenu à partir d'une expression régulière est un automate non déterministe, on peut rendre cet automate déterministe, et on peut en plus minimiser cet automate.

4. Mise en œuvre d'un analyseur lexical

Pour implanter un analyseur lexical, on doit suivre les étapes suivantes :

1. Ecrire une expression régulière pour chaque unité lexicale.
2. Créer un AEF pour chaque expression régulière.
3. Créer l'automate union de ces automates.
4. Rendre cet automate déterministe et minimal.
5. Ecrire un programme simulant le fonctionnement de cet automate.

Exemple

Nous imaginons un langage qui permet de réaliser les opérations arithmétiques simples dont les unités lexicales sont : les identificateurs, les nombres entiers, les opérateurs et l'affectation. On suppose que les unités lexicales sont séparées par des blancs.

Pour implanter l'analyseur lexical de ce langage, on va suivre les étapes suivantes :

1. Les expressions régulières

Lettre = [A-Z a-z]

Chiffre = [0-9]

Identificateur = Lettre (*Lettre*/*Chiffre*)*

Nombre = Chiffre (*Chiffre*)* ou $(Chiffre)^+$

Opérateur = + / - / * / ÷

Affectation = :=

2. Les automates

T. ALLAOUI - UATL

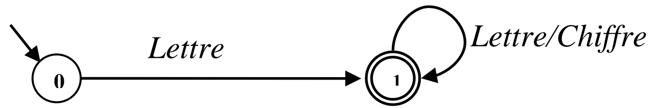


Figure 5 : l'automate de Identificateur

Figure 6 : l'automate de Nombre

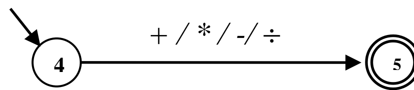


Figure 7 : automate de Opérateur

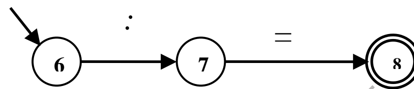


Figure 8 : Automate de Affectation

3. L'automate union

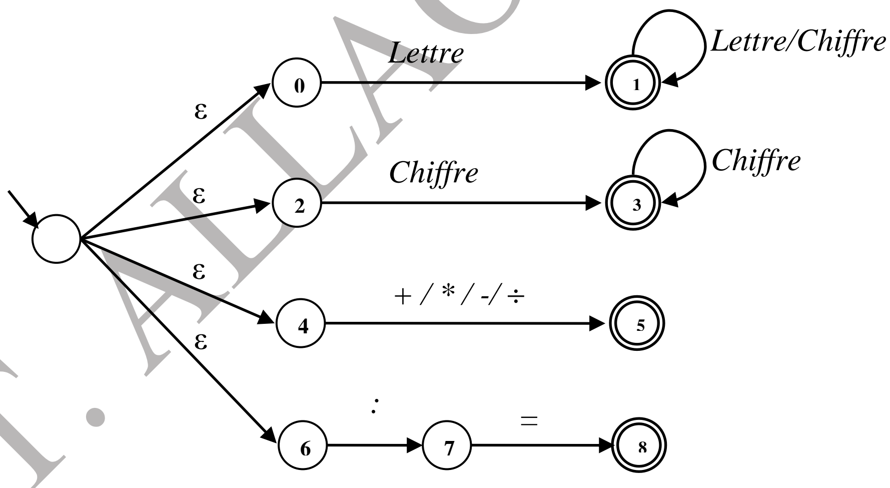


Figure 9 : l'automate union indéterministe

4. L'automate déterministe et minimal

T. ALLAOUI - UATL

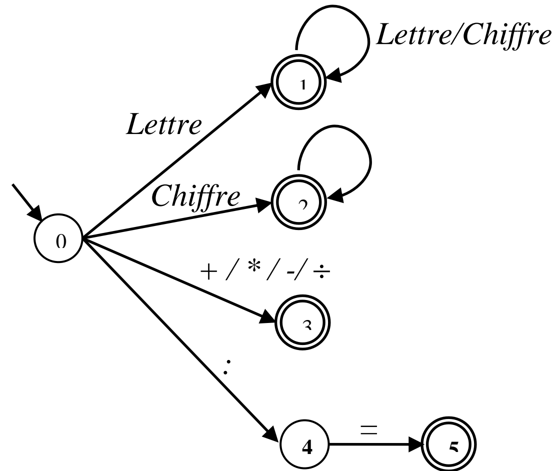


Figure 10 : Représentation graphique d'automate

Etat	*Lettre*	*Chiffre*	*Opérateur*	*:*	*=*
0	1	2	3	4	-
1	1	1	-	-	-
2	-	2	-	-	-
4	-	-	-	-	5

Tableau 1 : Représentation matricielle de l'automate

5. Le programme qui simule le fonctionnement de l'automate :

```
Début
    état :=état initial
    c := le 1er caractère du fichier d'entrée
    Si c ∉ à l'alphabet alors signaler une erreur fsi
    Tant que not eof
    Faire
        Tant que état ≠ final
        Faire
          état := Table (état, c)
          Lire(c)  Si c ∉ à l'alphabet alors signaler une erreur fsi
        Fait
        Si état n'est pas défini alors signaler une erreur
        Sinon
          Cas état parmi
            1 : Rendre (identificateur)
            2 : Rendre (Nombre)
            3 : Rendre (Opérateur)
            5 : Rendre (Affectation)
          Fin cas
        Fsi
    Fait
Fin
```

5. Erreurs lexicales

T. ALLAOUI - UATL

Les erreurs lexicales détectables sont très peu car cet analyseur a une vue très limitée du programme source.

L'analyseur lexical va signaler une erreur lorsqu'il y a un caractère illégal dans le programme source, ou lorsqu'il rencontre une suite de caractères n'ayant pas un sens lexical.

Si l'analyseur lexical rencontre **far** par exemple dans le programme source, s'agit t-il du mot clé **for** erroné ou d'un identificateur, l'analyseur lexical va reconnaître cette chaîne comme un identificateur même si elle est erronée, la détection de cette erreur sera dans l'étape suivante.

6. Résoudre quelques problèmes d'analyse lexicale

6.1 Les mots clés et les identificateurs

Dans la plupart des langages de programmation, les mots clés sont réservés, et on remarque que ces mots clés représentent un sous ensemble des identificateurs, ils seront donc reconnus par l'unité lexicale identificateur. Comment distinguer les identificateurs des mots clés à la reconnaissance d'une unité lexicale identificateur ?

Il y a deux solutions pour ce problème :

1. Les mots clés seront les 1ers dans la table de symboles, lors de reconnaissance d'un identificateur, l'analyseur lexical va comparer cet identificateur avec les mots clés dans la table, il va rendre identificateur si la chaîne reconnue n'est pas un mot clé.
2. Mettre tous les mots clés dans une table spéciale des mots clés, et on utilise une fonction permettent de dire si cet identificateur est un mot clé ou non.

6.2 Règles de priorité

Soit l'exemple suivant :

0 → Zéro

011 → Trois

0^+1^* → Autre

L'automate final de ces expressions régulières est :

Etat	0	1	2	3	4	5
0	**1 (f)**	**2 (f)**	**2 (f)**	**-**	**-**	**-**
1	**-**	**3 (f)**	**4 (f)**	**5 (f)**	**4 (f)**	**4 (f)**

L'analyseur lexical pourra reconnaître soit 0 soit 011 dans le cas où la chaîne 011 lui est présentée. Cette ambiguïté est relevée en faveur de la chaîne la plus longue.

ALLAOUI - UATL

Un autre problème posé est que la chaîne 011 pourrait être reconnue soit par l'expression régulière 011 soit par l'expression régulière 0^+1^* . Ce conflit sera résolu en reconnaissant l'expression régulière qui apparaît en 1er.

Pour la chaîne 010010110 par exemple, l'analyseur lexical doit rendre : autre autre trois zéro.

T. ALLAOUI - UATL

7. Exercices

Exercice 1

Sur l'alphabet X= {*a*, *b*, *c*}, donner les expressions régulières décrivant :

1. Les mots qui commencent par *b*.
2. Les mots qui commencent par a et se terminent par *b*.
3. Les mots contenant exactement trois *a*.
4. Les mots contenant au moins trois *a*.
5. Les mots contenant le facteur *abca* au moins deux fois.

Exercice 2

a. Sur l'alphabet X = {0, 1}, donner les expressions régulières décrivant :

1. Les entiers pairs.
2. Les entiers impairs.

b. Donner l'expression régulière qui représente les nombres octaux et Hexadécimaux

Exercice 3

Donner les automates d'états finis déterministes reconnaissant les expressions régulières des exercices précédents

Exercice 4

Sur l'alphabet X = {*a*, *b*}, donner un automate d'états finis déterministe reconnaissant les langages suivants :

1. Les mots contenant un nombre pair de "*a*"
2. Les mots contenant un nombre pair de "*b*"
3. Les mots contenant un nombre pair de "*a*" et un nombre pair de"*b*"
4. Les mots contenant un nombre pair de "*a*" ou un nombre pair de"*b*"
5. Les mots qui n'ont pas plus de quatre *a* consécutifs

T. ALLAOUI - UATL

Exercice 5

Soient les expressions régulières suivantes :

0 → Zéro

011 → Trois

0^+1^* → Autre

Donner l'algorithme détaillé décrivant l'analyseur lexical de ce langage

Exercice 6

a. Ecrire l'algorithme détaillé décrivant l'analyseur lexical du langage des opérations arithmétiques simple vu en cours.

On suppose qu'on a modifié l'expression régulière de l'unité lexicale Identificateur : un identificateur est une chaîne de caractère commençant par une lettre ou un chiffre, suivi par des lettres ou des chiffres, si cette chaîne commence par un chiffre, elle doit contenir au minimum une lettre.

b. Donner les modifications nécessaires dans l'analyseur lexical

On veut imposer des critères sur la structure des identificateurs : la longueur max ne doit pas dépasser dix caractères, et la longueur min doit être supérieure à trois caractères.

c. Donner les modifications nécessaires dans l'algorithme.

Exercice 7

Sur Internet, les ordinateurs communiquent entre eux grâce au protocole IP (*Internet Protocol*), qui utilise des adresses numériques, appelées adresses IP.

Une adresse IP est une adresse de 32 bits, divisée en quatre parties de longueur 8 bits.

Les adresses IP sont réparties en classes, classe A, classe B et classe C,

Dans la classe A, le bit de poids fort est à zéro, une adresse IP de la classe A est de la forme suivante : 0X….X

Dans la classe B, le 1er bit de poids fort est à un, le 2ème bit est à zéro, une adresse IP de la classe B est de la forme suivante : 10X….X

Dans la classe C, le 1er bit de poids fort est à un, le 2ème bit est à un, et le 3ème bit est à zéro, une adresse IP de la classe C est de la forme suivante : 110X….X

Donner les étapes de création d'un analyseur lexical permettant de :

T. ALLAOUI - UATL

1. Lire une adresse IP obtenue à partir d'un générateur d'adresses, (les adresses sont générées en binaire).
2. Vérifier que cette adresse est correcte.
3. Définir la classe de cette adresse.

T. ALLAOUI - UATL

Chapitre 3: Analyse syntaxique

T. ALLAOUI-UATL

Chapitre3 : Analyse syntaxique

Objectifs de ce chapitre

- *Présenter l'étape d'analyse syntaxique.*
- *Décrire les méthodes d'analyse syntaxique.*

T. ALLAOUI - UATL

1. Introduction

A la fin de la 1ère étape de compilation, l'analyseur lexical génère une suite d'unités lexicales à partir du programme source.

L'analyseur syntaxique (2èmé étape de compilateur), prend comme entrée la suite de tokens et doit reconnaître si cette entrée appartient au langage utilisé. En d'autre terme, l'analyseur syntaxique doit vérifier que la suite de tokens respecte la syntaxe du langage.

Chaque langage de programmation a des règles indiquant la structure syntaxique d'un programme, la suite de tokens correcte est celle qui satisfait les règles grammaticales du langage utilisé.

Comment décrire les suites (les mots) acceptables par un langage ?

On peut penser comme 1ère idée à utiliser les expressions régulières et les AEF pour décrire la structure des langages, mais cette solution n'est pad efficace car les langages réguliers sont limités, et la plupart des langages de programmation dépassent ces limites, ils ne peuvent pas donc s'exprimer sous forme des expressions régulières. Pour cette raison, on va utiliser les grammaires, ces grammaires peuvent décrire comment agencer les unités lexicales.

Le problème est de vérifier qu'un programme (mot) appartient au langage généré par une grammaire, pour cela, on doit construire un arbre de dérivation pour ce mot, si un tel arbre existe, le programme est donc correct.

2. Rappels

2.1 Grammaires

Une grammaire est un quadruplet $G= (V_T, V_N, S_0, P)$ formé de :

- Un ensemble dit V_T non vide de symboles terminaux.
- Un ensemble dit V_N non vide de symboles non terminaux.
- Un symbole particulier $S_0 \in V_N$ appelé Axiome (Symbole de départ).
- Un ensemble P de production, qui sont des règles de la forme $S \rightarrow S_1S_2…S_k$ avec $S \in V_N$ et $S_i \in V_T \cup V_N$.

Remarque

Les symboles terminaux sont les unités lexicales extraites du programme source par l'analyseur lexical.

2.2 Dérivation

T. ALLAOUI - UATL

La dérivation est l'application d'une ou de plusieurs règles de production à partir d'un mot de $(V_T \cup V_N)^+$.

La dérivation obtenue par l'application d'une seule règle est notée $\rightarrow$, $\rightarrow^*$ est une dérivation obtenue par l'application de $n \geq 0$ règles de production.

Exemple

Soit la grammaire

$S \rightarrow aSb \mid \varepsilon$

On peut avoir les dérivations suivantes

$aSb \rightarrow aaSbb$

$S \rightarrow^* ab$

2.3 Mot reconnu par une grammaire

On dit que la chaîne de symboles terminaux w est reconnu par la grammaire ssi $S_0 \rightarrow^* w$

2.4 L (G)

Le langage généré par une grammaire G représente l'ensemble de tous les mots reconnus par cette grammaire

$L(G) = \{w \in V_T^* / S_0 \rightarrow w\}$

2.5 Arbre de dérivation

Soit w une chaîne de symboles terminaux du langage L (G), une dérivation $S_0 \rightarrow w$ existe, cette dérivation peut être schématisé par un arbre appelé Arbre de dérivation.

