Université Sidi Mohamed Ben Abdellah
Faculté des Sciences Dhar El-Mehraz Fès
Département d'Informatique

Principes et Techniques de Compilation

Par Noureddine Chenfour

1995-2010

1. Introduction

1.1 Historique et définitions

1.1.1 Définition

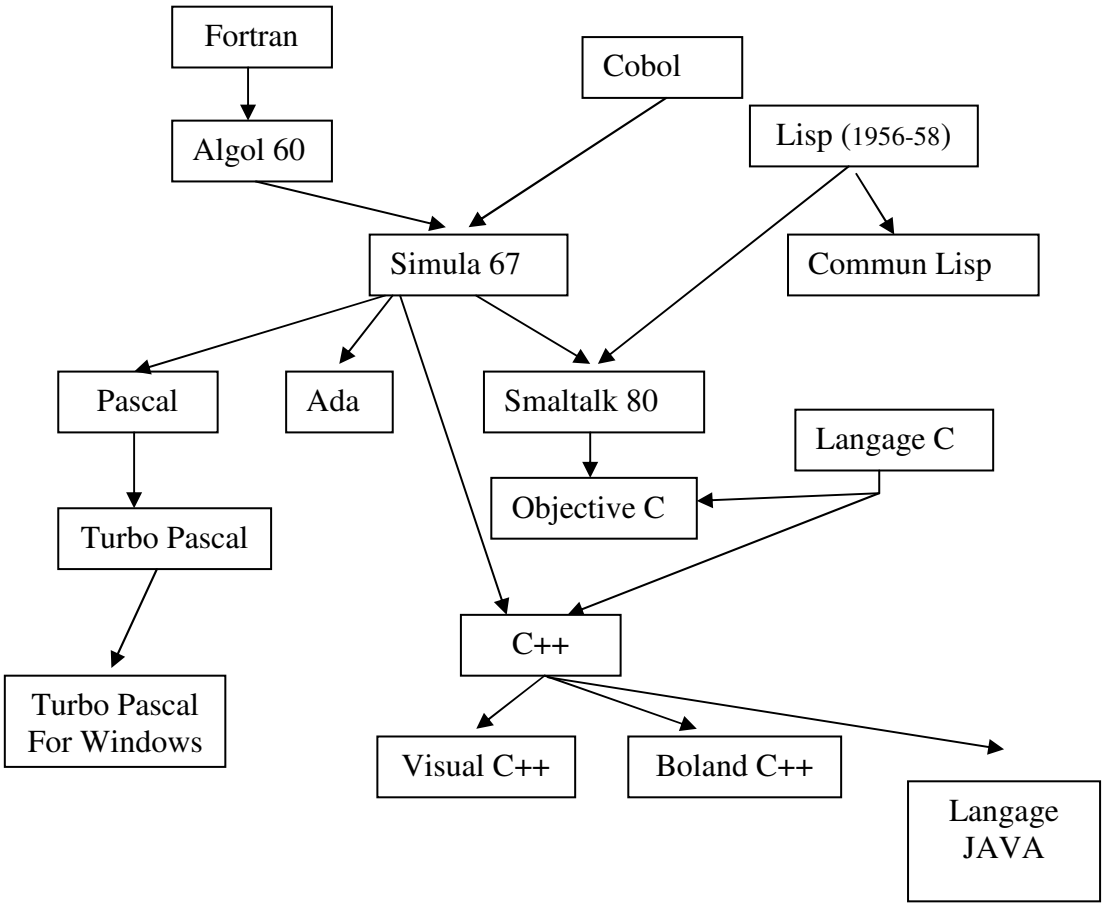
Un compilateur est un outil logiciel qui lit un programme source écrit dans un langage donné et le traduit en un programme correspondant écrit dans un autre langage. Cette opération est effectuée en signalant toute présence d'erreur dans le programme source.

1.1.2 Premiers Compilateurs

Les premiers compilateurs ont été réalisé très difficilement à cause du manque d'une démarche précise est automatisée. En effet parmi les premiers travaux effectué dans ce domaine étaient des programmes de traduction d'expressions arithmétiques en code machine. Cependant, le *langage d'assemblage* a demeuré le premier outil permettant d'écrire des programmes de plus en plus importants.

Au début des années 50, le compilateur *FORTRAN* (FORmula TRANslation) était en fin réalisé. Avec ce langage de programmation on pouvait écrire des programmes en utilisant une syntaxe plus évoluée et simple à manipuler.

L'expérience de réalisation du compilateur Fortran à conduit au développement de plusieurs techniques de construction systématiques d'un compilateur, qui seront étudiées dans ce cours. Ce qui a facilité par la suite la réalisation d'une multitude de compilateurs. Le schéma suivant montre l'évolution des compilateurs à travers différents domaines. On peut remarquer une amélioration structurelle au niveau de chaque nouveau langage.



1.1.3 Schéma symbolique d'un Compilateur

1.1.3.1 Langage binaire

C'est du code envoyé vers l'intérieur de la machine (microprocesseur, mémoire,..) à travers les bus, ayant une signification donnée (calcul, recherche, …) traduite par une configuration du système électronique interne et aboutissant au résultat désiré.

Exemple :

Le code : 0001 → chargement

0010 → opération d'addition

0011 → registre AX

Les variables sont aussi désignées par leurs adresses mémoires. On considère une variable X d'adresse 1000. Pour envoyer au microprocesseur la commande de charger la variable X dans le registre AX on écrit :

0001 0011 1000

qui peut se lire charger dans le registre AX la valeur de la variable X.

1.1.3.2 Langage d'assemblage

Un code en langage d'assemblage est une version mnémonique du code machine ou langage binaire, dans lequel on emploi des mots significatifs au lieu du code binaire pour designer les opérateurs, ainsi que des noms pour repérer les adresses mémoires (variables) et les registres (AX, BX, …).

Exemple :

```
MOV  AX, X
ADD   AX, Y
MOV  Z, AX
```

1.1.3.3 Assembleur

C'est le premier module pouvant être considéré comme un compilateur (de base), qui fait la traduction d'un programme écrit dans le langage d'assemblage en son équivalent en code machine. La forme la plus simple d'un assembleur effectue deux passes sur le texte d'entrée.

- 1ère passe : Consiste à lire le fichier d'entrée et assigner des adresses mémoires au identificateurs qui seront stockés dans une table appelée *Table de Symboles*.

Exemple :

Table de Symboles	
X	1000
Y	1010
Z	1100

- 2ème passe : Consiste à relire le texte est traduire chaque code opération (exemple MOV) et chaque nom de registre en la suite binaire qui le représente, et chaque identificateur en son adresse mémoire indiquée dans la table des symboles.

Exemple :

Le code assembleur de l'exemple précédent sera traduit comme suit :

```
0001 0011 1000
0010 0011 1010
0001 1100 0011
```

1.1.3.4 Interprète ou interpréteur

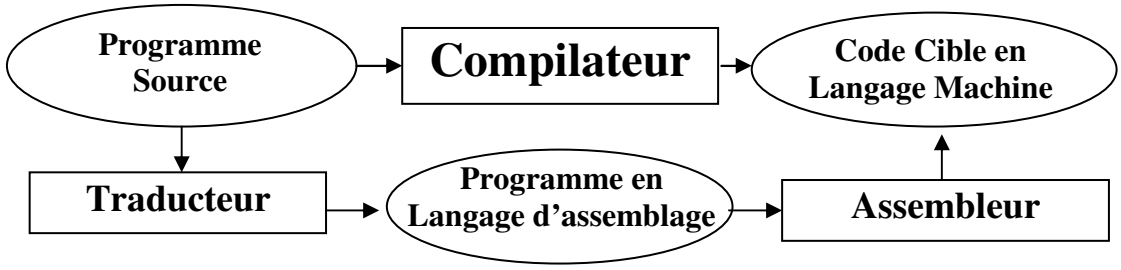
Un interpréteur ne produit pas du code final comme le cas des compilateurs, mais il procède lui même à l'exécution des instructions du programme source après leur interprétation. Il existe plusieurs langage qui disposent d'un interpréteur au lieu d'un compilateur.

Exemple :

Basic, Dbase, Lisp.