Exemple

Soit la grammaire

Exp → Exp + T / T

T → T * F / F

F → Nbr / id

Pour la chaîne Nbr * id + Nbr

Cette chaîne est reconnue par le langage, et on peut créer l'arbre de dérivation suivant

T. ALLAOUI - UATL

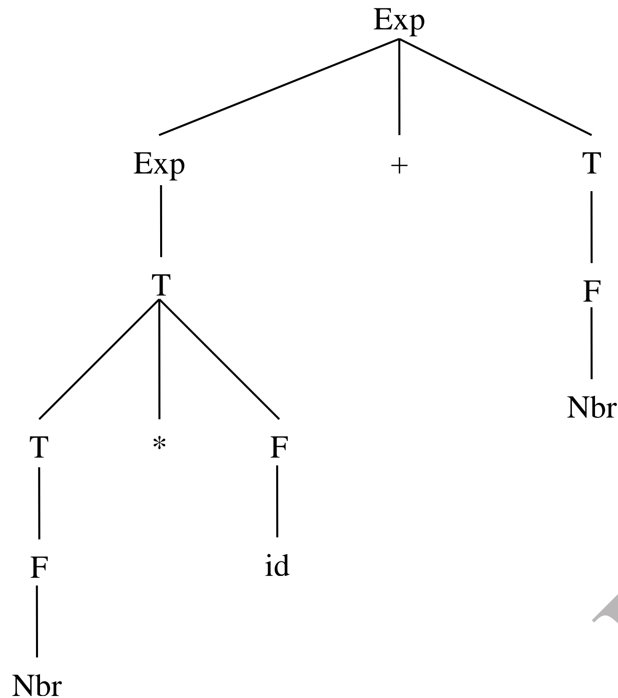


Figure 11 : Arbre de dérivation de la chaine : Nbr * id + Nbr

T. ALLAOUI - UATL

2.6 Grammaires particulier

Grammaire ambiguë

Une grammaire est dite ambiguë, si plusieurs arbres de dérivation existent pour un même mot de L(G), dans ce cas, le langage L (G) est également dit ambigu.

Grammaire récursive à gauche

Une grammaire est dite récursive à gauche s'il existe une production de la forme $A \rightarrow^* A\alpha$.

La récursivité à gauche empêche l'écriture d'un analyseur syntaxique, il est préférable d'éliminer la récursivité à gauche.

Cela revient à remplacer une production $A \rightarrow A\alpha / \beta$ par :

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \varepsilon$

Grammaire non factorisée

Une grammaire ayant des productions de la forme $A \rightarrow \alpha\beta_1 / \alpha\beta_2 / \ldots$ peut poser des problèmes lors de l'analyse, on doit donc factoriser cette grammaire, il faut donc remplacer les productions précédentes par :

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 / \beta_2$

3. Mettre en place d'un analyseur syntaxique

Analyser (parser) un mot (une suite d'unités lexicales), c'est établir si le mot appartient au langage engendré par la grammaire. On essaye de créer l'arbre de dérivation de ce mot à partir su symbole initial, si on arrive à cet arbre, on déduit que le mot est syntaxiquement correct.

Il existe deux méthodes permettant de construire l'arbre de dérivation, la méthode descendante (Top-down) et la méthode ascendante (Bottom-Up).

- **La méthode (analyse) descendante :** Cette méthode consiste à créer l'arbre de dérivation de haut (à partir de l'axiome) vers le bas (les unités lexicales), les règles de production sont utilisées de la gauche vers la droite.
- **La méthode (analyse) ascendante :** La construction de l'arbre se fait de bas en haut, les règles de production sont utilisées de la droit vers la gauche.

4. Analyse descendante

L'analyse syntaxique descendante essaye de construire un arbre de dérivation à partir de l'axiome vers les feuilles.

Deux méthodes déterministes existent pour cette analyse : l'analyse prédictive et l'analyse par la descente récursive.

4.1 Analyse prédictive LL

L'analyse LL (k) (Left to right scan, Left most dérivation) utilise une table appelée Table prédictive ou Table d'analyse.

- Les lignes de cette table sont indicées par les non terminaux.
- Les colonnes sont indicées par les symboles terminaux.
- Les cases de cette table représentent les règles de production à appliquer lors de la lecture du terminal correspondant.

Pour construire la table d'analyse, on doit créer les ensembles Premier et suivant :

- Pour toute chaîne α composée de terminaux et non terminaux, Premier (α) (First (α)) est l'ensemble de tous les terminaux (y compris ε) qui peuvent commencer une chaîne qui se dérive de α.
- Pour chaque non terminal A, Suivant (A) (Follow (A)) est l'ensemble de tous les terminaux a qui peuvent apparaître immédiatement à droite de A dans une dérivation $S \rightarrow \alpha A a \beta$.

T. ALLAOUI - UATL

Calculer Premier(X)

Appliquer les règles suivantes jusqu'à ce qu'aucun terminal ou ε ne puisse être ajouté à l'ensemble Premier

1. Si X est un terminal, Premier(X) = {X}
2. Si X → ε est une production, ajouter ε à premier (X).
3. Si X → aY est une production, et a un terminal alors, ajouter a à Premier(X)
4. Si X est un non terminal et X → Y1Y2…Yk une production, et si pour i = 1,2,…,k-1 Premier(Yi) contient ε, alors ajouter Premier(Yk) à Premier(X)
5. Si X → Y1Y2…Yk une production, et ε est dans Premier(Yi) pour tous les i =1, 2,.. k alors ajouter ε à Premier(X).

Calculer Suivant(X)

Appliquer les règles suivantes jusqu'à ce qu'aucun terminal ne puisse être ajouté à l'ensemble Suivant.

Ajouter un marqueur de fin de chaîne (# ou $) à Suivant(S), où S est l'axiome.

1. Si A → αBβ est une production, alors ajouter le contenu de Premier(β) sauf ε à Suivant(B).
2. S'il y a une production A → αB, alors ajouter Suivant(A) à Suivant(B).
3. S'il y a une production A → αBβ, avec ε ∈ Premier(β), alors ajouter Suivant(A) à Suivant(B).

Exemple

Soit la grammaire

E → TE'

E' → +TE' / ε

T → FT'

T' → * FT' / ε

F → (E) / id

Les non terminaux de cette grammaire sont : E, E', T, T' F.

En appliquant les règles précédentes, on obtient les ensembles Premier et Suivant :

Premier(E) = Premier((T) = Premier(F) = {(, id}

Premier(E') = {+, ε}

Premier(T') = {*, ε}

Suivant(E) = Suivant(E') = {), #}

Suivant(T) = Suivant(E) = {+,), #}

Suivant(F) = {+, *,), #}

T. ALLAOUI - UATL

Construire la table d'analyse

La table d'analyse est une table à deux dimension qui indique pour chaque non terminal A et pour chaque non terminal a ou #, la règle de production à appliquer.

Pour remplir la table d'analyse, on utilise les règles suivantes

1. Pour chaque terminal a dans Premier (α), ajouter A → α à M[A,a].
2. Si ε est dans Premier (α), ajouter A →α à M[A,a] pour chaque terminal a dans Suivant(A)
3. Si ε est dans Premier (α),et # est dans Suivant(A)ajouter A →α à M[A,#]
4. Chaque case vide dans M est une erreur syntaxique.

Exemple

Pour la grammaire précédente, on a déjà calculé les ensembles Premier et Suivant, la table d'analyse sera donc :

Non terminal	Symboles d'entrée					
	id	+	*	(	)	#
E	E → TE'	**erreur**	**erreur**	E → TE'	**erreur**	**erreur**
E'	**erreur**	E' →+TE'	**erreur**	**erreur**	E' → ε	E' → ε
T	T → FT'	**erreur**	**erreur**	T → FT'	**erreur**	**erreur**
T'	**erreur**	T' → ε	T' →*FT'	**erreur**	T' → ε	T' → ε
F	F → id	**erreur**	**erreur**	F → (E)	**erreur**	**erreur**

4.2 Analyseur syntaxique prédictif

Cet analyseur est dirigé par la table d'analyse, il utilise une pile pour déterminer si le flot d'entrée (le mot à analyser) appartient au langage.

Le fonctionnement de cet analyseur est donné par l'algorithme suivant

Positionner le pointeur Ps sur le premier symbole de w#

Soit X le symbole en sommet de la pile

Soit a le symbole repéré par Ps

Répéter

Si X est un terminal ou # alors
    Si X = # alors
        Si a = # alors Accepter
        Sinon Erreur
        Fsi
    Sinon
        Si X = a alors
            Enlever X de la pile
            Avancer Ps
        Sinon Erreur
        Fsi
    Fsi
Sinon (X est un non terminal)
    Si M[X,a] = x → Y1Y2…Yk alors
        Enlever X de la pile
        Mettre Yk, Yk-1, …, Y2, Y1 dans la pile
        Emettre en sortie X → Y1Y2…Yk
    Sinon (la case est vide)
        Erreur
    Fsi
Fsi

Jusqu'à Accepter ou Erreur

Exemple

Analyser la chaine id+id*id pour la grammaire des expressions

Pile	Entrée	Sortie
#E	id+id*id#	E → TE'
#E'T	id+id*id#	T → FT'
#E'T'F	id+id*id#	F → id
#E'T'id	id+id*id#	
#E'T'	+id*id#	T' → ε
#E'	+id*id#	E' →+TE'
#E'T+	+id*id#	
#E'T	id*id#	T → FT'
#E'T'F	id*id#	F → id
#E'T'id	id*id#	
#E'T'	*id#	T' →*FT'
#E'T'F*	*id#	
#E'T'F	id#	F → id
#E'T'id	id#	
#E'T'	#	T' → ε
#E'	#	E' → ε
#	#	Accepter

4.3 Grammaire LL(1)

Une grammaire LL(1) est une grammaire pour laquelle chaque cellule de la table d'analyse contient une seule règle de production au plus.

L : left to right scannig

L : left most dérivation

1: un seul symbole d'entrée

Définition formelle d'une grammaire LL(1)

Une grammaire G est dite LL(1) ssi ses règles de production respectent les conditions suivantes :

Soit $A \rightarrow \alpha / \beta$:

1. $\alpha \rightarrow^* \varepsilon$ ou $\beta \rightarrow^* \varepsilon$ mais pas les deux.
2. Premier $(\alpha) \cap$ Premier $(\beta) = \varnothing$.
3. Si $\beta \rightarrow^*$ alors Premier $(\alpha) \cap$ Suivant $(A) = \varnothing$.

Remarque

Une grammaire ambiguë ou récursive à gauche n'est jamais L.L.(1).

Une condition nécessaire pour qu'une grammaire soit LL(1) :

- G n'est pas ambiguë.
- G n'est pas récursive à gauche.
- G est factorisée à gauche.

4.4 Analyse par la descente récursive

C'est la 2ème méthode déterministe d'analyse descendante, c'est une analyse simple à implémenter qui consiste à écrire une procédure pour chaque non terminal de la grammaire. L'analyse se fait par appels successifs des procédures. Cette analyse a un inconvénient, c'est qu'elle est spécifique à une grammaire donnée, et elle exige un langage de programmation de compilateur acceptant les appels récursifs.

Pour qu'on puisse réaliser cette analyse, on doit ajouter la production $Z \rightarrow S\#$, tel que S est l'axiome, on doit en plus transformer la grammaire en une grammaire LL(1)

Exemple

Etant donnée la grammaire

$S \rightarrow aAb / \varepsilon$

$A \rightarrow cA / ab$

1. Ecrire un analyseur par la descente récursive pour cette grammaire.
2. Analyser la chaîne : acabb#

T. ALLAOUI - UATL

On doit ajouter un non terminal Z, la grammaire sera donc

Z → S#

S → aAb / ε

A → cA / ab

On va maintenant écrire une procédure pour chaque non terminal, on utilise dans les procédures tc pour indiquer le symbole courrant dans la chaîne, et ts pour le prochain symbole dans la chaîne.

Les procédures

```
Procédure Z ( ) ;
Début
      S ( ) ;
      Si tc = # alors Accepter
           Sinon Erreur
      Fsi
Fin
Procédure S ( ) ;
Début
      Si tc = a alors
           t c := ts ;
           A ( ) ;
           Si tc = b alors
                t c := ts
                Sinon Erreur
           Fsi
      Fsi
Fin
Procédure A ( ) ;
Début
      Si tc = c alors
           t c := ts
           A ( ) ;
           Sinon
           Si tc = a alors
                t c := ts
                Si tc = b alors
                     Tc := ts
                     Sinon erreur
                Fsi
                Sinon Erreur
           Fsi
      Fsi
Fin
```

L'analyse de la chaîne acabb#

Pile	Chaîne	Action
#Z	acabb#	Appel de S
#ZS	cabb#	Appel de A
#ZSA	cabb#	Appel de A
#ZSAA	abb#	Avancer
#ZSAA	bb#	Avancer
#ZSAA	b#	Fin de A
#ZSA	b#	Fin de A
#ZS	b#	Avancer
#ZS	#	Fin de S
#Z	#	Fin de S
#Z	#	Accepter

T. ALLAOUI - UATL

5. Analyse ascendante

Au contraire de la méthode descendant, l'analyse syntaxique ascendant crée un arbre de dérivation à partir des feuilles vers l'axiome, en remplaçant la partie droite d'une production trouvée dans la chaîne par la partie gauche.

L'analyseur le plus connu est l'analyseur LR, on peut distinguer 4 méthodes pour cet analyseur : L.R(0), SL.R(1), L.R(1), LAL.R(1). Ces méthodes utilisent le même algorithme qui simule un AEFD, en utilisant une table appelée la table d'analyse.

La différence entre ces méthodes réside dans le contenu de la table d'analyse utilisée.

L'analyseur LR est un analyseur qui lit le mot d'entrée à partir de gauche vers la droite (Left to right scanning) pour construire une dérivation droite (Right most derivaion). Cet analyseur utilise donc une table d'analyse, le flot d'entrée (la chaîne à analyser), et une pile contenant l'état d'analyse.

La table d'analyse est une table à deux dimensions, qui est composée de deux parties : partie *Action* et partie *Transition*.

Les actions qui se trouvent dans la table sont :

- **Décaler (Shift) :** empiler un terminal de la chaîne.
- **Réduire (Reduce) :** remplacer la partie droite d'une production rencontrée sur le sommet de la pile par la partie gauche de la production.
- **Accepter :** le mot est accepté.
- **Erreur :** l'analyseur signale une erreur.

5.1 Analyse LR(0)

Cette méthode est la plus simple parmi les 4 méthodes.

Dans cette analyse, on crée un AEFD un peu particulier, les transitions d'un état à un autre peuvent être par des terminaux ou des non terminaux.

L'automate décrit les différents états de l'analyseur.

Après la construction de l'automate, on doit remplir la table d'analyse, cette table permet de définir l'action à faire pour l'état courant de l'automate et le symbole lu dans la chaîne.

Chaque état de l'automate est composé d'un ensemble d'éléments appelés item LR(0).

Item LR(0)

Un item LR(0) est une règle de production de la grammaire avec un point repérant une position dans la partie droite.

La règle E → TE' par exemple fournit les items LR(0) suivant

T. ALLAOUI - UATL

E → •TE'

E → T•E'

E → TE'•

Pour une production A → ε, un seul item existe A → •

Construction de l'automate

Pour créer l'AEFD, on doit augmenter la grammaire en ajoutant la règle S' → S#.

Pour définir les états et les transitions de cet automate, on a besoin de 2 fonctions : *Fermeture* et *Transition*

Fermeture d'un item (Closure)

Cette fonction permet de définir tous les items qui peuvent être regroupés dans un seul état de l'automate.

Pour un ensemble **I** d'items pour une grammaire G, appliquer les règles suivantes pour créer Fermeture (I) jusqu'à ce qu'aucun nouvel item ne puisse être ajouté :

1. Chaque item de **I** est un item de Fermeture (I).
2. Si A → α•Bβ est un item de Fermeture (I) et B→•γ est un item, alors ajouter B→•γ à Fermeture (I).

Fonction Transition (Goto)

Cette fonction indique la transition de l'état en cours après la lecture d'un symbole de la chaîne.

Pour un ensemble d'item **I**, et un symbole **X** de la grammaire :

- Goto (I, X) = Fermeture {(A → αX•β) / A → α•Xβ est un item de I}

Autrement dit, Goto est l'état obtenu en déplaçant le point après X.

Les états de l'automate

1. L'état initial de cet automate est la fermeture de S' → S #.
2. Pour un état I : Pour chaque item A → α•Bβ dans I, trouver Goto(I,B), si Goto (I,B) est nouveau, il est donc un nouvel état.
3. Répéter jusqu'à ce qu'aucun nouvel état ne puisse être ajouté.

Exemple

Soit la grammaire S → (S) / ( )

La grammaire augmentée est

S' → S#	1
S → (S)	2
S → ( )	3

T. ALLAOUI- UATL

On va maintenant donner les états de l'automate

I_0 = {(S' → •S#)

(S → • (S))

(S → • ( ))}

I_1 = Goto (**I_0**, S) = {(S' → S•#)}

I_2 = Goto (**I_0**, ( ) = {(S → (•S)

(S → (•)

(S → • (S)

(S → • ( ))}

I_3 = Goto (**I_2**, S) = {(S → (S•))}

I_4 = Goto (**I_2**,)) = {(S → ( )• }

I_2 = Goto (**I_2**, ( )

I_5 = Goto (**I_3**,)) ={((S → (S) •)}

Construire la table d'analyse

C'est une table à deux dimensions contenant deux parties : *Action* et *Transition*

Les colonnes de la partie Action sont indicées par les terminaux et #, les colonnes de la partie Transition sont indicées par les non terminaux, les lignes sont indicées par les états de l'automate.

Pour remplir la table, on applique les règles suivantes :

Pour chaque état Ii

1. Si Goto (Ii, a) = Ij, alors mettre Sj dans la case M [Ii, a].
2. Si l'item S'→S•# ∈ Ii, alors mettre Accepter dans la case M [Ii, #].
3. Si Goto (Ii, A) = Ij, alors mettre j dans la case M [Ii, A].
4. Si A→ α• ∈ Ii, alors mettre Rk dans toutes les cases M [Ii, a], tel que k est le numéro de la règle de production.
5. Les cases vides représentent des erreurs.

Application

Pour la grammaire précédente, la table d'analyse est donnée comme suit :

T. ALLAOUI - UATL

Etats	Action			Transition
	(	)	#	S
I_0	S2	Erreur	Erreur	1
I_1	Erreur	Erreur	Accepter	
I_2	S2	S4	Erreur	3
I_3	Erreur	S5	Erreur	
I_4	Reduce3	Reduce3	Reduce3	
I_5	Reduce2	Reduce2	Reduce2	

UATL

Algorithme d'analyse

Initialement, la pile contient l'état initial, et Ps est pointée sur le 1er caractère de la chaîne

Début

Répéter

Soit E l'état en sommet de la pile et a le symbole repéré par Ps

Si M [E, a] = SE' alors

Empiler a , puis E'

Avancer Ps

Sinon

Si M [E, a] = R par A → β alors

Dépiler 2 * longueur de β

Soit E' l'état en sommet de la pile

Empiler A puis M [E', A]

Emettre en sortie A → β

Sinon

Si M [E, a] = Accepter alors Accepter

Sinon erreur

Fsi

Fsi

Fsi

Jusqu'à Accepter ou Erreur

Fin

Pour analyser la chaîne (( )) #

Pile	Chaîne	Action
0	(( )) #	S2
0 (2	( )) #	S2
0 (2 (2	)) #	S4
0 (2 (2) 4	) #	R3 : S→ ( )
0 (2 S 3	) #	S5
0 (2 (S 3) 5	#	R2 :S→ (S)
0 S 1	#	Accepter

Exemple 2

Soit la grammaire

S → (S) / ε

Donner la table d'analyse de cette grammaire

Pour donner la table d'analyse de cette grammaire, on doit augmenter la grammaire, et on doit définir les ensembles des items LR(0)

La grammaire augmentée est :

S' → S#   1

S → (S)   2

S → ε   3

Ensembles des items LR(0) (états de l'automate)

$\mathbf{I_0}$ = {(S' → •S#)

(S → • (S))

(S → •)}

$\mathbf{I_1}$ = Goto ($\mathbf{I_0}$, S) = {(S' → S•#)}

$\mathbf{I_2}$ = Goto ($\mathbf{I_0}$, ( ) = {(S → (•S)

(S → • (S)

(S → •)}

$\mathbf{I_3}$ = Goto ($\mathbf{I_2}$, S) = {(S → (S•))}

$\mathbf{I_2}$ = Goto ($\mathbf{I_2}$, ( )

$\mathbf{I_4}$ = Goto ($\mathbf{I_3}$,)) = {(S → (S)•)}

T. ALLAOUI - UATL

La table d'analyse sera donc de la forme suivante

Etats	Action			Transition
	(	)	#	S
I_0	**S2/R3**	**R3**	**R3**	**1**
I_1	**Erreur**	**Erreur**	**Accepter**	
I_2	**S2/R3**	**R3**	**R3**	**3**
I_3	**Erreur**	**S4**	**Erreur**	
I_4	**R2**	**R2**	**R2**	

On remarque que cette table contient des cases multi définies, on a des cases où l'action est décaler et réduire en même temps, on appelle ça le conflit décaler-réduire (Shift Reduce). Lors d'une analyse, on ne peut pas définir l'action à faire, on ne peut pas donc réaliser une analyse LR(0).

Toute grammaire ayant une table d'analyse LR(0) multi définie n'est pas une grammaire LR(0).

On voit bien que la méthode LR(0) est limitée, on ne peut pas traiter toutes les grammaires en utilisant cette méthode, il faut donc trouver une méthode plus puissante permettant de traiter une grande classe de grammaires.

5.2 Analyse SLR(1)

C'est la 2ème méthode d'analyse ascendante, cette méthode est plus puissante que la méthode précédente, et permet d'éliminer le conflit Shift Reduce rencontré dans l'analyse LR(0).

Le principe de cette méthode est le même avec LR(0), pour remplir la table d'analyse, on utilise les fonctions Fermeture et Transition, la différence est dans la méthode de remplissage de la table.

Pour construire la table d'analyse SLR(1), on applique les règles suivantes :

Pour chaque état Ii

1. Si Goto (Ii, a) = Ij, alors mettre Sj dans la case M [Ii, a].
2. Si l'item S'→S•# ∈ Ii, alors mettre Accepter dans la case M [Ii, #].
3. Si Goto (Ii, A) = Ij, alors mettre j dans la case M [Ii, A].
4. Si A→ α• ∈ Ii, alors pour chaque terminal a appartenant à Suivant (A), mettre Rk dans la case M [Ii, a], tel que k est le numéro de la règle de production.

T. ALLAOUI - UATL

5. Les cases vides représentent des erreurs.

Pour remplir la table d'analyse SLR(1), on doit calculer l'ensemble suivant pour chaque non terminal de la grammaire.

Exemple

Pour la grammaire

S' → S# 1

S → (S) 2

S → ε 3

On a déjà calculé les ensembles des items

Calculer Suivant

Suivant (S') = {#}

Suivant (S) = {), #}

La table d'analyse SLR(1)

Etats	Action			Transition
	(	)	#	S
I_0	S2	R3	R3	1
I_1	Erreur	Erreur	Accepter	
I_2	S2	R3	R3	3
I_3	Erreur	S4	Erreur	
I_4		R2	R2	

On constate que cette table est mono définie, la méthode SLR(1) a éliminée le conflit Shift Reduce, la grammaire est donc SLR(1)

Exemple d'analyse :

La chaîne (( )) #

Pile	Chaîne	Action
0	(( )) #	S2
0 (2	( )) #	S2
0 (2 (2	)) #	R3 : S→ ε
0 (2 (2 S 3	)) #	S4
0 (2 (2 S 3) 4	) #	R2 : S→ (S)
0 (2 S 3	)#	S4
0 (2 S 3) 4	#	R2 : S→ (S)

T. ALLAOUI - UATL

0 S 1	#	Accepter

Exemple 2

Soit la grammaire

S' → S# 1

S → Z 2

S → a 3

Z → aZb 4

Z → ε 5

Ensembles des items (états de l'automate)

$\mathbf{I_0}$ = {(S' → •S#)

(S → • a)

(S → • Z)

(Z → •aZb)

(Z → •)}

$\mathbf{I_1}$ = Goto ($\mathbf{I_0}$, S) = {(S' → S•#)}

$\mathbf{I_2}$ = Goto ($\mathbf{I_0}$, a) = {(S → a•)

(Z → a• Zb)

(Z → •aZb)

(Z → •)}

$\mathbf{I_3}$ = Goto ($\mathbf{I_0}$, Z) = {(S → Z•)}

$\mathbf{I_4}$ = Goto ($\mathbf{I_2}$, Z) = {(Z → a Z•b)}

$\mathbf{I_5}$ = Goto ($\mathbf{I_2}$, a) = {(S → a•Zb)

(Z → •aZb)

(Z → •)}

$\mathbf{I_6}$ = Goto ($\mathbf{I_4}$, b) = {(Z → aZb•)}

$\mathbf{I_4}$ = Goto ($\mathbf{I_5}$, Z)

$\mathbf{I_5}$ = Goto ($\mathbf{I_5}$, a)

Les ensembles Suivant

Suivant (S') = {#}

Suivant (S) = {#}

Suivant (Z) = {), #}

T. ALLAOUI - UATL

La table d'analyse

Etats	Action			Transition	
	a	b	#	S	Z
I_0	S2	R5	R5	1	3
I_1	-	-	Accepter	-	-
I_2	S5	R5	R3/R5	-	4
I_3	-	-	R2	-	-
I_4	-	S6	-	-	-
I_5	S5	R5	R5	-	4
I_6	-	R4	R4	-	-

On remarque bien le conflit Reduce/Reduce dans la table d'analyse, puisque la table est multi définie, la grammaire n'est pas donc une grammaire SLR (1).

Malgré la puissance de l'analyse SLR, elle reste limitée, et on ne peut pas traiter toutes les grammaires avec cette analyse. On doit trouver à nouveau une méthode plus puissante.

On peut améliorer la méthode SLR, en enrichissant les items LR(0) par une information supplémentaire, la forme générale d'un item devient [$A \rightarrow \alpha \bullet B\beta$, a], où $A \rightarrow \alpha B\beta$ est une production, et a un symbole terminal, un tel item est appelé item LR(1), et la méthode d'analyse utilisant cet item est la méthode LR(1).

5.3 Analyse LR(1)

C'est la méthode d'analyse la plus puissante, qui permet de traiter une grande classe de grammaires.

Dans cette méthode, on utilise des items un peu différents, un item LR(1) est composé de deux parties, la 1ère partie est appelée le cœur ou le noyau d'item, c'est le même avec un item LR(0).

La 2ème partie est un caractère terminal appelé caractère de prévision ou look ahead,

Dans un item LR(1), 1 indique la longueur de la 2ème partie.

Un item LR(1) de la forme [$A \rightarrow \alpha \bullet$, a] signifie : réduire par $A \rightarrow \alpha$ uniquement lorsque le prochain symbole en entrée est a dans la chaîne.

Pour réaliser une analyse LR(1), on doit définir les ensembles des items LR(1), et pour définir ces dernier, on va modifier les fonctions *Fermeture* et *Transition*.

T. ALLAOUI - UATL

Fonction Fermeture

Pour un ensemble **I** d'items pour une grammaire G, appliquer les règles suivantes pour créer Fermeture (I) jusqu'à ce qu'aucun nouvel item ne puisse être ajouté :

1. Chaque item de **I** est un item de Fermeture (I).
2. Si [A → α•Bβ, b] est un item de Fermeture (I), alors pour toute règle B→γ et tout terminal a ∈ Premier (βb), ajouter [B→•γ, a] à Fermeture (I).