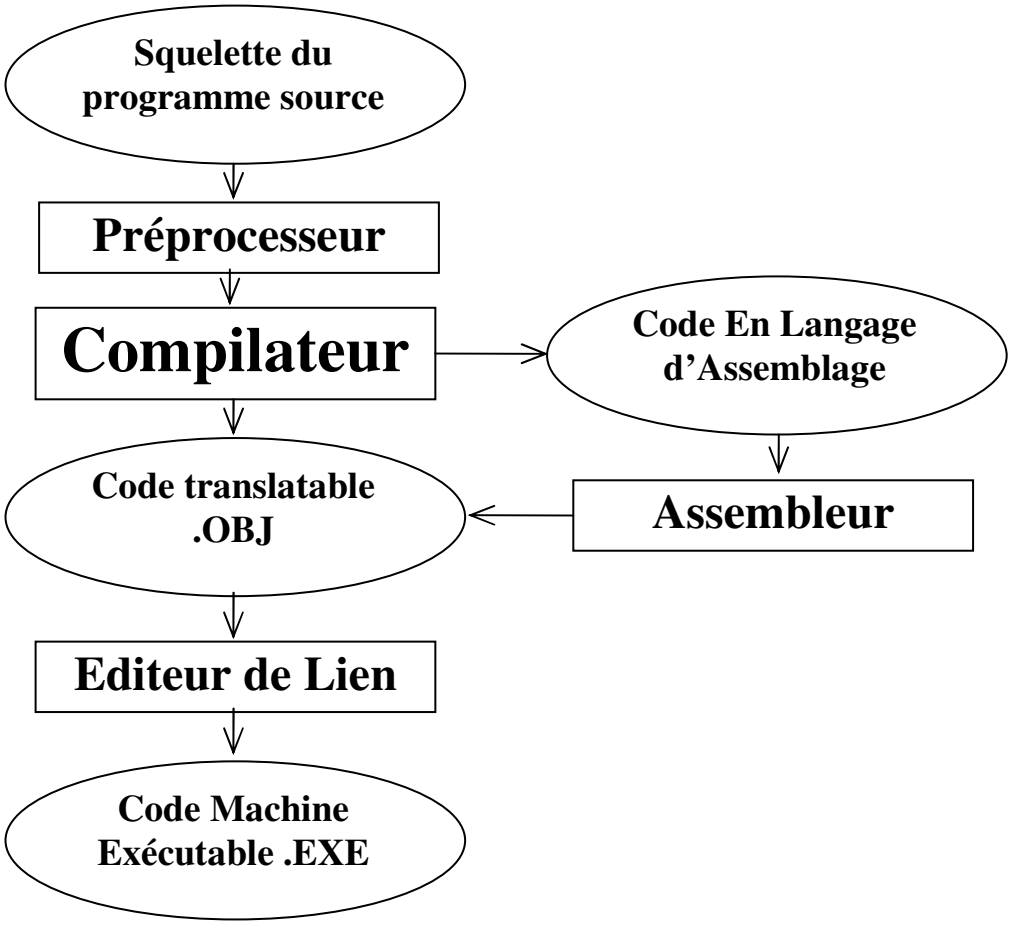
1.1.3.5 Schéma d'un Compilateur

Un compilateur reçoit le code d'entrée du programme source et le transforme en code machine. Cette opération et réalisée soit directement soit en traduisant tout d'abord le code source en langage d'assemblage qui sera délivré à l'assembleur qui le traduit à son tour en code machine.



1.2 Environnement d'un Compilateur

Pour qu'un compilateur puisse être complet et parvenir à générer du code machine exécutable, d'autres modules doivent être ajoutés à celui-ci sans en faire partie. Cette vision apporte une modularité de haut niveau facilitant le travail à la fois au programmeur (l'utilisateur du langage) grâce aux instructions du *préprocesseur* ainsi qu'au réalisateur du compilateur qui dispose déjà de deux module séparés : *l'assembleur* et *l'éditeur de lien*.



1.2.1 Le préprocesseur

Un programme source peut être divisé en modules sur des fichiers séparés. La tache de reconstitution du programme source est délivrée à un outils ne faisant pas partie du compilateur : le préprocesseur. Celui-ci s'occupe aussi du développement des abréviations ou macro-définitions (situées en général en tête du programme) en instructions du langage source. Ainsi, en langage C par exemple les instructions du préprocesseur sont présentées par le symbole # (#include , #define …) et doivent être transformées avant que le compilateur démarre son traitement. En effet, les instructions du préprocesseur ne respectent pas la syntaxe du langage et doivent alors suivre un pré-traitement à l'avance.

1.2.2 Editeur de lien (chargeur-relieur)

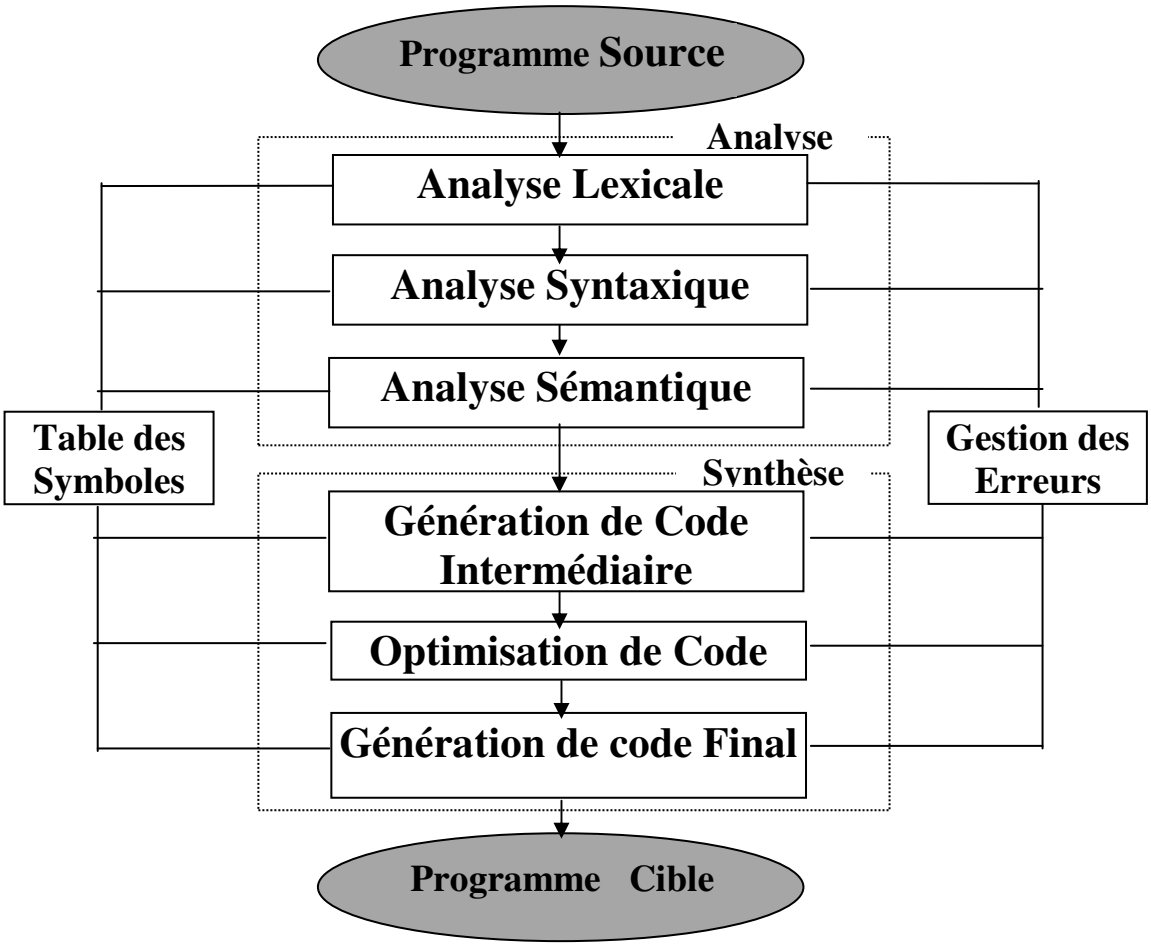
Après la phase de compilation on obtient du code en langage machine translatable (.obj). Ce code nécessite un traitement supplémentaire avant d'être exécutable. Il s'agit de l'édition de lien (Linkage) qui est confiée à l'éditeur de lien dont la tache est de relier le code machine obtenu à des routines précompilées des bibliothèque du langage. Une autre charge de l'éditeur de lien est la translation des adresses des différents codes liés. En effet les adresses d'un code .obj commence initialement à l'adresse 0, et doivent être translatées pour autoriser la fusion des différents code et aussi pouvoir occuper leur emplacement d'exécution en mémoire.