Fonction Transition (Goto)

Pour un ensemble d'item **I**, et un symbole **X** de la grammaire :

- Goto (I, X) = Fermeture {(A → αX•β, a) / (A → α•Xβ, a) est un item de I}

Exemple

Pour la grammaire précédente

Ensembles des items LR(1)

$\mathbf{I_0}$ = {(S' → •S, #)

(S → • a, #)

(S → • Z, #)

(Z → •aZb, #)

(Z → •, #)}

$\mathbf{I_1}$ = Goto ($\mathbf{I_0}$, S) = {(S' → S•, #)}

$\mathbf{I_2}$ = Goto ($\mathbf{I_0}$, a) = {(S → a•, #)

(Z → a• Zb, #)

(Z → •aZb, b)

(Z → •, b)}

$\mathbf{I_3}$ = Goto ($\mathbf{I_0}$, Z) = {(S → Z•, #)}

$\mathbf{I_4}$ = Goto ($\mathbf{I_2}$, Z) = {(Z → a Z•b, #)}

$\mathbf{I_5}$ = Goto ($\mathbf{I_2}$, a) = {(S → a•Zb, b)

(Z → •aZb, b)

(Z → •, b)}

$\mathbf{I_6}$ = Goto ($\mathbf{I_4}$, b) = {(Z → aZb•, #)}

$\mathbf{I_7}$ = Goto ($\mathbf{I_5}$, Z) ={(Z → a Z•b, b)}

$\mathbf{I_5}$ = Goto ($\mathbf{I_5}$, a)

$\mathbf{I_8}$ = Goto ($\mathbf{I_7}$, b) = {(Z → aZb•, b)}

T. ALLAOUI - UATL

Construire la table d'analyse LR(1)

Pour construire la table d'analyse LR(1), on applique les règles suivantes :

Pour chaque état Ii

1. Si Goto (Ii, a) = Ij, alors mettre Sj dans la case M [Ii, a].
2. Si l'item [S'→S•, #]∈ Ii, alors mettre Accepter dans la case M [Ii, #].
3. Si Goto (Ii, A) = Ij, alors mettre j dans la case M [Ii, A].
4. Si [A→ α•, a] ∈ Ii, alorsmettre Rk dans la case M [Ii, a], tel que k est le numéro de la règle de production.
5. Les cases vides représentent des erreurs.

La table d'analyse de la grammaire précédente :

Etats	Action			Transition	
	a	**b**	**#**	**S**	**Z**
I_0	**S2**	**-**	**R5**	**1**	**3**
I_1	**-**	**-**	**Accepter**	**-**	**-**
I_2	**S5**	**R5**	**R3**	**-**	**4**
I_3	**-**	**-**	**R2**	**-**	**-**
I_4	**-**	**S6**	**-**	**-**	**-**
I_5	**S5**	**R5**	**-**	**-**	**4**
I_6	**-**	**-**	**R4**	**-**	**-**
I_7	**-**	**S8**	**-**	**-**	**-**
I_8	**-**	**R4**	**-**	**-**	**-**

Cette table ne contient pas des conflits, la grammaire est LR(1).

Exemple2

Soit la grammaire

S' → S#   1

S → CC   2

C → cC   3

C → d   4

T. ALLAOUI - UATL

L'analyse LR (1)

$\mathbf{I_0}$ = {(S' → •S, #)

(S → • CC, #)

(C → • cC, c /d)

(Z → •d, c /d)

$\mathbf{I_1}$ = Goto ($\mathbf{I_0}$, S) = {(S' → S•, #)}

$\mathbf{I_2}$ = Goto ($\mathbf{I_0}$, C) = {(S → C•C, #)

(C → •cC, #)

(C → •d, #)}

$\mathbf{I_3}$ = Goto ($\mathbf{I_0}$, c) = {(C → c•C, c /d)

(C → •cC, c/d)

(C → •d, c/d)}

$\mathbf{I_4}$ = Goto ($\mathbf{I_0}$, d) = {(C → d•, c/d)}

$\mathbf{I_5}$ = Goto ($\mathbf{I_2}$, C) = {(S → CC•, #)}

$\mathbf{I_6}$ = Goto ($\mathbf{I_2}$, c) = {((C → c•C, #)

(C → •cC, #)

(C → •d, #)}

$\mathbf{I_7}$ = Goto ($\mathbf{I_2}$, d) = {(C → d•, #)}

$\mathbf{I_8}$ = Goto ($\mathbf{I_3}$, C) = {(C → cC•, c/d)}

$\mathbf{I_3}$ = Goto ($\mathbf{I_3}$, c)

$\mathbf{I_4}$ = Goto ($\mathbf{I_3}$, d)

$\mathbf{I_9}$ = Goto ($\mathbf{I_6}$, C) = {(|C → cC•, #)}

$\mathbf{I_6}$ = Goto ($\mathbf{I_6}$, c)

$\mathbf{I_7}$ = Goto ($\mathbf{I_6}$, c)

La table d'analyse de cette grammaire sera donc

Etats	Action			Transition	
	c	d	#	S	C
I_0	S3	S4	-	1	2
I_1	-	-	Accepter	-	-
I_2	S6	S7	-	-	5
I_3	S3	S4	-	-	8
I_4	R4	R4	-	-	-
I_5	-	-	R2	-	-
I_6	S6	S7	-	-	9
I_7	-	-	R4	-	-
I_8	R3	R3	-	-	-
I_9	-	-	R30	-	-

Analyser la chaîne cccdcd#

Pile	Chaîne	Action
0	cccdcd #	S3
0 c3	ccdcd #	S3
0 c3c3	cdcd #	S3
0 c3c3c3	dcd #	S4
0 c3c3c3d4	cd #	R4 :C→ d
0 c3c3c3C8	cd #	R3 :C→ cC
0 c3c3C8	cd #	R3 :C→ cC
0c3C8	cd #	R3 :C→ cC
0C2	cd #	S6
0C2c6	d #	S7
0C2c6d7	#	R4 :C→ d
0C2c6C9	#	R3 :C→ cC
0C2C5	#	R2 :S→ CC
0S1	#	Accepter

T. ALLAOUI - UATL

La méthode LR(1) est une méthode intéressante, mais son automate a un nombre élevé d'états (des milliers pour quelques langages), sa table d'analyse occupe donc un espace mémoire très important.

Par contre, la table d'analyser SLR occupe moins d'espace, mais cette méthode est limitée est ne permet pas de traiter une grande classe de grammaires.

Il serait très utile de trouver une méthode qui a la puissance de LR(1), et a une table d'analyse réduit.

5.4 Analyse LALR(1)

La méthode LALR(1) (Look Ahead LR(1)), est une méthode intermédiaire entre SLR et LR51), le nombre d'états dans sa table d'analyse est le même avec celle se SLR, et elle permet de traiter une grande classe de grammaires.

Ensembles d'items LALR(1)

Ces ensembles sont créés à partir des items LR(1).

Dans l'analyse LR(1), on peut trouver des états ayant exactement le même cœur, la différence entre ces états n'est que dans les caractères de prévision.

La méthode LALR(1) consiste à fusionner les états dont les items ont les mêmes cœurs, le nouvel item crée a le noyau des items LR(1), et la 2ème partie contient tous les caractères de prévision des items LR(1) fusionnés.

Le nombre des états obtenus est exactement le même avec celui de l'analyse SLR.

La table d'analyse

Pour créer la table d'analyse LALR(1), on regroupe les lignes qui représentent les états fusionnés dans une seule ligne, autrement dit, on superpose les lignes de la table LR(1) pour former les lignes de la table LALR(1).

Exemple

Pour la grammaire précédente, on remarque que les états 3 et 6 ont les mêmes cœurs, la même chose pour les états 4 et 7 et les états 8 et 9.

Les états 3 et 6 seront fusionnés dans un nouvel état dont la forme est :

$\mathbf{I_{36}}$ = { (C → c•C, c / d / #)

(C → •cC, c / d / #)

(C → •d, c / d / #)}

T. ALLAOUI - UATL

Les états 4 et7 seront remplacés par l'état

I_{47} = {(C → d•, c/d/#)}

Et les états 8et 9 seront remplacés par l'état

I_{89} = {(C → cC•, c/d/#)}

Les autres états restent inchangés.

La table d'analyse LALR(1) de cette grammaire est

Etats	**Action**			**Transition**	
	c	**d**	**#**	**S**	**C**
I_0	**S36**	**S47**	**-**	**1**	**2**
I_1	**-**	**-**	**Accepter**	**-**	**-**
I_2	**S36**	**S47**	**-**	**-**	**5**
I_{36}	**S36**	**S47**	**-**	**-**	**89**
I_{47}	**R4**	**R4**	**R4**	**-**	**-**
I_5	**-**	**-**	**R2**	**-**	**-**
I_{89}	**R3**	**R3**	**R3**	**-**	**-**

Exemple

Soit la grammaire

S' → S

S → Aa / bAc / Bc / bBa

A → d

B → d

Cette grammaire est une grammaire LR(1), mais elle n'est pas LALR(1), la table d'analyse LALR(1) contient des conflits Reduce/Reduce.

Remarques

- Malgré les conflits, la méthode LALR(1) est une méthode puissante.
- Pratiquement, la plupart des langages de programmation sont des langages LALR(1).
- Si on veut classer les méthodes, on obtient LR(0) ⊂ SLR(1) ⊂ LALR(1) ⊂ LR(1)

T. ALLAOUI - UATL

5.5 Utilisation des grammaires ambiguës

Une grammaire ambiguë n'est jamais LR(k), ∀ k, mais, les grammaires ambiguës permet de décrire d'une manière courte et plus naturelle certains langages.

Pour résoudre ce problème, on a deux solutions possibles :

La 1ère solution consiste à trouver une grammaire non ambiguë équivalente à la grammaire ambiguë, mais cette solution est coûteuse à cause de deux raisons :

1. Généralement, la grammaire équivalente possède un nombre important de règles de production, la table d'analyse sera donc très volumineuse.
2. La grammaire équivalente peut contenir des règles de production triviale, l'analyseur va consommer un temps important dans des actions de réduction simples.

Exemple

La grammaire des expressions

E → E+E / E*E / (E) / id

La grammaire non ambiguë équivalente est

E → E+T / T

T → T*F / F

F → (E) / id

On remarque bien le nombre de règles de production, et les règles de la forme E → T.

La 2ème solution consiste à utiliser des règles concernant la priorité et l'associativité des opérations pour lever l'ambiguïté sans changer la grammaire.

Exemple

On va analyser la grammaire des expressions en utilisant la méthode SLR(1)

ALLAOUI - UATL

$\mathbf{I_0}$ = {(E' → •E#)

(E → • E+E)

(E → • E*E)

(E → •(E))

(E → • id)}

$\mathbf{I_1}$ = Goto ($\mathbf{I_0}$, E) = {(E' → E•#)

(E → E•+E)

(E → E•*E)}

$\mathbf{I_2}$ = Goto ($\mathbf{I_0}$, ( ) = {(E → (• E))

(E → • E+E)

(E →• E*E)

(E → •(E))

(E → • id)}

$\mathbf{I_3}$ = Goto ($\mathbf{I_0}$, id) = {(E → id•)}

$\mathbf{I_4}$ = Goto ($\mathbf{I_1}$, +) = {(E → E+•E)

(E → • E+E)

(E →• E*E)

(E → •(E))

(E → • id)}

$\mathbf{I_5}$ = Goto ($\mathbf{I_1}$, *) = {(E → E*•E)

(E → • E+E)

(E →• E*E)

(E → •(E))

(E → • id)}

$\mathbf{I_6}$ = Goto ($\mathbf{I_2}$, E) = {(E → (E•))

(E → E•+E)

(E → E•*E)}

$\mathbf{I_2}$ = Goto ($\mathbf{I_2}$, ( )

$\mathbf{I_3}$ = Goto ($\mathbf{I_2}$, id)

$\mathbf{I_7}$ = Goto ($\mathbf{I_4}$, E) = {(E → (E+E•)

(E → E•+E)

(E → E•*E)}

$\mathbf{I_2}$ = Goto ($\mathbf{I_4}$, ( )

$\mathbf{I_3}$ = Goto ($\mathbf{I_4}$, id)

$\mathbf{I_8}$ = Goto ($\mathbf{I_5}$, E) = {(E → (E*E•)

(E → E•+E)

(E → E•*E)}

$\mathbf{I_2}$ = Goto ($\mathbf{I_5}$, ( )

$\mathbf{I_3}$ = Goto ($\mathbf{I_5}$, id)

$\mathbf{I_3}$ =

$\mathbf{I_9}$ = Goto ($\mathbf{I_6}$,)) = {(E → (E) •)

$\mathbf{I_4}$ = Goto ($\mathbf{I_6}$, +)

$\mathbf{I_5}$ = Goto ($\mathbf{I_6}$, *)

$\mathbf{I_4}$ = Goto ($\mathbf{I_7}$, +)

$\mathbf{I_5}$ = Goto ($\mathbf{I_7}$, *)

$\mathbf{I_4}$ = Goto ($\mathbf{I_8}$, +)

$\mathbf{I_5}$ = Goto ($\mathbf{I_8}$, *)

Suivant (E)={+, *,), #}

La table d'analyse SLR(1)

Etats	Action						Transition
	id	+	*	(	)	#	S
I_0	S3			S2		-	1
I_1	-	S4	S5			Accepter	-
I_2	S3	-	-	S2	-	-	6
I_3	-	R4	R4	-	R4	R4	-
I_4	S3	-	-	S2	-	-	7
I_5	S3	-	-	S2	-	-	8
I_6	-	S4	S5	-	S9	-	-
I_7	-	S4/R1	S5/R1	-	R1	R1	-
I_8	-	S4/R2	S5/R2	-	R2	R2	-
I_9	-	R3	R3	-	R3	R3	-

Cette table est multi définie, la grammaire n'est pas donc SLR(1).