1.3 Phases d'un Compilateur

1.3.1 Définition d'une phase

Une phase est une opération transformant le programme d'une représentation à une autre. Cette transformation peut être seulement logique, c.à.d une opération qui ne crée pas de fichiers intermédiaires mais seulement une nouvelle connaissance ou décomposition du programme.

Un compilateur typique peut être décomposé en six phases représentées dans le schéma ci-après. Les trois premières phases constituent une première grande étape du compilateur : *l'analyse*. pendant cette étape, le compilateur se contente d'analyser (lexicalement, syntaxiquement et sémantiquement) le code source pour le préparer à l'étape suivante. La deuxième étape regroupe les phases restantes et sera appelée phase de *synthèse*. Lors de cette étape le compilateur génère le code cible correspondant.



1.3.2 Définition d'une passe

Il arrive souvent que l'on regroupe les activités de plusieurs phases dans un seul module appelé passe. Chaque passe est en général caractérisée par une relecture du fichier source et une production d'un nouveau fichier.

Il existe une grande variété de compilateurs, mais en général on en distingue deux type :

- Compilateurs à une passe.
- Compilateur multi-passes.

Il est préférable de n'utiliser qu'un nombre réduit de passes (deux par exemple) pour réduire le temps de compilation.

1.3.3 Analyseur lexical (scanner)

L'analyseur lexical constitue la première phase du compilateur. Sa tache principale est d'isoler en fonction de règles lexicales simples une suite d'entités primaires ou unités lexicales appelées lexèmes, composant le programme écrit dans un langage donné.

1.3.4 Analyseur syntaxique (parser)

C'est un module du compilateur qui vérifie que les entités en entrée apparaissent conformément aux règles définissant la syntaxe du langage. Il regroupe les lexèmes en structures grammaticales pouvant être représentées par un arbre syntaxique.

1.3.5 Analyseur sémantique

Il est chargé de contrôler les structures syntaxiquement correctes pour vérifier si elles ont un sens logique dans le programme. Le contrôle de type (correspondance de type) constitue la tache principale de ce module.

1.3.6 Générateur de code intermédiaire

Il transforme le programme ainsi analysé en une représentation intermédiaire à partir de laquelle il sera ensuite plus facile de générer n'importe quelle type de code final.

1.3.7 Optimiseur de code

Ce module a pour but d'améliorer le code intermédiaire de façon à ce que le code machine résultant sera plus rapide à exécuter. Pour cela certaines instructions seront remplacées par d'autres ou éliminées dans le cas de redondance.

1.3.8 Générateur de code final

C'est la phase finale d'un compilateur, qui consiste à la production du code final à partir du code intermédiaire. Exp : code machine, code en langage d'assemblage.

1.3.9 Traitement des erreurs

Au cours des phases de la compilation d'un programme, toute présence d'erreur doit être signalée et traitée de façon à ce que la compilation puisse continuer et que d'autres erreurs puissent être détectées.

1.4 Outils pour la construction automatique d'un compilateur

Grâce à la présence de méthodes systématiques pour la construction d'un compilateur, deux outils complémentaires ont été développés pour la construction automatique d'un compilateur à l'aide d'une simple spécification des unités lexicales et de la syntaxe du langage.

1.4.1 Constructeur d'un analyseur lexical (Lex)

Lex est un outil pour la construction automatique d'un analyseur lexical. Il reçoit un fichier de spécification des différentes unités lexicales du langage à l'aide de la notation des expressions régulières, et il produit un programme (source) qui permet de faire l'analyse lexicale d'un fichier d'entrée.

1.4.2 Constructeur d'analyseur syntaxique (YACC)

YACC (Yet Another Compiler Compiler) – Déjà un compilateur de compilateur) est un outil permettant de construire d'une manière automatique un analyseur syntaxique dans lequel on peut aussi intégrer toutes les phases d'un compilateur jusqu'à la génération de code. Ainsi YACC reçoit une spécification de la syntaxe du langage fondée sur la notation des grammaires, et il travail en collaboration avec Lex qui lui fourni le module d'analyse lexicale.

Chapitre 2

Analyse Lexicale

2.1 Définition

Un analyseur lexical constitue la première phase d'un compilateur. C'est une machine dont la matière première est l'ensemble des caractères du texte à compiler et dont la sortie est une séquence d'unités lexicales que l'analyseur syntaxique récupère ensuite. Les unités lexicales produites et plus précisément les identificateurs seront stockées dans une table appelée *table des symboles*.

Remarques :

1 - L'opération de lecture du texte source doit être accompagnée de taches supplémentaires selon le langage étudié. Par exemple éliminer les blancs du texte ou convertir le texte en majuscule.

2 - L'analyseur lexical doit signaler toute présence d'erreur dans le texte d'entrée. Il ne doit pas s'arrêter devant la première erreur rencontrée, mais il doit être capable de continuer le traitement du programme source et en détecter d'autres. Peut d'erreurs sont détectables à ce niveau car l'analyseur lexical à une vision très restreinte sur le programme source. Quelques erreurs pouvant être détectées à ce niveau sont :

- Un caractère n'appartenant pas à l'alphabet du langage,
- Un commentaire non fermé,
- Une chaîne de caractère non terminée
- Une constante caractère contenant plus que 1 ou 2 caractères.

2.2 Lexèmes et Unités lexicales

L'analyseur reçoit en entée une séquence de caractères tirés à partir d'un fichier texte, qu'il doit rassembler unité par unité chacune avec sa définition. Les unités extraites du fichier sont appelées des **lexèmes**. Soit par exemple le fichier contenant la ligne suivante :

Alpha,20.5 "traitement par les Afd" …

Les lexèmes qu'on peut extraire sont les suivants :

- Alpha qui est un identificateur

- , qui est un symbole
- 20.5 un nombre
- "traitement par les Afd" qui est une chaîne littérale.

Le type de chacune des unités constitue toute une classe à laquelle il peut appartenir différents lexèmes. Par exemple la classe des identificateurs contient tous les identificateurs qu'on peut imaginer respectant seulement la règle : suite de caractères alphanumériques débutant par un caractère alphabétique. Une classe de lexèmes (de même catégorie) constitue se qu'on appelle une **unité lexicale**. A chaque lexème on associe alors une unité lexicale bien déterminée. Une même unité lexicale peut être associée à plusieurs lexèmes. Comme exemple d'unités lexicales on peut citer : *identificateur, nombre, symbole, chaîne littérale, séparateur, opérateur de relation, opérateur arithmétique, mot clé, etc.*

On défini alors une **unité lexicale** comme un symbole qui représente une classe d'entités tirées à partir du fichier source et obéissant à une même règle. L'unité lexicale est donc un symbole terminal qui rentre dans la constitution du vocabulaire du langage source.

Un **lexème** est une suite de caractères du programme source qui constitue une entité élémentaire représentant une unité lexicale donnée.

Exemple :

Lexème	**Unité lexicale**
Alpha	Identificateur
While	Mot clé
20.5	Nombre

Avant donc de commencer l'analyse d'un fichier texte, il est nécessaire de préciser quelles sont les unités lexicales qu'on peut accepter. Celles-ci seront alors décrites par l'intermédiaire d'une représentation adéquate qui s'apprête à être implémenter sous forme de programme d'analyse. la meilleure notation utilisée est celle des expressions régulières. Les avantages d'une expression régulière sont les suivants :

1- Une expression régulière est une notation simple et universelle.

2- Une expression régulière peut être transformée systématiquement en un programme d'analyse. Cette transformation passe par les étapes suivantes :

- Transformation Expression Règulière ➔ Automate Fini Non déterministe (AFN)
- Transformation AFN ➔ Automate Fini Déterministe (AFD)
- Transformation AFD ➔ Programme d'analyse

Les trois transformations sont systématiques et peuvent être complètement automatisées.

2.3 Expressions régulières

2.3.1 Alphabet

C'est un ensemble fini et non vide d'éléments appelés des symboles ou caractères.

Exemples :

A = { 0, 1 } Alphabet binaire pour la construction du langage machine.

A = {a, b, c, !, %} est aussi un alphabet.

2.3.2 Chaîne

C'est une séquence finie de symboles tirés d'un alphabet donné pour constituer une seule entité.

Exemple :

Soit l'alphabet A = { a, b, c, d, e } ,

ε, a, abc, bbe, dd sont des chaînes de l'alphabet A.

2.3.3 Opérateurs de base

2.3.3.1 Opérateur de Concaténation : « . »

La concaténation de deux chaînes est une chaîne formée par les symboles de la première chaîne suivi de ceux de la deuxième. On note :

S1 **.** S2 ou tout simplement : S1 S2

2.3.3.2 Opérateur d'Union : « | »

On utilise l'opérateur d'union « | » pour designer l'une ou l'autre de deux chaînes.

Exemple :

abc | ce signifie l'une des deux chaînes.

2.3.3.3 Opérateur de Fermeture : « * »

l'opérateur de fermeture * est utilisé pour designer la concaténation d'une chaîne avec elle même un nombre quelconque de fois (éventuellement nul => ε).

Exemple :

(ab) * signifie : ε | ab | abab | ababab ...

2.3.4 Opérateurs supplémentaires

2.3.4.1 Opérateur de Fermeture positive : « + »

l'opérateur de fermeture positive + est utilisé pour designer la concaténation d'une chaîne avec elle même une, 2 ou plusieurs fois.

$(S)^+$ notée aussi S^+ signifie : S | S S | S S S …

S^+ est équivalente à : S S*

S* est équivalente à : ε | S^+

2.3.4.2 Opérateur d'option : « ? »

l'opérateur d'option ? est utilisé pour désigner la chaîne sur laquelle il est appliqué ε.

(S)? notée aussi S ? signifie : S | ε

2.3.5 Expression régulière

Une expression régulière est une notation utilisée pour la description d'une unité lexicale d'un langage. Toutes les unités d'un langage dit régulier sont dénotés par des expressions régulières.

Pour un alphabet *A* donné, les expressions régulières sont formées par les règles suivantes :

1) ε et les éléments de *A* sont des expressions régulières.
2) Si α et β sont deux expressions régulières alors α | β , α β , (α)*, $(\alpha)^+$et (α) ? sont des expressions régulières.

2.3.6 Définition régulière

pour des commodités de notation on peut avoir besoin de donner des noms aux expressions régulières et de définir des expressions régulières en fonction de ces noms. Une définition régulière est une suite de définitions de la forme :

$d_1 \rightarrow \quad exp_1$
$d_2 \rightarrow \quad exp_2$
…
$d_n \rightarrow \quad exp_n$

Avec d_i est le nom qui va représenter l'expression régulière exp_i, et expi étant définie sur les symboles de base de l'alphabet ainsi que les noms d'expressions définies avant d_i, càd exp_i est une expression régulière sur $A \cup \{d_1, \ldots, d_{i-1}\}$.

Les d_i sont alors les unités lexicales du langage.

Exemple :

lettre → a | b | … | z | A | … | Z
chiffre → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
id → lettre (lettre | chiffre)*

2.4 Les Automates à états Finis déterministe (AFD)

2.4.1 Définition

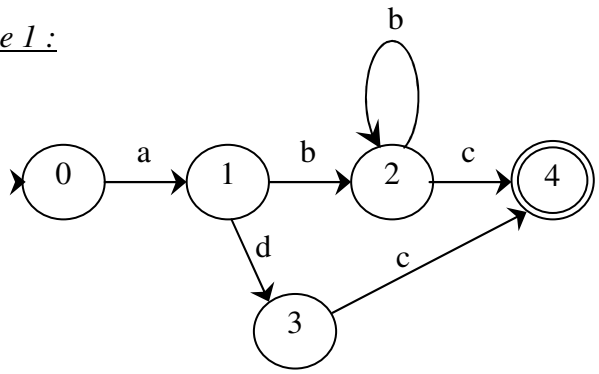
Un Automate Fini est un diagramme graphique permettant de préciser schématiquement les étapes qu'il faut suivre pour identifier et extraire des lexèmes dont l'expression régulière est bien définie. Il s'agit d'un ensemble d'états qui indique la progression d'un analyseur lexical sur le texte d'entrée. Le passage d'un état à l'autre est réalisé lorsqu'on accepte un symbole (caractère) bien déterminé.

Les états sont alors schématisés par des cercles. Le passage d'un état à l'autre est réalisé par l'intermédiaire d'un arc reliant entre les deux état et étiqueté par le symbole accepté.

Le dernier état sur lequel on se trouve après avoir reconnu un lexème est appelé **état final** ou **état accepteur**. Un état accepteur est en général schématisé par un cercle double.

Le premier état avec lequel on démarre l'analyse est appelé **état initial** (en général l'état 0) peut être schématisé à l'aide d'un cercle ordinaire pointé par une pointe de flèche.

Exemple 1 :



Cet automate représente l'expression régulière : a (b^+| d) c

Remarques :

1- Un Automate Fini dispose d'un seul état initial, mais il peut avoir plusieurs états accepteurs.

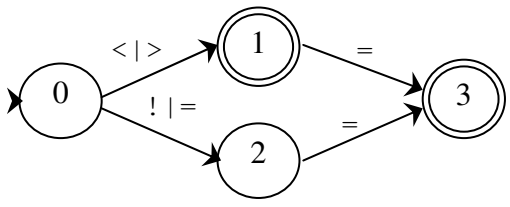
2- Une transition peut être réalisée sur un symbole ou une union de symboles mais jamais sur une concaténation de symboles

Exemple 2 :

Soit la définition régulière des opérateur de relation C :

```
oprel      →    < | <= | > | >= | != | ==
```

Elle est représentée par l'automate suivant :

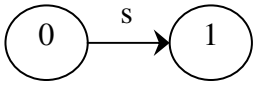


La transition de l'état 0 vers l'état 1 est réalisé si le symbole d'entrée est < ou >. On peut noter de la même manière <,>.

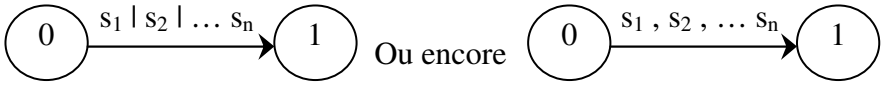
2.4.2 Règles élémentaires de construction d'un automate

A partir des exemples déjà traités, il est possible de déduire quelques règles élémentaires de construction d'un automate à partir d'une expression régulière. Elles peuvent être énoncées comme suit :

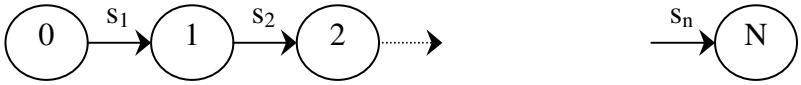
1- Si l'ER est un simple symbole (s) :



2- Si l'ER est une Union de symbole ($s_1 \mid s_2 \mid \dots s_n$) :

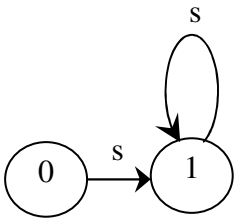


3- Si l'ER est une concaténation de symboles ($s_1\ s_2 \dots s_n$) :



4- Si l'ER est une fermeture (s*) :

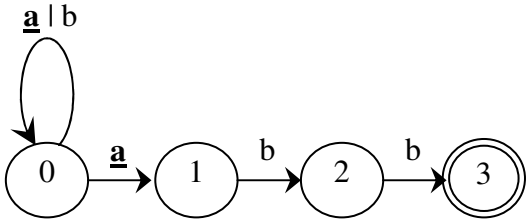
5- Si l'ER est une fermeture positive (s^+) :



En appliquant ces règles à l'expression régulière suivante :

(a | b)* a b b

on obtient l'automate suivant :

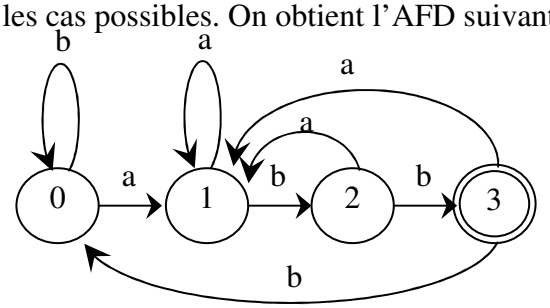


Cependant celui-ci est *Non Déterministe*. En effet, à partir de l'état 0 si on reçoit le symbole « a », le prochain état peut être 0 ou 1. Il s'agit alors d'un *non déterminisme* : il n'est pas possible de déterminer d'une manière unique le prochain état vers lequel on va transiter. L'automate obtenu est appelé « *Automate Fini Non déterministe* » et noté **AFN**. En contre partie un **AFD** (*Automate Fini Déterministe*) est un automate dans lequel les décisions de transition sont uniques. La relation de transition est alors une *fonction*. A un état de départ et à un symbole en entrée on associe un seul état destination.