Pour éliminer le conflit dans cette table, on va utiliser les règles suivantes :

- + et * sont associatives à gauche.
- * est plus prioritaire que +.

Elimination des conflits

Dans la case [I_7, +], nous avons deux actions, R1 et S4 :

R1 signifie qu'au sommet de la pile on a E+E qui doit être réduit en E, S4 signifie qu'il faut empiler le symbole + qui se trouve à la tête de la chaîne, on a donc une chaîne de la forme a+b+c, cette chaîne doit être écrite sous la forme (a+b)+c, car + est associative à gauche, on favorise donc la réduction, on élimine S4 et on met dans la case R1 seulement.

Dans la case [I_7, *], nous avons deux actions, R1 et S5 :

S5 signifie qu'il faut empiler le symbole * qui se trouve à la tête de la chaîne, dans ce cas, la chaîne d'entrée est de la forme a+b*c, puisque * est plus prioritaire que +, la chaîne doit être écrite sous la forme a+(b*c), on élimine donc R1, et on met dans la case S5 seulement.

Dans la case [I_8, +], nous avons deux actions, R2 et S4 :

La chaîne d'entrée est de la forme a*b+c, puisque * est plus prioritaire que +, la chaîne aura la forme (a*b)+c, on élimine donc S4, et on met dans la case R2 seulement.

Dans la case [I_8, *], nous avons deux actions, R2 et S5 :

On a donc une chaîne de la forme a*b*c, on peut écrire cette chaîne sous la forme (a*b)*c, car * est associative à gauche, on favorise donc la réduction, la case de la table va contenir R2 seulement.

Après ces opérations, on obtient la table d'analyse SLR(1) suivante :

Etats	**Action**						**Transition**
	id	+	*	(	)	#	**S**
I_0	**S3**			**S2**		-	**1**
I_1	-	**S4**	**S5**			**Accepter**	-
I_2	**S3**	-	-	**S2**	-	-	**6**
I_3	-	**R4**	**R4**	-	**R4**	**R4**	-
I_4	**S3**	-	-	**S2**	-	-	**7**
I_5	**S3**	-	-	**S2**	-	-	**8**
I_6	-	**S4**	**S5**	-	**S9**	-	-
I_7	-	**R1**	**S5**	-	**R1**	**R1**	-
I_8	-	**R2**	**R2**	-	**R2**	**R2**	-
I_9	-	**R3**	**R3**	-	**R3**	**R3**	-

On constate que l'utilisation des règles permet de résoudre les problèmes de conflits et permet d'analyser des grammaires ambiguës en utilisant la méthode LR.

T. ALLAOUI-UATL

6. Erreurs syntaxiques

Dans un programme donné, on risque de rencontrer des erreurs syntaxiques, ces erreurs sont provoquées généralement par une unité lexicale mal placée, ou par l'oubli d'une ou de plusieurs unités lexicales. Autrement dit, l'origine des erreurs syntaxiques est un programme qui n'est pas conforme à la grammaire.

Exemple

- Une affectation sans partie droite x :=
- Un begin sans end

Lorsque l'analyseur syntaxique détecte une erreur, il doit :

- Indiquer la présence de l'erreur d'une manière claire et précise (emplacement, cause).
- Traiter rapidement chaque erreur.
- Le traitement des erreurs ne doit pas ralentir la compilation d'un programme correcte.

Heureusement, la plupart des erreurs ne mettent en jeu qu'un seul token (, à la place de ; ou l'oubli d'un ; par exemple), dans ce cas, l'analyseur syntaxique peut facilement localiser et même donner la cause de l'erreur.

D'autres erreurs se produisent longtemps avant leur détection (end sans begin par exemple), la tâche de l'analyseur dans ce cas devient très difficile.

6.1 Récupération en mode panique

C'est une méthode simple qui est applicable dans la plupart des méthodes d'analyse.

L'analyseur syntaxique met à jour un ensemble de points de synchronisation pendant le déroulement de l'analyse. Lorsque l'analyseur détecte une erreur, il va ignorer toutes les unités jusqu'à en rencontrer un point de synchronisation.

L'avantage de cette méthode est sa simplicité, mais l'élimination d'un e grande partie du programme sans vérifier sa validité représente l'inconvénient majeur de cette méthode.

6.2 Récupération au niveau du syntagme

Dans cette méthode, l'analyseur peut effectuer des corrections locales lors de détection d'une erreur ; par exemple, remplacer une, par un ; ou insérer un ; manquant.

Le choix de correction n'est pas toujours évident, il faut en plus choisir des remplacements qui ne conduisent pas à des boucles infinies (l'insertion d'un symbole avant le caractère courant). La correction des erreurs qui se produisent longtemps avant la détection est un autre problème de cette méthode.

T. ALLAOUI - UATL

6.3 Récupération par production des erreurs

Lorsqu'on a une idée sur les erreurs courantes, on ajoute à la grammaire des règles de production qui engendrent des constructions erronées.

Exemple

Pour détecter une) mal placée, on ajoute les règles :

Error → op)

Error → Exp)

Lors de l'utilisation d'une règle générant une erreur, on envoie un message d'erreur détaillé. L'avantage de cette méthode est la rapidité de détection et les messages d'erreurs permettant de corriger directement l'erreur, mais le nombre important d'erreurs traitées nécessite un nombre important de règles dans la grammaire.

6.4 Récupération par correction globale

Cette méthode permet de minimiser les changements effectués

Entrée : une chaîne C syntaxiquement incorrecte

Sortie : une chaîne C' remplaçant C telle que :

- Le nombre de correction (insertion, remplacement et suppression) soit minimal.
- C' est syntaxiquement correcte.

Malheureusement, cette méthode reste théorique car elle est trop coûteuse en mémoire et en temps.

T. ALLAOUI - UATL

Exercices

Exercice 1

Soit les grammaires suivantes

G1 : S → Abc / Baa
A → Aab / Ab / a
B → Bb / b

G2 : S →AB
A → BS / b
B → SA / a

G3: S → A / a
A → AB / BC
B → Ca / a
C → Aa / S

Eliminer si nécessaire la récursivité à gauche dans ces grammaires.

Exercice 2

Donner les ensembles Premier et Suivant des grammaires suivantes

S → bS / Bb
B → aB / ε

S → Bac / a
A → bSBa / ε
B → AbcS / ε

Exercice 3

Soit les grammaires suivantes

G1 : S → A / B / a
A → BBA
B → dB / ε

G2 : S →ASBa / ab
A → aA / c
B → bB

G3: S → aAB/ aaB
A → aA / b
B → ba

Ces grammaires sont-elles LL(1) ?

Exercice 4

Soit la grammaire suivante :

S → AB / aSBb / CSB
A → bA / ε
B → dB / ε
C → cC / ε

6. Cette grammaire est-elle LL (1).

T. ALLAOUI- UATL

7. Donner sa table d'analyse.
8. Donner un analyseur par la descente récursive pour cette grammaire
9. Analyser la chaîne aadb par les deux méthodes.

Exercice 5

Soit la grammaire suivante :

S → UaTb / b

U → c / ε

T → YU / aTe

Y → d / ε

1. Cette grammaire est-elle LL (1).
2. Donner sa table d'analyse.
3. Analyser la chaîne adcaacebb

Exercice 6

Soit la grammaire suivante :

S → (T) / a / b

T → T , S / S

1. Cette grammaire n'est pas LL (1), pourquoi ?
2. Donner une grammaire LL (1) équivalente à cette grammaire, et donner sa table d'analyse
3. Analyser la chaîne (a,(b,a),a)
4. Donner un analyseur par la descente récursive pour cette grammaire.
5. Analyser la chaîne ((a),b

Exercice 7

Soit la grammaire **G** suivante

G : S → aAc

A → Abb / b

1. Calculer les ensembles d'items LR (0) de la grammaire **G**, et donner sa table d'analyse
2. Cette grammaire est-elle LR (0) ?

T. ALLAOUI - UATL

Exercice 8

Soit la grammaire **G** suivante

G : S → AaB / B

A → bB / c

B → A

Cette grammaire est-elle LR (0)? Est-elle SLR (1) ?

Exercice 9

1. Montrer que la grammaire **G** suivante n'est ni LR (0), ni SLR (1) :

G **:** S → Ta / Ub

T → Tcc / c

U → Ucc / cc

2.a. Donner la table d'analyse LR (1) de cette grammaire.

2.b. Analyser les chaînes : ccccca, cccb

Exercice 10

Soit la grammaire suivante :

S → V = E / E

E → V

V → *E

V → id

10. Cette grammaire est-elle LR (0), SLR (1), LR (1), LALR (1) ?

Exercice 11

Soit la grammaire suivante :

E → S∨S / S∧S / ¬S / (S) / v / f.

4. Donner la table d'analyse SLR (1) de cette grammaire, Conclure.
5. Utiliser les règles suivantes pour éliminer les conflits dans la table d'analyse

T. ALLAOUI - UATL

$\neg$ est plus prioritaire que $\wedge$, $\wedge$est plus prioritaire que $\vee$, $\neg$ est un opérateur unaire, , $\wedge$ et $\vee$ sont associatifs à gauche, et les expressions parenthésées sont évaluées en premier.

3. Analyser la chaine $\neg$ (t $\wedge$ f $\vee$ t)

T. ALLAOUI - UATL

Chapitre 4: Traduction dirigée par la synatxe

T. ALLAOUI - UATL

Chapitre4 : Traduction dirigée par la syntaxe

Objectifs de ce chapitre :

- *Pourquoi l'analyse sémantique*
- *Décrire les méthodes d'analyse sémantique*
- *La traduction dirigée par la syntaxe*

T. ALLAOUI - UATL

1. Introduction

Après les deux premières étapes de compilation, le programme à compiler doit être lexicalement et syntaxiquement correct, mais il peut contenir des erreurs de sens.

Les analyseurs lexicaux et syntaxiques ne sont pas aptes à assurer la correction de certaines propriétés du langage, telles que l'usage des variables (on ne peut pas utiliser une variable qui n'a pas été déclarée), ou le contrôle des opérations (on ne peut pas multiplier un entier avec une chaîne).

On doit donc assurer qu'un programme bien formé lexicalement et syntaxiquement a un sens → Analyse sémantique.

L'analyse sémantique effectue des vérifications de sémantique (signification) sur le code source en cours de compilation, en utilisant un ensemble de règles de construction des programmes appelées les règles sémantiques.

Ces règles sont définies par le langage de programmation, et elles sont associées à chacune des règles de production d'une grammaire.

Il existe deux notations pour associer des règles sémantiques aux productions : les définition dirigée par la syntaxe et les schémas de traduction.

2. Définitions dirigées par la syntaxe

Une définition dirigée par la syntaxe (DDS) est une généralisation d'une grammaire non contextuelle, à chaque symbole de cette grammaire (terminal ou non terminal), on associe un ensemble d'attributs.

Un attribut peut être un nombre, un type, une chaîne de caractères, une adresse mémoire,…

Chaque règle de production de la grammaire possède un ensemble de règles sémantiques permettant de calculer la valeur des attributs associés aux symboles qui apparaissent dans cette production.

2.1 Définitions

Grammaire attribuée

Une grammaire décorée par des actions sémantiques calculant les valeurs des attributs est appelée grammaire attribuée

Arbre syntaxique décoré

On peut ajouter à chaque nœud d'un arbre de dérivation les valeurs des différents attributs calculés par le nom terminal de ce nœud, on parle alors d'arbre syntaxique décoré ou annoté.

2.2 Notation

T. ALLAOUI - UATL

- On notera X.a l'attribut a du symbole X.
- Les règles sémantiques d'une règle de production A $\rightarrow \alpha$ ont la forme b= f(c_1, c_2, …, c_n) où f est une fonction.

Exemple

Pour la grammaire suivante :

S $\rightarrow$ aSb / aS /cSacS / ε

On peut définir l'attribut nba qui calcule le nombre de a, et nbc qui calcule le nombre de c

Pour la règle de production S $\rightarrow$ cSacS par exemple, on peut définir deux règles sémantiques :

S_0.nba:= S_1.nba+1

S_0.nbc:= S_1.nbc+2

2.3 Types des attributs

On peut distinguer deux types d'attributs, selon la façon don't ils sont calculés: synthétisés et hérités

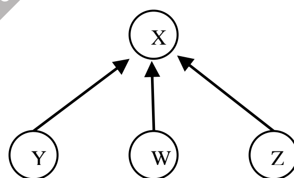
Attributs synthétisés

Un attribut synthétisé est calculé pour le nom terminal en partie gauche de la règle de production en fonction de la valeur d'attributs de symboles en partie droite.

X $\rightarrow$ YWZ

X.a = f(Y.b, W.e, Z.c)

Autrement dit, l'attribut synthétisé pour le nom terminal d'un nœud est calculé à partir des valeurs des fils de ce nœud.



.

Remarque

Si un non terminal X a un attribut synthétisé a, et une règle de production de la forme X $\rightarrow \alpha$ tel que $\alpha \in T$, alors la valeur de a est fournie par l'analyseur lexical

Exemple

Soit la DDS suivante qui associe un attribut synthétisé val aux non terminaux d'une grammaire :

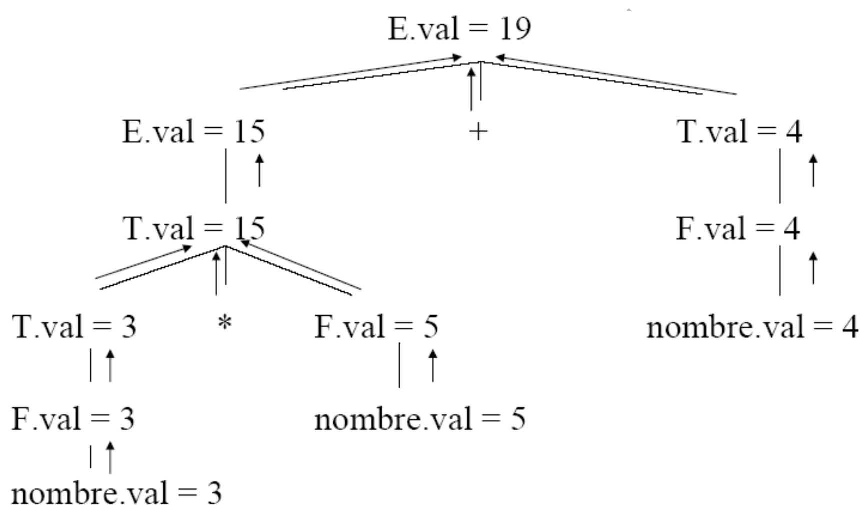
T. ALLAOUI - UATL

Règles de production	Règles sémantiques
S → E	Ecrire E.val
E → E1+T	E.val :=E1.val+T.val
E → T	E.val := T.val
T → T1*F	T.val := T1.val * F.val
T → F	T.val := F.val
F →(E)	F.val :=E.val
F → nbr	F.val := nbr.vallex

Le terminal nbr a un attribut synthétisé vallex (valeur lexical) dont la valeur est fournie par l'analyseur lexical.