Remarque :

Il est parfois plus simple de construire un AFN à partir d'une expression régulière que de construire un AFD.

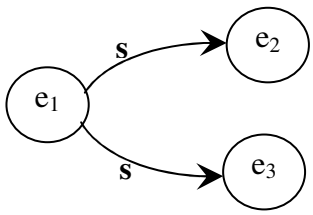
Pour l'exemple précédent, il est possible de construire un AFD correspondant avec plus de raisonnement au niveau de chaque état de l'automate où il est nécessaire de traiter tous les cas possibles. On obtient l'AFD suivant :



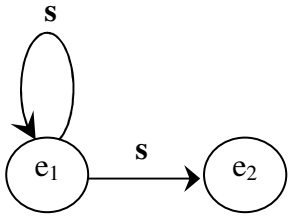
2.4.3 Automates Finis Non déterministes

Un AFN est un automate qui présente l'une des deux situations suivantes :

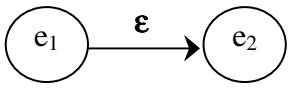
1- A partir d'un même état de départ, il existe deux arc sortant sur le même symbole, partant vers deux états différents.



En cas particulier :



2- Il existe une transition sur le symbole ε, appelée ε-transition. Celle-ci signifie un changement d'état sans consommation de symbole. Ce qui est un non déterminisme : faut-il ou non changer d'état quand rien ne se passe en entrée.



2.4.4 Automates Finis Déterministes (AFD)

2.4.4.1 Définition 1 :

Un AFD ou encore Automate à états Finis Déterministe est un automate ne présentant aucune des deux situations de non déterminisme précédentes.

2.4.4.2 Définition 2 :

Un AFD est un Quintuplet $(\sum, e_0, E, T, A)$ où :

$\sum$	Alphabet du langage utilisé par l'automate
e_0	Etat Initial
E	Ensemble des états de l'AFD
T	**Fonction** de Transition
A	Ensemble des états Accepteurs

2.4.5 Construction d'un AFD à partir d'une expression régulière

La construction passe par 2 étapes :

1. *Etape N° 1 :*

   Transformation ER ➔ AFN

2. *Etape N° 2 :*

   Transformation AFN ➔ AFD

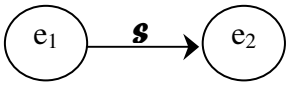
2.4.5.1 Construction d'un AFN à partir d'une Expression régulière

Dans la suite les règles de transformation de base ; qui, appliquées récursivement, elles permettent de transformer automatiquement n'importe quelle expression régulière en un AFN.

Règle N° 1 :

Soit **A** l'alphabet du langage.

Pour tout symbole ***s*** appartenant à **A**, l'automate correspondant est le suivant :



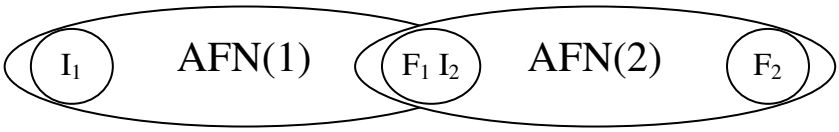
Règle N°2 :

Soient les deux expressions régulières suivantes :

ER1 ayant comme AFN : *AFN(1)*

ER2 ayant comme AFN : *AFN(2)*

L'AFN de la concaténation *ER1* . *ER2* est le suivant :



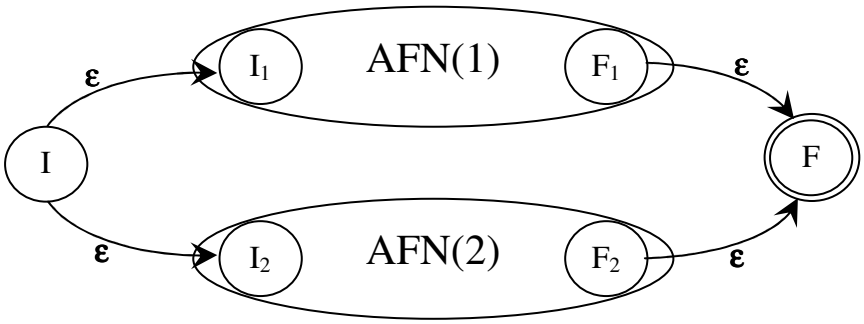
Règle N°3 :

Soient encore les deux expressions régulières suivantes :

ER1 ayant comme AFN : *AFN(1)*

ER2 ayant comme AFN : *AFN(2)*

L'AFN de l'union *ER1* | *ER2* est le suivant :

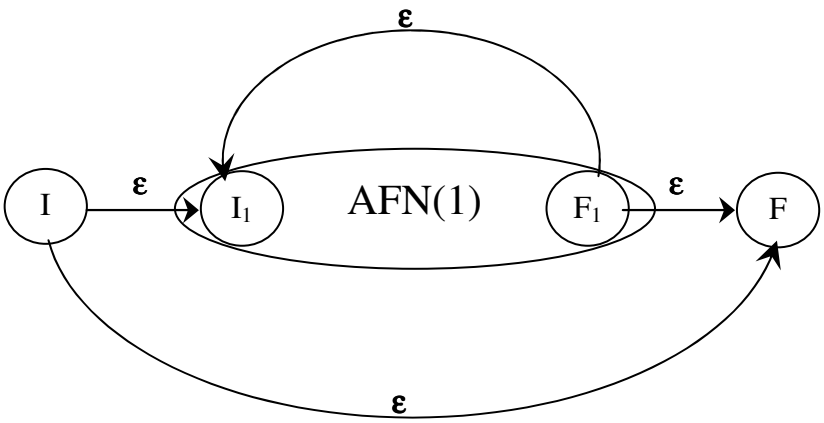


Règle N°4 :

Soit l'expression régulière suivante :

ER1 ayant comme AFN : *AFN(1)*

L'AFN de la fermeture *ER1** est le suivant :

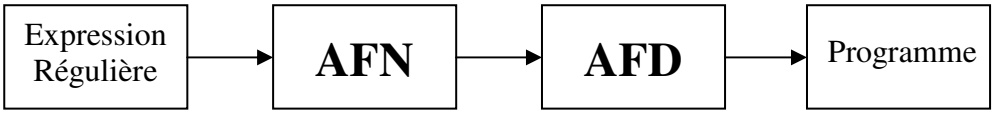


2.4.5.2 Construction d'un AFD à partir d'un AFN

2.5 Implémentation d'un analyseur lexical

Pour implanter un analyseur lexical, une phase de conception est tout d'abord nécessaire. Pendant cette phase toutes les unités lexicales sont spécifiées à l'aide de la notation des expressions régulières. Ensuite, ces expressions sont transformées en des AFD (Automates Finis Déterministes). Puis, les AFD sont transformés facilement en un programme.

La transformation des expressions régulières en des AFD est une opération qui peut être réalisée intuitivement comme elle peut être développée automatiquement par un programme à l'aide d'un passage par des AFN (Automates Finis Non déterministes).

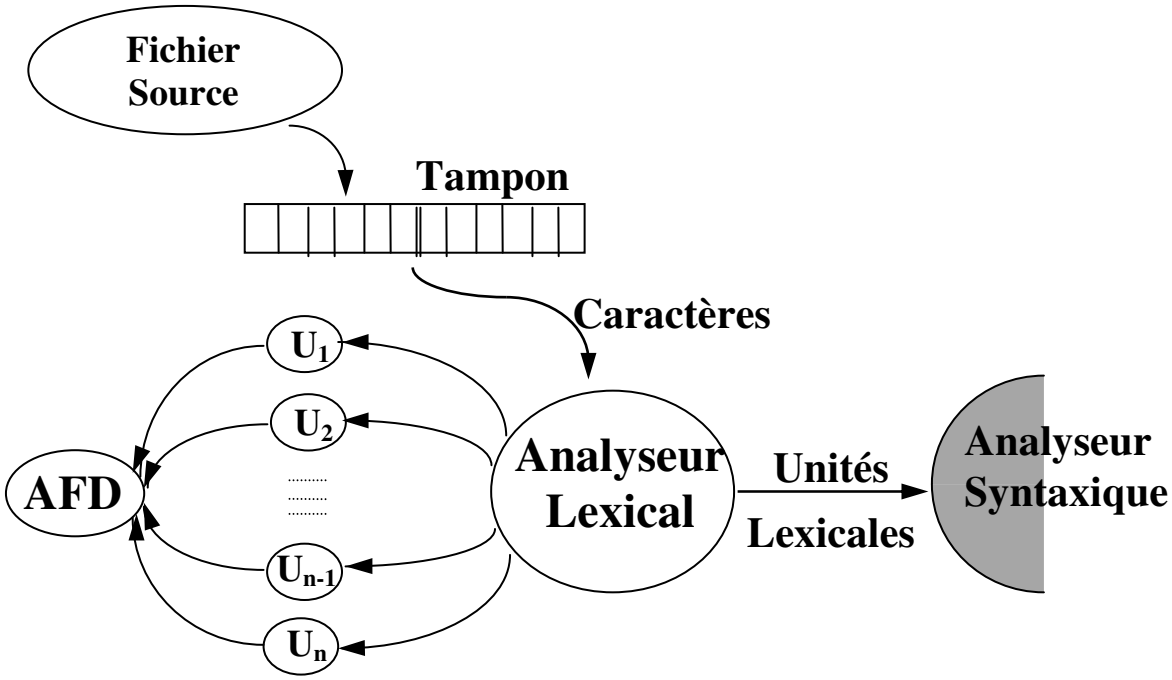


Dans une première partie de ce paragraphe nous présentons l'implantation d'un analyseur Lexical basé sur les AFD. Dans une deuxième partie nous allons faire une description des algorithmes de construction d'un AFD à partir d'une Expression Régulière. Nous supposons alors dans un premier lieu que la transformation des ER en AFD est une opération évidente.

Extraction d'une unité lexicale

Schéma général de l'analyseur lexical

- **Schéma général**



- **Diagramme de classe**

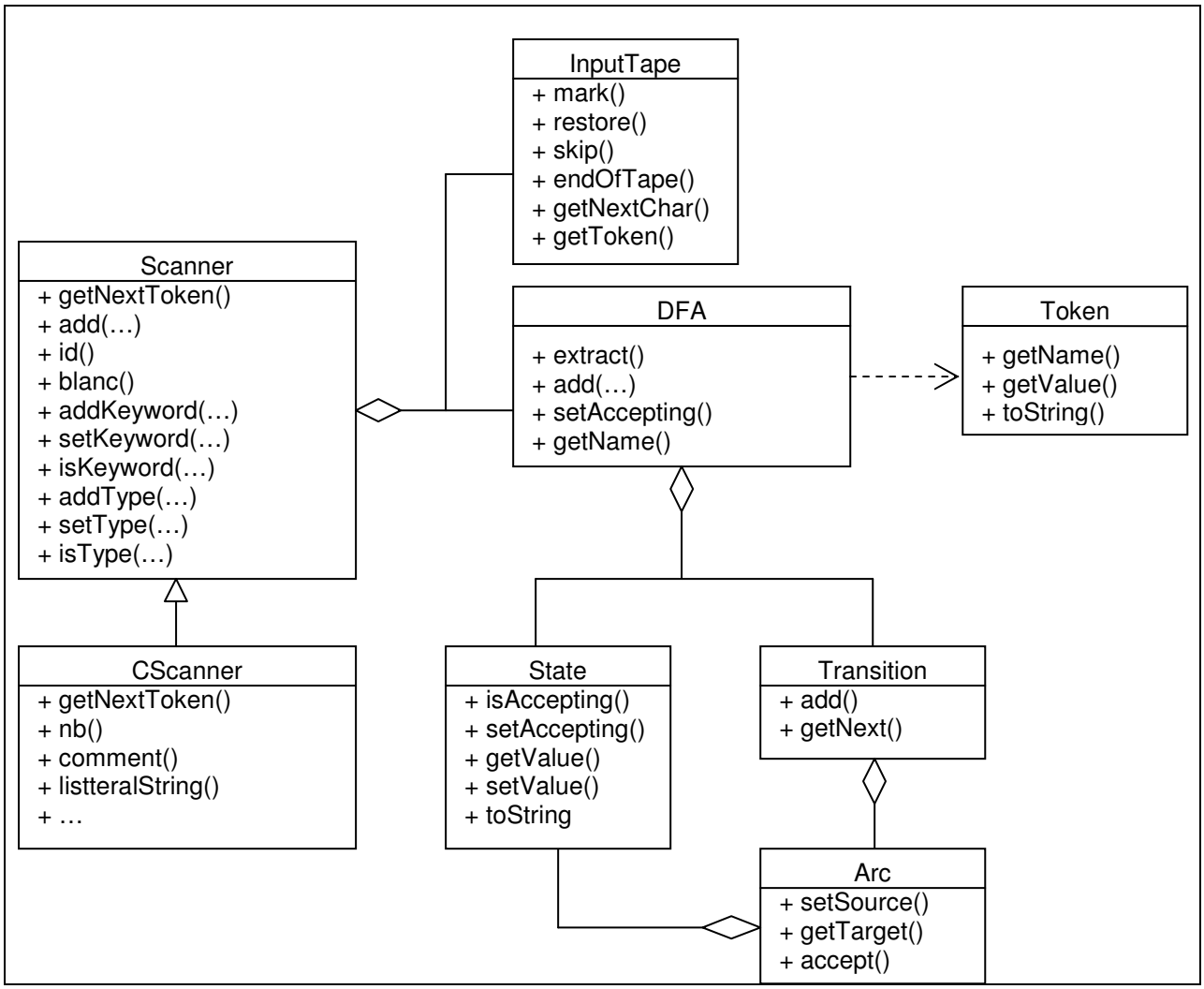


Fig 2.2 : Diagramme de Classe UML de l'analyseur Lexical

Chapitre 3

Analyse Syntaxique

3.1 Objectif et outils de base

- Vérifier si les éléments qui proviennent de l'analyseur lexical forment une chaîne d'unités lexicales acceptées ordre est représentation par le langage.
- La représentation du langage est spécifiée par une **grammaire**.
- La grammaire est définie en général à l'aide des unités lexicales qui constituent le **vocabulaire terminal**, et des nouvelles entités qui définissent les constructions du langage et forment un **vocabulaire non terminal**.

$\Rightarrow$ La première tache à effectuer est la constitution de la grammaire.

Exemple :

E $\rightarrow$ E + T | E -T | T

T $\rightarrow$ T * F | T / F | F

F $\rightarrow$ (E) | nombre | identificateur

3.2 Définition

L'analyseur syntaxique est la 2ème phase du compilateur. C'est un module qui reçoit en entrée une liste de lexèmes (unités lexicales) extraits par l'intermédiaire de l'analyseur lexical (Scanner) et vérifie si leur ordre d'apparition dans le fichier d'entrée est conforme à la syntaxe prévue par le langage. La syntaxe doit donc être définie d'une manière régulière et sans ambiguïté. Ceci nécessite une spécification de la syntaxe réalisée à l'aide d'un langage de spécification syntaxique simple et universelle. Il s'agit de la notation des **grammaires**.

3.3 Les grammaires

3.3.1 Vocabulaire Terminal

L'analyseur syntaxique reçoit une suite de lexèmes et d'unités lexicales qui constituent un ensemble d'entités lexicales utilisées pour définir la grammaire du langage, appelé VOCABULAIRE TERMINAL noté VT.

Exemple :

```
VT = {id, nb, :=, begin, end, ...}
```

3.3.2 Règles de Production et Vocabulaire Non-Terminal

Pour définir la syntaxe du langage, on utilise des règles syntaxiques appelées **Règles de Production** (ou simplement des **Productions**). Chaque règle permet de fixer un ordre syntaxique bien déterminé au niveau d'une construction donnée du langage. Ainsi, à chaque construction on associe un nom qui la détermine et pouvant être réutilisées pour définir d'autres constructions du langage. Ce nom est appelé un non-terminal et constitue un élément du VOCABULAIRE NON-TERMINAL noté VN.