Exemple d'analyse

Pour la chaîne 3*5+4, on obtient



Remarque

Une DDS qui utilise uniquement des attributs synthétisés est appelée une définition S-attribuée.

Attributs hérités

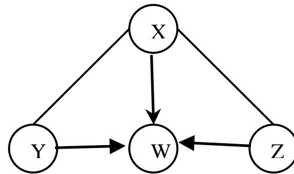
Un attribut hérité est calculé pour un non terminal se trouvan en partie droite de la règle en fonction de la valeur d'attributs des autres non terminaux en partie droite et/ou du non terminal en partie gauche.

X →YWZ

W.b = f(X.a, Y.c, Z.e)

T. A. UATL

Autrement dit, l'attrinut hérité est un attribut dont la valeur à un nœud d'un arbre syntaxique est calculée à partir des attributs du père et/ou de frères de ce nœud.



Exemple

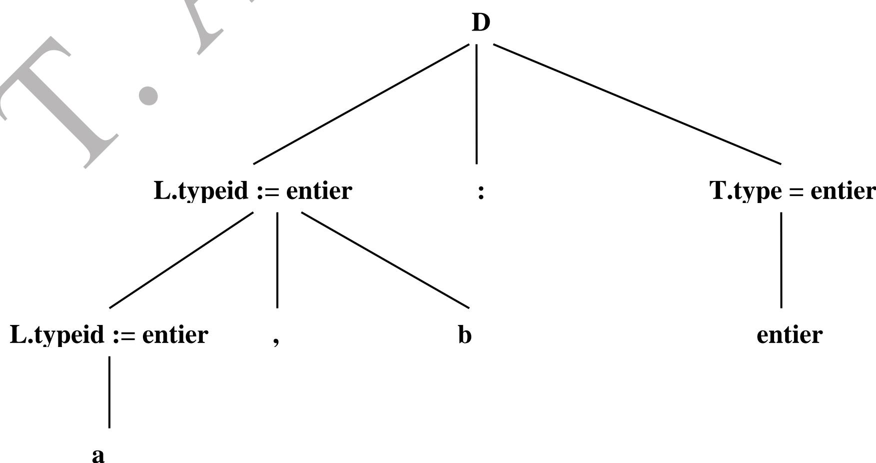
La DDS suivante utilise un attribut hérité typeid

Règles de production	Règles sémantiques
D → L : T	L.typeid := T.type
T → entier	T.type :=entier
T → réel	T.type := réel
L → id	AjouterType (id.entrée, L.typeid)
L → L1, id	L1.typeid := L.typeid AjouterType (id.entrée, L.typeid)

La procédure AjouterType permet d'ajouter le type de chaque identificateur à son entrée dans la table de symboles indiquée par l'attribut entrée.

Exemple

Pour la chaine a, b : entier



Remarque

T. ALLAOUI - UATL

Une DDS qui utilise uniquement des attributs hérités est appelée une définition L-attribuée.

3. Graphe de dépendances

Dans une DDS, l'ordre d'exécution des règles sémantiques n'est pas défini, mais on doit avoir un moyen permettant de définir un ordre d'exécution ou d'évaluation.

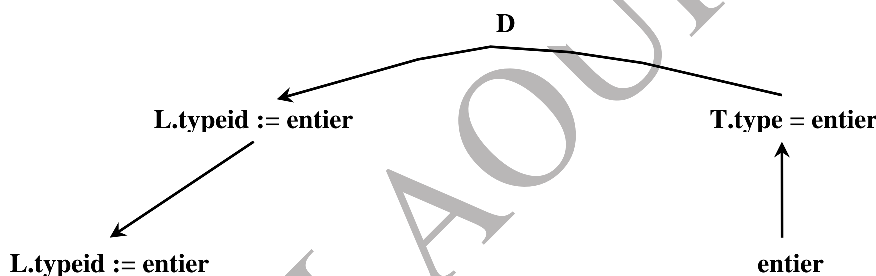
Soit b = f(c_1, c_2,…, c_n), puisque b dépend de c_1, c_2, …, c_n, alors la règle sémantique définissant b doit être évaluée après les règles sémantiques définissant c_1, c_2, …, c_n,

On peut donc construire un graphe orienté appelé le graphe de dépendance, ce graphe décrit les interdépendances entre les attributs aux nœuds d'un arbre syntaxique décoré.

Comment construire ce graphe ?

- Chaque attribut a un sommet dans le graphe.
- Si un attribut b dépend d'un attribut c, alors il y a un arc du sommet correspondant à c au sommet correspondant à b.

Pour l'arbre de l'exemple précédent, on peut tracer le graphe de dépendances suivants :



Pour définir un ordre d'évaluation des attributs d'un ASD, on doit utiliser un tri topologique du graphe de dépendances associé à cet arbre.

Un tri topologique d'un graphe de dépendances est un ordonnancement a_1, a_2, …, a_n des sommets du graphe tel que, si $a_i \rightarrow a_j$, alors a_i doit apparaître avant a_j dans l'ordonnancement.

Le tri topologique d'un graphe de dépendances n'est pas unique, tout tri permet de définir un ordre valide d'évaluation des règles sémantiques associées aux nœuds d'un arbre syntaxique.

Pour l'exemple précédent, on peut définir le tri suivant :

S1 := réel

S2 := S1

AjouterType (id.entrée, S2)

Remarque (les arbres syntaxiques décorés)

T. ALLAOUI - UATL

Les arbres syntaxiques décorés sont créés lors de l'analyse syntaxique.

L'ordre de création de cet arbre peut ne pas correspondre à l'ordre d'évaluation des attributs, on doit donc faire plusieurs parcours dans cet arbre pour pouvoir évaluer les différents attributs.

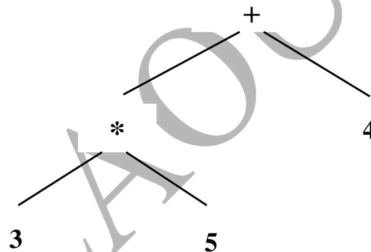
De plus, cet arbre occupe un grand espace mémoire car il contient une grande quantité d'informations, certaines informations sont inutiles pour l'analyseur sémantique.

On doit utiliser une autre représentation permettant d'éviter le parcours d'une part, et ayant une forme condensée qui ne représente que les informations nécessaires d'autre part → Arbres Abstraits.

4. Les arbres abstraits

L'utilisation des arbres abstraits permet de séparer l'analyse syntaxique et l'analyse sémantique, cet arbre n'est créé qu'après la création de l'arbre syntaxique, on peut donc éviter les éventuels parcours.

La forme d'un arbre abstrait est une forme réduite d'un arbre syntaxique, qui ne contient que les données importantes. Pour la chaîne 3 * 5 + 4 par exemple, l'arbre abstrait est :



T. ALLAOUI - UATL

Construire un arbre abstrait

L'arbre abstrait est créé de la même façon que l'arbre syntaxique décoré, on utilise une DDS permettant de créer l'arbre abstrait, et on attache des attributs aux nœuds de l'arbre.

Exemple

Pour les expressions arithmétiques, on utilise les fonctions suivantes pour créer les nœuds d'arbre abstrait :

- CréerNoeud (op, gauche, droit) : créer un nœud « opérateur » d'étiquette op avec deux champs contenant des pointeurs vers gauche et droit.
- CréerFeuille (id, entrée) : créer un nœud « identificateur » d'étiquette id avec un champ contenant entrée, un pointeur vers l'entrée de l'identificateur dans la table de symboles.

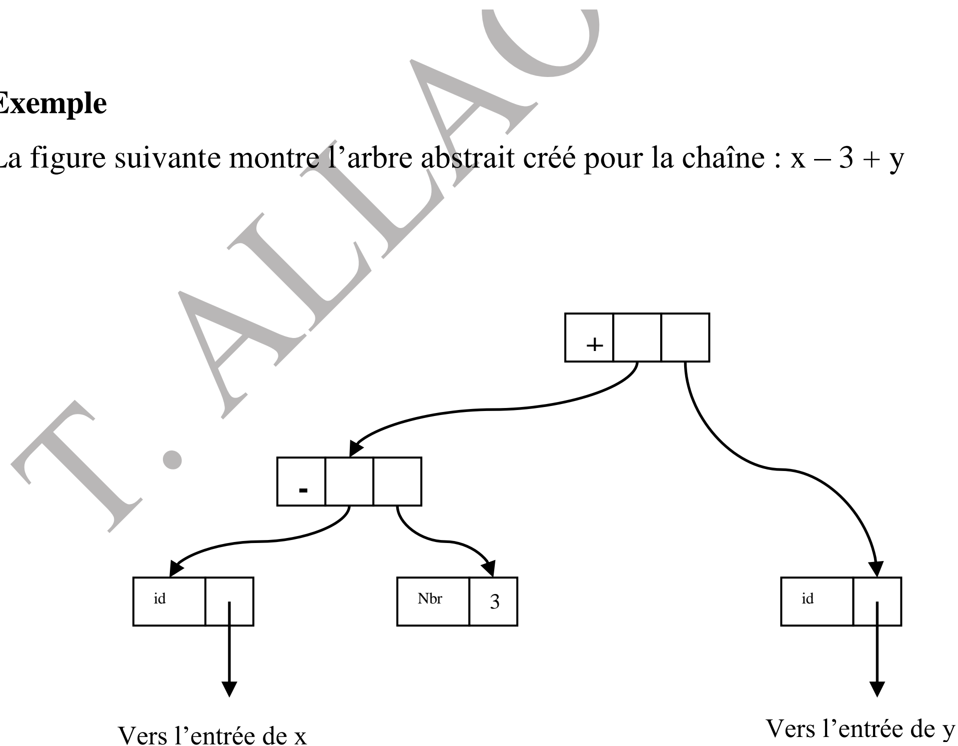
- CréerFeuille (Nbr, val) : créer un nœud « nombre » d'étiquette Nbr, avec un champ contenant val, la valeur du nombre.

La DDS S-attribuée suivante utilise un attribut p-nœud permettant d'exprimer la construction d'un arbre abstrait :

Règles de production	Règles sémantiques
E → E1+T	E. p-nœud := CréerNoeud ('+', E1. p-nœud, T. p-nœud)
E → E1-T	E. p-nœud := CréerNoeud ('-', E1. p-nœud, T. p-nœud)
E → T	E. p-nœud := T. p-nœud
T →(E)	T. p-nœud := E. p-nœud
T → id	T. p-nœud := CréerFeuille (id, id.entrée)
F → nbr	T. p-nœud := CréerFeuille (Nbr, Nbr.val)

Exemple

La figure suivante montre l'arbre abstrait créé pour la chaîne : x – 3 + y



5. Evaluation des attributs

T. ALLAOUI

L'évaluation des attributs dépend de la relation entre l'analyse syntaxique et sémantique, l'analyse syntaxique et l'analyse sémantique peuvent être liés, les attributs seront donc évalués lors de l'analyse syntaxique.
On peut également séparer les deux analyses, dans ce cas les attributs sont évalués après l'analyse syntaxique.

5.1 Evaluation après l'analyse syntaxique

Le calcul des attributs s'effectue sur l'arbre syntaxique généré à la fin de l'analyse syntaxique par des parcours de cet arbre selon l'ordre d'évaluation des attributs.
Cette méthode est avantageuse puisqu'on n'est pas dépendant de l'ordre de création des sommets pour évaluer les attributs, mais le stockage de l'arbre de l'arbre dans la mémoire rend cette méthode très coûteuse.

5.2 Evaluation pendant l'analyse syntaxique

Les attributs sont évalués lors de l'analyse syntaxique en utilisant une pile permettant de conserver les valeurs des attributs, cette pile peut être la même pile de l'analyseur syntaxique, et on peut utiliser une autre pile.
Le problème de cette méthode est la dépendance entre l'ordre de création des nœuds de l'arbre et l'évaluation des attributs, si on a une grammaire S-attribuées, on doit utiliser une analyse ascendante, si la grammaire est L-attribuées, on doit utiliser une analyse descendante, la grammaire contenant les attributs synthétisés et hérités est difficile à traiter.

Analyse ascendante et définition S-attribuées

- Les attributs synthétisés sont évalués par un analyseur syntaxique ascendant au fur et à mesure avec la lecture du texte d'entrée.
- Les valeurs des attributs sont stockées dans la pile de l'analyseur syntaxique ou dans une autre pile.
- Si on une règle de la forme A.a = f (X.b, Y.c, Z.e), on peut calculer la valeur de l'attribut de A avant de réduire WYZ en A, en utilisant les valeurs des attributs de X, Y, Z qui sont déjà disponibles dans la pile.

Exemple

Pour la grammaire des expressions arithmétiques, on donne la DDS et la traduction des actions avec une pile :

T. ALLAOUI - UATL

Règles de production	Règles sémantiques	Traduction avec une pile
S → E	Ecrire E.val	
E → E1+T	E.val :=E1.val+T.val	tmpT := Depiler () tmpE := Depiler () Empiler (tmpT+tmpE)
E → E1-T	E.val :=E1.val-T.val	tmpT := Depiler () tmpE := Depiler () Empiler (tmpT-tmpE)
E → T	E.val := T.val	
T → T1*F	T.val := T1.val * F.val	tmpF := Depiler () tmpT := Depiler () Empiler (tmpT*tmpF)
T → F	T.val := F.val	
F →(E)	F.val :=E.val	
F → nbr	F.val := nbr.vallex	Empiler (nbr)

T. ALLAOUI - U

Analyse descendante et définition L-attribuées

- Les attributs hérités peuvent être évalués par un analyseur descendant.
- L'information véhiculée par les attributs se propage de gauche à droite.

Exemple

Pour calculer le niveau d'imbrication des) dans un système de parenthèses bien formé, on définit une DDS L-attribuée utilisant un attribut hérité niv qui indique le niveau d'imbrication :

Règles de production	Règles sémantiques	Traduction avec une pile
S' → S	S.niv := 0	Empiler (0)
S → (S)S	S_1.niv := S_0.niv+1 S_2.niv := S_0.niv	tmp := Depiler () Empiler (tmp) Empiler (tmp+1)
S → ε	Ecrire S.niv	Ecrire Depiler()

Analyser la chaîne : (()(()))()# (voir TD)

6. Schéma de traduction

Dans une DDS, on peut associer des règles sémantiques aux règles de la grammaire, mais l'ordre d'exécution de ces actions n'est pas défini → problème : Quand appliquer Quelle action ?

Pour résoudre ce problème, on utilise les schémas de traduction.

6.1 Définition

Un schéma de traduction est une grammaire non contextuelle dans laquelle :

- Des attributs (synthétisés ou hérités) sont associés aux symboles de la grammaire.
- Des actions sémantiques, délimitées par des accolades { }, sont insérées dans les parties droites des productions.

Autrement dit, un schéma de traduction est une DDS dans laquelle l'ordre d'exécution des actions sémantiques est imposé (les règles sémantiques sont placées à l'endroit où on souhaite les évaluer)

6.2 Exemples

Calculer le nombre de a dans une chaîne

S' → { nba := 0} S

S → a { nba := nba +1}S

Une operation arithmétique

T → T*F { T.val := T1.val * F.val}

T. ALLAOUI - UATL

Lorsqu'on utilise les schémas de traduction, on doit assurer que la valeur d'un attribut est disponible avant qu'une action s'y réfère.

Pour les attributs synthétisés, mettre l'action à la fin de la règle de prodction.

Attributs hérités et hérités :

- Un attribut hérité d'un symbole en partie droite d'une production doit être calculé dans une action située avant ce symbole.
- Une action ne doit pas faire référence à un attribut synthétisé d'un symbole situé à droite de l'action.
- Un attribut synthétisé d'un non terminal en partie gauche ne peut être calculé qu'après que tous les attributs dont il dépend ont été calculés. L'action calculant ce symbole peut être placée à la fin de la partie droite de la production.

T. ALLAOUI - UATL

Exercices

Exercice 1

Soit la grammaire **G** suivante

G: S → XED

X → + / - / ε

E → BE / B

B → 0 / … / 9

D → **.**E

1. Donner une DDS permettant de calculer la valeur dénotée par un mot de L(G)

Exercice 2

1. Donner une DDS donnant le nombre de a, b et c dans un mot de (a/b/c)*
2. Donner l'arbre syntaxique décoré pour le mot bbacbaaabcc#.

Exercice 3

1. Ecrire une DDS permettant de traduire un entier sous forme binaire en sa valeur décimale.
2. Donner un exemple d'un arbre syntaxique décoré.

Exercice 4

On considère un robot qui peut être commandé pour se déplacer d'un pas vers l'est, l'ouest, le nord ou le sud à partir de sa position courante.

Une séquence de tels déplacements peut être engendrée par la grammaire

S → début L fin

L → PL / P

P → est /oust / nord / sud

La position du robot est donnée dans le plan (le nord étant en haut). (0,0) est la position initiale du robot.

1. Ecrire une DDS affichant les positions successives (x,y) du robot.

T. ALLAOUI - UATL

2. Donner un arbre décoré pour le mot `début est est nord ouest sud sud nord ouest fin`

Exercice 5

Considérons la grammaire suivante qui génère des expressions arithmétiques formées de constantes entières et réelles et de l'opérateur +

E → E + T /T

T → N.N / N

N → Chiffre N/ chiffre

Chiffre → 0/   /9

Lorsque l'on additionne 2 entiers, le résultat est un entier, sinon c'est un réel

1. Ecrire une DDS donnant le type de l'expression

2. Donner un arbre décoré pour le mot 5+3.05+10

T. ALLAOUI - UATL

Chapitre 5: Contrôle de type

T. ALLAOUI- UATL

Chapitre 5 : Contrôle de type

Objectifs de ce chapitre :

- *Comprendre le rôle du contrôle de type*
- *Comment implémenter un contrôleur de type ?*

T. ALLAOUI - UATL

1. Introduction

Le contrôle de type est une tâche importante parmi les tâches de l'analyseur sémantique, ce contrôle permet de garantir certaines contraintes du langage source :

- Un identificateur doit être déclaré une seule fois dans un programme.
- Pour une affectation de la forme id := Exp, le résultat de Exp obtenu après l'évaluation de ses attributs et id doivent avoir le même type.
- Un opérateur n'est appliqué qu'à des opérandes compatibles, par exemple, l'addition d'une matrice est une chaîne de caractère n'est pas accepté.

Si une contrainte n'est pas acceptée, le contrôleur de type doit signaler une erreur de type.

Le contrôleur de type permet également de réaliser des changements ou des modifications d'une manière automatique, telle que la conversion de type lors d'une opération arithmétique.

2. Définitions

Expressions de type

Une expression de type peut être :

- Un type de base d'un langage : entier, réel, booléen.
- Ou un constructeur de type appliqué à d'autres types de base, Tableau (I, T) par exemple est une expression de type qui dénote le type d'un tableau dont les éléments sont de type T.

Système de typage

Un système de typage est un ensemble de règles permettant d'associer des expressions de type à une partie d'un programme (une ou plusieurs règles de productions).

La solution la plus naturelle pour associer ces expressions aux règles de grammaire est donc d'utiliser les DDS ou les ST afin de définir les systèmes de typage.

3. Un contrôleur de type d'une grammaire simple

Pour implémenter un contrôleur de type, on doit associer des règles sémantiques aux règles de grammaires, le rôle de ces règles (actions) est de vérifier que le type est correct pour chaque règle de production.

Soit la grammaire suivante :

S → D ; E

D → id : T

T → caractère / entier / réel / booléen / tableau [nbr] de T

E → nbr / id / E mod E

T. ALLAOUI - UATL

La DDS de contrôle de type associée à cette grammaire sera donc:

Règles de production	Règles sémantiques
S → D : E	
D → id : T	AjouterType (id.entrée, L.typeid)
T → caractère	T.type :=caractère
T → entier	T.type :=entier
T → réel	T.type := réel
T → booléen	T.type := booléen
T → tableau [nbr] de T_1	T.type :=tableau (1..nbr.val, T_1.type)
E → nbr	E.type := entier
E → id	E.type := Trouver(id.entrée)
E → E_1 mod E_2	E.type := Si (E_1.type=entier et E_2.type=entier) alors entier sinon Erreur de type Fsi
E → E_1 [E_2]	E.type := Si (E_2.type=entier et E_1.type=tableau (i, j) alors j sinon Erreur de type Fsi

Remarque

Le contrôle de type ne s'applique pas seulement sur les symboles de grammaire, on peut même contrôler le type d'une instruction ou d'une fonction.

Généralement, un type « vide » est associé à une instruction si elle est correcte, sinon, une erreur de type doit être signalée.

Exemple

Pour une instruction d'affectation, son type est correct si la partie droite et la partie gauche de l'affectation ont le même type, dans ce cas, le type d'affectation est vide, dans le cas contraire, une erreur sera signalée.

La même chose pour une instruction conditionnelle, si le type de condition n'est pas booléen, on doit signaler une erreur.

T. ALLA

Règles de production	Règles sémantiques
S → id := E	S.type := Si (id.type = E.type) alors vide Sinon erreur de type Fsi
S → Si E alors S	S.type := Si (E.type = booléen) alors S_1.type Sinon erreur de type Fsi

4. Conversion de type

La conversion de type est généralement nécessaire dans les opérations arithmétiques, par exemple, l'opération d'addition doit être effectuée à des opérandes de même type, mais on peut additionner un entier à un réel, une conversion de type d'entier vers le type réel est faite pour pouvoir effectuer l'addition.

Exemple

Soit DDS suivante

Règles de production	Règles sémantiques
E → nbr	E.type := entier
E → nbr.nbr	E.type := réel
E → id	E.type := Trouver(id.entrée)
E → E_1 op E_2	E.type := Si E_1.type=entier et E_2.type=entier alors entier Sinon Si [(E_1.type=entier et E_2.type=entier) ou (E_1.type=entier et E_2.type=réel) ou (E_1.type=réel et E_2.type=entier)] alors réel sinon Erreur de type Fsi Fsi

5. Encore des définitions

Opérateurs surchargés

Un opérateur surchargé est un opérateur qui a des significations différentes suivant le contexte.

L'opérateur * par exemple est un opérateur surchargé, car la signification de * dans A*B dépend de A et B : entiers, réels, matrices,…

Fonctions polymorphes

Une fonction (ou un opérateur) est dite polymorphe lorsque l'on peut l'appliquer à des arguments de types variables (et non fixés à l'avance). On peut dans certains langages, définir par exemple une fonction calculant la longueur d'une liste d'éléments, quelque soit le type de ces éléments, cette fonction est donc polymorphes.

T. ALLAOUI - UATL

Chapitre 7: Génération du code intermédiaire

T. ALLAOUI-UATL

Chapitre7 : Génération du code intermédiaire

Objectifs de ce chapitre :

- *Pourquoi utiliser une représentation intermédiaire?*
- *Présenter les différentes formes du code intermédiaire.*
- *Comment créer un code intermédiaire ?*

T. ALLAOUI - UATL

1. Introduction

Dans un modèle de compilation par Analyse-Synthèse, la 1ère partie est indépendante du code cible, alors que la 2ème partie est très liées à la machine cible et ses caractéristiques.

Après les trois premières étapes d'analyse, le programme source reçoit un code simple qui est intermédiaire entre le code source et le code objet, ce code est appelé le code intermédiaire.

A partir de ce code intermédiaire, la partie finale de compilation produit le code objet.

Ce code intermédiaire qui est indépendant de la machine cible permet de :

- Créer des compilateurs pour plusieurs machines en modifiant seulement la partie finale.
- Assurer un certain degré de portabilité.
- Optimiser le code intermédiaire avant de générer le code cible.

2. Différentes formes du code intermédiaire

Plusieurs formes sont utilisées pour représenter un code intermédiaire, les trois formes principales sont les arbres abstraits, le code postfixé, et le code à trois adresses.

2.1. Les arbres abstraits

Les arbres abstraits peuvent être considérés comme des formes intermédiaires.

Cet arbre ne représente que des informations nécessaires, on peut dire qu'un arbre abstrait est une représentation condensée du programme source.

Pour créer des arbres abstraits, on peut associer des règles sémantiques aux règles de production, on obtient donc des DDS ou des ST permettant de créer l'arbre abstrait pour tout programme source (Voir chapitre 4, section 4).

2.2. Code postfixé (notation polonaise)

C'est une autre forme du code intermédiaire, dans un tel code, l'opérateur vient toujours après ses opérandes.

Le code postfixé d'une expression E est défini récursivement par :

1. Si E est une variable ou constante, alors le code postfixé de E = E.
2. Si E est une expression de la forme E_1 op E_2, où op est un opérateur binaire, alors le code postfixé de E est E'_1 E'_2 op où E'_1 et E'_2 sont les codes de E_1 et E_2 respectivement.

Exemple

Pour l'expression arithmétique 5+9*2

cp (5+9*2) = cp (5) cp (9*2) +

T. ALLAOUI - UATL

= 5 cp (9) cp (2) * +

= 5 9 2 * +

Comment créer un code postfixé ?

Pour créer un code postfixé d'un fragment du langage, on utilise les règles sémantiques qui seront associées aux règles de productions.

Exemple : les expressions arithmétiques et l'affectation :

Soit la grammaire suivante

I → id := E

E → E +E

E → E *E

E → id

Pour créer le code postfixé de cette grammaire, on utilise le schéma de traduction suivant :

I → id := E {I.code := id.nom // E.code // ':='}

E → E +E {E.code := E_1.code // E_2.code // '+'}

E → E *E {E.code := E_1.code // E_2.code // '*'}

E → id {E.code := id.nom}

// est l'opérateur de concaténation.

2.3. Code à trois adresses

Le code à trois adresses est une représentation intermédiaire très proche du code assembleur, et dans laquelle le programme est une séquence d'instructions à un seul opérateur chacune, ces instructions peuvent être numérotées.

La terminologie code à trois adresses peut se justifier par le fait que chaque instruction contient en général trois adresses, deux pour les opérandes et une pour le résultat.

La forme générale d'une instruction est :

<num> x := y op z

- <num> est le n° d'instruction.
- y et z sont les opérateurs.
- x est le résultat

On peut citer les instructions à trois adresses suivantes :

T. ALLAOUI - UATL

Instruction	Forme	Signification
Affectation binaire	<num> x := y op z	op est binaire
Affectation unaire	<num> x := op y	op est unaire
Affectation indicée	<num> x := y [i] ou <num> x[i] := y	Affecter le contenu de la case dans la variable ou l'inverse
Copie	<num> x := y	Copier y dans x
Branchement inconditionnel	<num>Aller à L	Aller à l'instruction étiquetée par L
Branchement inconditionnel	<num> Si Exp aller à L	Si exp est vérifiée, aller à l'instruction étiquetée par L
Appel de procédure	param 1 param 2 …. param n Appeler p,n	Appeler la procédure avec les n paramètres qu'ils la précèdent

Exemple

Pour l'instruction a := b*c+b*(b-c)*(-c), on peut générer la suite des instructions suivante :

<1> t1 := b*c

<2> t2 := b-c

<3> t3 := b*t2

<4> t4 := - c

<5> t5 := t3*t4

<6> t6 := t1+t5

<7> a := t6

Remarques

- Le nombre de variables temporaires qu'on peut utiliser est illimité.
- Les variables temporaires créées seront chargées dans la table de symboles.