Une production sera alors définie comme suit :

```
Partie Gauche → Partie Droite
```

La partie gauche est le non-terminal, la partie droite est une suite de **terminaux** et de **non-terminaux** fournissant ainsi la structure de la construction syntaxique identifiée par le non terminal à gauche.

Exemple : déclaration d'une liste d'indentificateurs en Pascal

Dec → id LI : type ;

LI → , id LI | ε

Dans ce cas :

VT = { id : ; , type }

VN = { Dec, LI }

Remarque :

Pour distinguer entre les terminaux et les non-terminaux, ces derniers vont toujours commencer par une lettre majuscule

3.3.3 Définition d'une grammaire

Une grammaire est définie sous forme d'une liste de productions permettant de couvrir toutes les possibilités syntaxiques du langage. Elle est définie régulièrement comme un quadruplet : **G(VT, VN, P, S)**

VT : Vocabulaire Terminal

VN : Vocabulaire Non terminal

P : un tableau de productions

S : appelé Axiome de la grammaire, est le premier non terminal définissant la première production de la grammaire.

3.3.4 Règles d'écriture des grammaires

1. Les seuls opérateurs utilisés pour définir les grammaires sont :
   - la concaténation : « . »
   - L'union : « | »

2. Les structures optionnelles sont définies dans des ***productions séparées*** en utilisant l'alternative ε :

   Pascal → Entete . Dec . Corps

   Entete → program id ; | ε

3. Les structures répétées sont définies à l'aide de la récursivité :

   Lid → id Rid

   Rid → , id Rid | ε

   Règles générales :

   Soit une construction α à répéter :

   3.1 Pour réaliser α^* :

   A → α A | ε

   3.2 Pour réaliser α^+ :

   A → α B

   B → α B | ε

Exemple :

Pascal	→	Entete Dec Corps .
Entete	→	program id ; | ε
Dec	→	D . Dec | ε
D	→	var Lid : type ;
Lid	→	id Rid
Rid	→	, id Rid | ε
Corps	→	begin LI end
LI	→	Inst LI | ε
Inst	→	IA | IIf | IBoucle | ….
….		

VT = {program id ; . var : type , begin end }
VN = {Pascal, Entete, Dec, D, Lid, Rid, Corps, LI, Inst, IA, IIf, IBoucle}
Axiome = Pascal

● ***Remarque 1 :***

Un Non-Terminal peut être défini à base d'un autre Non-Terminal qui sera défini en dessous.

Exemple :

A ⟶ a B | C

B ⟶ B X | d

X ⟶ b C | e

● ***Remarque 2 :***

L'opérateur d'option n'existe pas, il est remplacé par l'alternative ε

A ⟶ a | ε

● ***Remarque 3 :***

Les alternatives d'une production peuvent être définies séparément :

A ⟶ B | a C | d

⟺

A ⟶ B
A ⟶ a C
A ⟶ d

● ***Remarque 4 :***

Les opérateurs de fermeture * et + sont remplacés par la récursivité

*) α^* ⟹ A ⟶ α A | ε

*) $\alpha^+ = \alpha . \alpha^*$

α^+ ⟹ A ⟶ α B

B ⟶ α B | ε

3.3.5 Principe d'utilisation des grammaires :

3.3.5.1 Dérivation :

Soient X, Y Є V^*

On dit que X se dérive en Y si le Y peut être obtenu à partir de X par application de 0, 1 ou plusieurs règles de Production. On écrit :

X ⟹ Y ou X ⟹ *Y

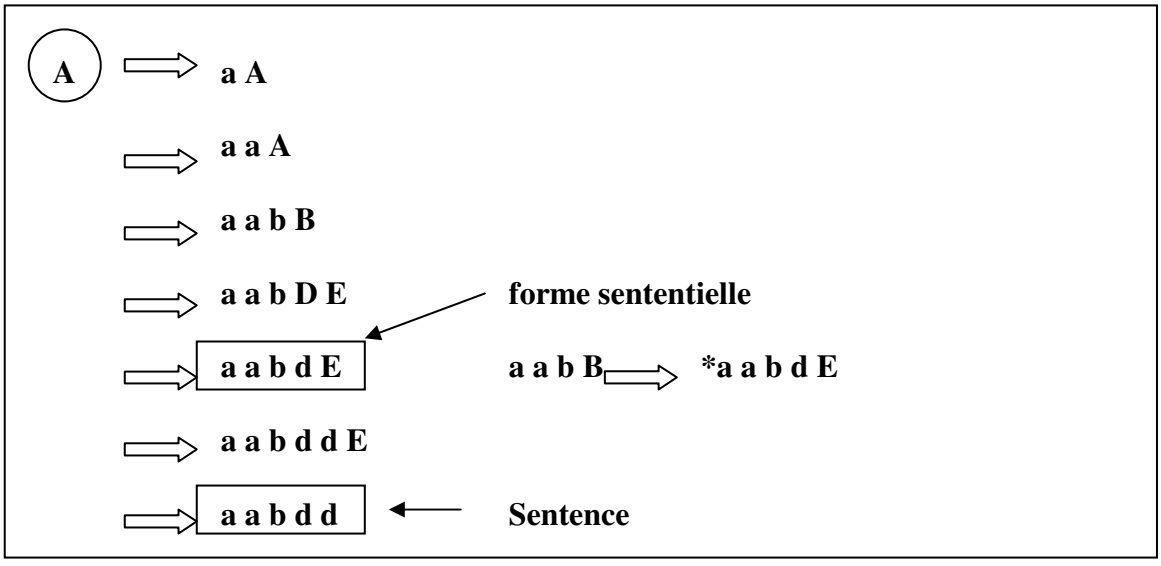
Exemple :

A ⟶ a A | b B

B ⟶ a A | D E

D ⟶ c | d

E ⟶ d E | Є



3.3.5.2 Forme sententielle – Protophrase

C'est une phrase de V^* qui se dérive de l'axiome.

Exemple : a B A D

A ⟹ a A

⟹ A b B n'est pas une forme sententielle

NB : Une forme sententielle est une forme d'instruction correcte.

Exemple :

-E : if n'est pas une forme sententielle de Pascal

-id := E ; est une forme sententielle pour le langage Pascal.

Pour la grammaire précédente : b D d E

A ⟹ b B

⟹ B D E

⟹ B D d E est une forme sententielle

3.3.5.3 Sentence :

C'est une forme sententielle qui appartient à V_t^*.

Exemple :

```
Begin
Id := nb * (id + nb ) ;
End.
```

Est une sentence du langage Pascal.

Exemple : Pour la grammaire précédente :

- b c d est une sentence

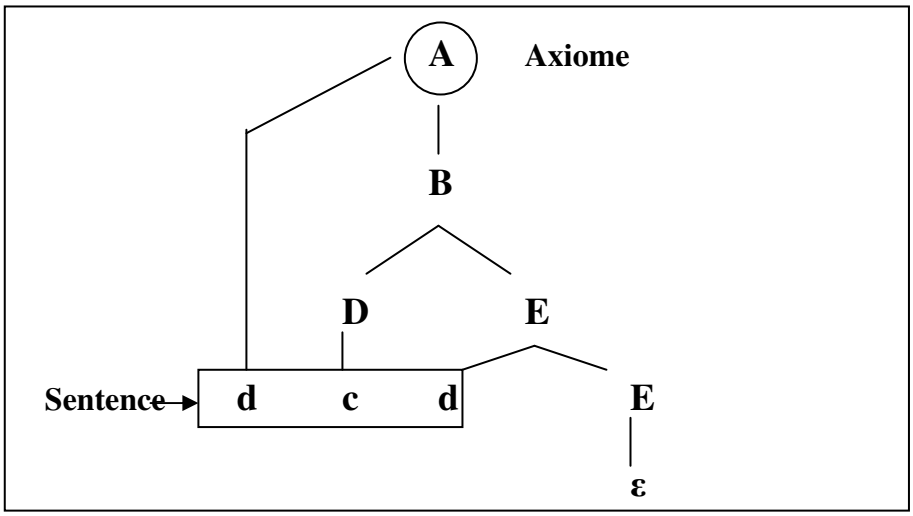
A ⟹ b D
⟹ B D E
⟹ b D d E
⟹ b c d E
⟹ b c d

3.3.5.4 Arbre syntaxique :

Un arbre syntaxique est le schéma des dérivations nécessaires pour obtenir une sentence depuis l'axiome.

L'axiome va définir la racine de l'arbre. Les feuilles de l'arbre sont les constituants de la sentence.