Structures du code à trois adresses

Il existe plusieurs structures pour stocker le code à 3 adresses en mémoire, les quadruplets, les triplets et les triplets indirects sont les plus connus.

1. Les quadruplets

Un quadruplet est composé de 4 champs, un champ pour l'opérateur, un champ pour le 1er argument, un champ pour le 2ème argument, et un champ pour le résultat.

Exemple

a := b* -c + b* -c

Le code à 3 adresses correspondant sous forme de quadruplets est :

	Op	Arg1	Arg2	Résultat
<0>	Moins	c		t1
<1>	*	b	t1	t2
<2>	Moins	c		t3
<3>	*	B	T3	t4
<4>	+	t2	T4	t5
<5>	:=	t5		a

Remarque

L'inconvénient de cette forme est le nombre élevé de variables temporaires utilisées, chaque variable temporaire créée sera chargée dans la table de symboles, il est préférable de générer le minimum de variables temporaires.

2. Les triplets

Un triplet est une structure plus économique que celle de quadruplet, le triplet est composé de 3 champs pour ses 3 éléments, l'opérateur et les deux opérandes.

Le n° de triplet est utilisé pour faire référence au résultat de l'opération réalisée par le triplet, on peut donc minimiser le nombre de variables temporaires utilisées.

Exemple

Toujours le même exemple

a := b* -c + b* -c

	Op	Arg1	Arg2
<0>	Moins	c	
<1>	*	b	<0>
<2>	Moins	c	
<3>	*	B	<2>
<4>	+	<0>	<3>
<5>	Affecter	a	<4>

3. Les triplets indirects

Le triplet indirect est une liste de pointeurs vers des triplets, cette liste indique l’ordre d’exécution des triplets.

Exemple

Toujours le même exemple

a := b* -c + b* -c

	Instruction
<0>	<12>
<1>	<13>
<2>	<14>
<3>	<15>
<4>	<16>
<5>	<17>

	Op	Arg1	Arg2
<12>	Moins	c	
<13>	*	b	<0>
<14>	Moins	c	
<15>	*	B	<2>
<16>	+	<0>	<3>
<17>	Affecter	a	<4>

3. Production du code à 3 adresses

Pour créer le code à 3 adresses d'un programme donné, on doit construire une DDS ou un ST dans lesquels on associe des règles sémantiques à la grammaire reconnaissant le programme.

3.1. Expressions arithmétiques et Affectation

La forme générale d'une instruction d'affectation est id := E, où E est une expression arithmétique.

Le code à 3 adresses pour cette instruction est composé de la séquence évaluant E dans un temporaire, suivie par l'affectation de ce temporaire dans id.

On va construire une DDS S-attribuée, cette DDs utilise deux attributs associés à E :

- E.place : cet attribut contient la valeur de E
- E.code : cet attribut contient la séquence de code à 3 adresses permettant d'évaluer E.

On utilise également deux fonctions :

- NouveauTmp : pour créer une nouvelle variable temporaire de la forme t_i
- Générer : pour générer le code à 3 adresses adéquat.

Règles de production	Règles sémantiques
I → id := E	I. code := E. code // Générer (id. place ' :=' E. place)
E → E_1 op E_2	E. place := NouveauTmp E. code := E_1. code // E_2. code // Générer (E. place ' :=' E_1.place 'op' E_2.place)
E → - E_1	E. place := NouveauTmp E. code := E_1. code // Générer (E.place ' :=' 'Moins'E_1.place)
E →(E_1)	E. place := E_1. place E. code := E_1. code
E → id	E. place := id. place E. code := ' '

// est l'opérateur de concaténation

3.2. Expressions booléennes

T UATL

Les expressions booléennes peuvent être évaluées de la même façon que les expressions arithmétiques.

Exemple

1) Pour l'expression 'non a et b ou c' , on peut donner le code à 3 adresses suivant :

t1 := non a

t2 := t1 et b

t3 := t2 ou c

2) Une expression relationnelle telle que 'a < b' est équivalente à 'si a< b alors 1 sinon 0', le code à 3 adresses de cette expression est :

20 : si a < b aller à 23

21 : t := 0

22 : aller à 24

23 : t := 1

À partir de ces exemples, on peut définir la DDS suivantes :

Règles de production	Règles sémantiques
E → E_1 ou E_2	E. place := NouveauTmp E. code := E_1. code // E_2. code // Générer (E. place ' :=' E_1.place 'ou' E_2.place)
E → E_1 et E_2	E. place := NouveauTmp E. code := E_1. code // E_2. code // Générer (E. place ' :=' E_1.place 'et' E_2.place)
E → non E_1	E. place := NouveauTmp E. code := E_1. code // Générer (E.place ' :=' 'non'E_1.place)
E →(E_1)	E. place := E_1. place E. code := E_1. code
E → vrai	E. place := NouveauTmp E. code :=Générer (E.place ' :=' '1')
E → faux	E. place := NouveauTmp E. code :=Générer (E.place ' :=' '0')
E → id_1 oprel id_2	E. place := NouveauTmp E. code := Générer ('si' id_1.place oprel id_2.place 'aller à' inst +3)//

3.3. Expressions conditionnelles

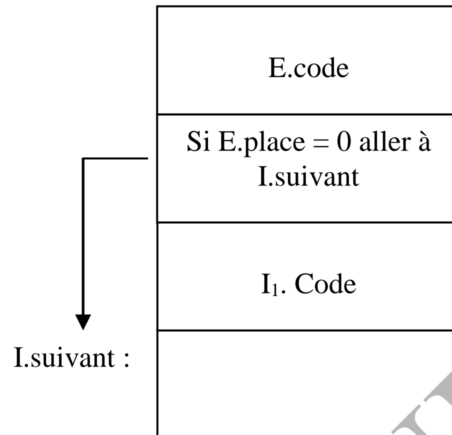
Une expression conditionnelle a la forme :

- I → si E alors I_1
- I → si E alors I_1 sinon I_2

1) I → si E alors I_1

Si E est vérifiée, on exécute donc I_1, sinon on va exécuter l'instruction qui vient après I.

Le schéma d'exécution de cette instruction est :



La DDS permettant de créer le code à 3 adresses de cette instruction est :

Règles de production	Règles sémantiques
I → si E alors I_1	I. suivant := NouveauEtiq I. code := E. code // Générer ('si'E.place ':=' '0' 'aller à' I.suivant) // I_1.code// Générer (I.suivant ' :')

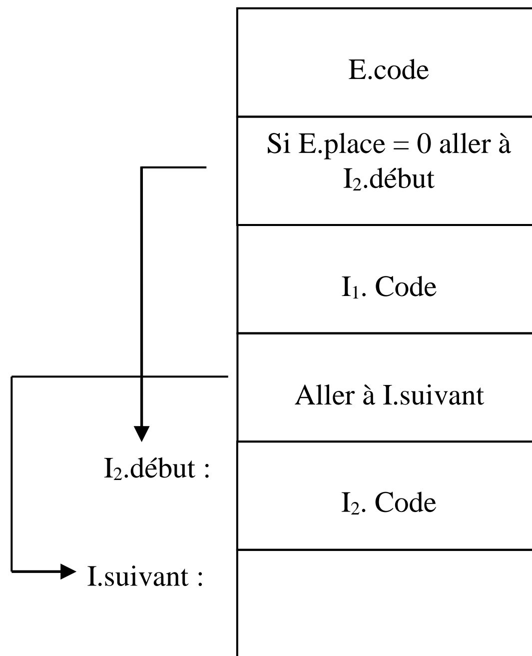
L'instruction NouveauEtiq permet de gréer une nouvelle étiquette.

2) I → si E alors I_1 sinon I_2

Si E est vérifiée, on exécute donc I_1, sinon on va exécuter l'instruction I_2.

Le schéma d'exécution peut être représenté comme suit :

T. ALLAOUI - UATL



La DDS sera :

Règles de production	Règles sémantiques
I → si E alors I_1 sinon I_2	I. suivant := NouveauEtiq I_2. début := NouveauEtiq I. code := E. code // Générer (‘si’E.place ‘:=’ ‘0’ ‘aller à’ I_2. début) // I_1.code// Générer (‘aller à’ I.suivant) Générer (I_2. début ‘ :’) // I_2. code // Générer (I. suivant ‘ :’)

3.4. La boucle Tant que

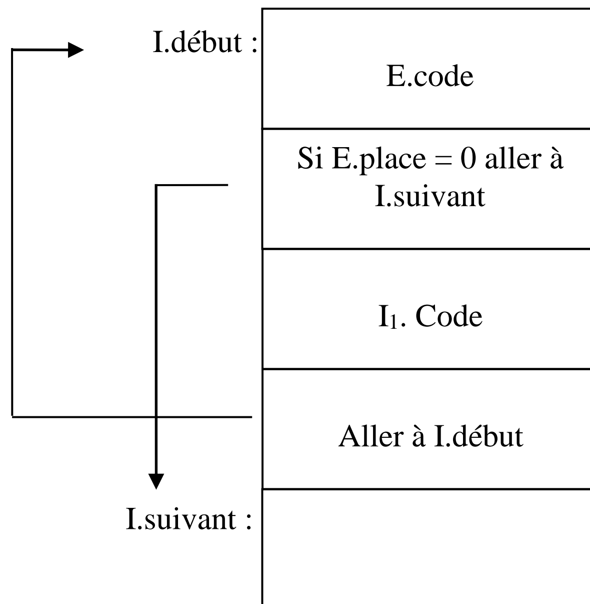
La forme de cette instruction est :

I → tant que E faire I_1

On exécute la boucle tant que la condition E est vérifiée.

Nous donnons le schéma d'exécution suivant

T. ALLAOUI - UATL



La DDS qui représente cette instruction est :

Règles de production	Règles sémantiques
I → tant que E faire I_1	I. début := NouveauEtiq I. suivant := NouveauEtiq I. code := Générer (I.début ‘ :’)//E. code // Générer (‘si’E.place ‘:=’ ‘0’‘aller à’ I. suivant) // I_1.code// Générer (‘aller à’ I.début) Générer (I. suivant ‘ :’)

Code Court-circuit

On peut optimiser le temps d’évaluation d’une expression booléenne, il est possible de ne pas évaluer entièrement l’expression.

Par exemple, dans une expression de la forme E → E_1 ou E_2, Si E_1 est vraie alors E est vraie sans vérifier E_2.

De la même façon, dans E → E_1 et E_2, si E_1 est fausse, alors E est fausse sans vérifier E_2.

Ce style d’évaluation est appelé le code court-circuit.

T. ALLAOUI - UATL

Les DDS utilisant le code court-circuit

On associe deux étiquettes à une expression booléenne :

- E.vrai : désigne l'instruction à exécuter si E est vraie.
- E.faux : désigne l'instruction à exécuter si E est fausse.

On peut réécrire les DDS précédentes de la manière suivante :

1) Les expressions booléennes :

Règles de production	Règles sémantiques
E → E_1 ou E_2	E_1.vrai := E.vrai E_1.faux := NouveauTmp E_2.vrai := E.vrai E_2.faux := E.faux E.code := E_1.code // Générer (E_1.faux ' :') // E_2.code)
E → E_1 et E_2	E_1.vrai := NouveauTmp E_1.faux := E.faux E_2.vrai := E.vrai E_2.faux := E.faux E.code := E_1.code // Générer (E_1.vrai ' :') // E_2.code)
E → non E_1	E_1.vrai := E.faux E_1.faux := E.vrai E.code := E_1.code
E →(E_1)	E_1.vrai := E.vrai E_1.faux := E.faux E. code := E_1.code
E → vrai	E. code :=Générer ('aller à' E.vrai)
E → faux	E. code :=Générer ('aller à' E.faux)

E → id_1 oprel id_2	E. code :=
	Générer ('si' id_1.place oprel id_2.place 'aller à' E.vrai)//
	Générer ('aller à' E.faux)//

2) Les instructions conditionnelles

Règles de production	Règles sémantiques
I → si E alors I_1	E.vrai := NouveauEtiq E.faux := I.suivant I_1.suivant := I.suivant I.code := E.code // Générer (E.vrai ' :') // I_1.code

Règles de production	Règles sémantiques

I → si E alors I_1 sinon I_2	E.vrai := NouveauEtiq
	E.faux := NouveauEtiq
	I1. suivant := I. suivant
	I2. suivant := I. suivant
	I. code := E. code //
	Générer (E.vrai ' :' ')// I_1. code //
	Générer ('aller à' I.suivant)
	Générer (E.faux ' :') // I_2. code //
	Générer (I. suivant ' :')

3) La boucle Tant que

Règles de production	Règles sémantiques

I → tant que E faire I_1	I. début := NouveauEtiq
	E.vrai := NouveauEtiq
	E.faux := I. suivant
	I_1.suivant := I.début
	I. code :=
	Générer (I.début ' :')//E. code //
	Générer (E.vrai ':') // I_1.code//
	Générer ('aller à' I.début)
	Générer (I. suivant ' :')

T. ALLA

Exercices

Exercice 1

Donner le code intermédiaire sous forme postfixée de l'instruction suivante :

A := 2 + (3 * 5) - 8

Exercice 2

Donner le code à 3 adresses des instructions suivantes :

x := 2 + 3 * A[6]

b := (x ≥ (y+1))

x := f(3,5)

Exercice 3

Ecrire le programme suivant en code à 3 adresses

```
x := 2 ;
y := 3 * x + 1 ;
For i := 1 à 10
faire
        x := x + 3 ;
        y :=2*x –y +1 ;
fait
z := x +y ;
```

Exercice 4

Traduire en quadruplets l'expression booléenne suivante :

a = b or b > c and not (d > a)

Exercice 5

1) Donner une DDS ou un ST permettant de générer le code à 3 adresses de l'instruction case du Pascal.

2) Proposer un exemple de cette instruction, et donner son code à 3 adresses en quadruplets.

T. ALLAOUI - UATL