Exp1 :



Exp2 :

DP	→ Var	Sd	DP | ε
Sd	→ Ld	R	
R	→ Ld	R | ε	
Sid	→ , id	Sid | ε	

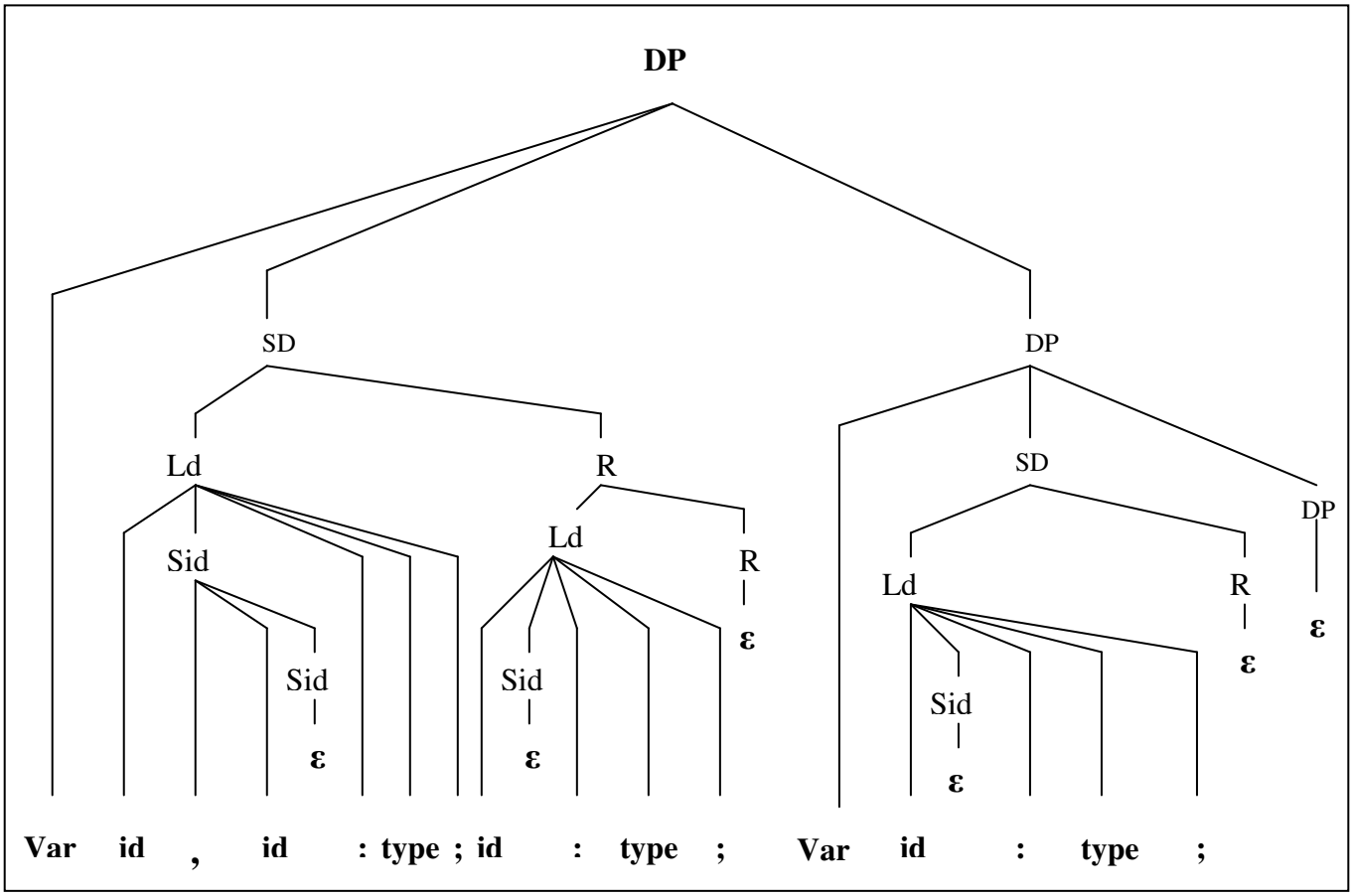
Exemple de déclaration :

```
Var     x, y : integer ;
          c  : char ;
Var       z  : real ;
```

Analyseur lexical ⇒

```
Var     id, id : type ;
          id : type ;
```

VN = { DP, Sd, R, Ld, Sid }
VT = { Var, id,: , type, ; , , }
Axiome : DP

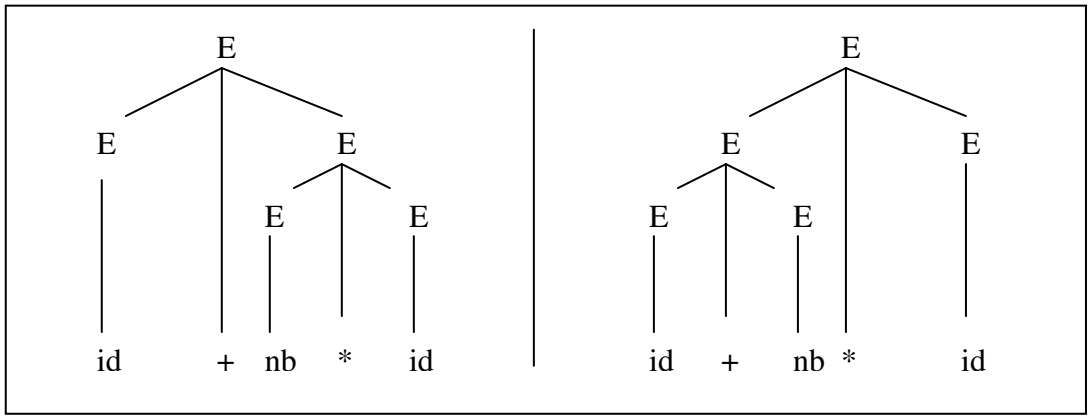


3.3.5.5 Grammaire ambiguë :

Une grammaire est dite ambiguë si à une même sentence correspond deux ou plusieurs arbres syntaxiques (en utilisant une dérivation gauche).

Exemple : E ⟶ E + E | E – E | E / E | E * E | id | nb | (E)

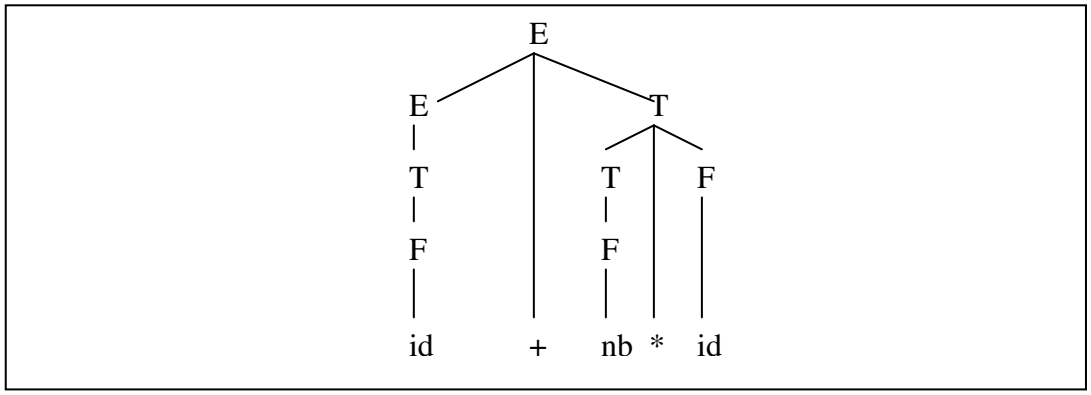
Soit la sentence : id + nb * id



Faux

Solution :

E ⟶ E + T | E – T | T
T ⟶ T * F | T / F | F
F ⟶ nb | id | (E)



⇨ Un seul arbre syntaxique.

3.4 Méthode générale

La méthode la plus simple pour la construction de l'analyseur syntaxique est la **méthode récursive descendante**.

⇨ Associer à chaque non terminal une fonction booléenne qui vérifie si la construction est correcte.

⇨ La construction est identifiée par la partie droite de la production associé au non terminal.

Inconvénients :

❍ Problème de rebroussement.

❍ Possible uniquement avec quelques grammaires.

3.5 Analyseur prédictif descendant (LL(1))

- Le problème de rebroussement est créé par les alternatives :

  $A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$

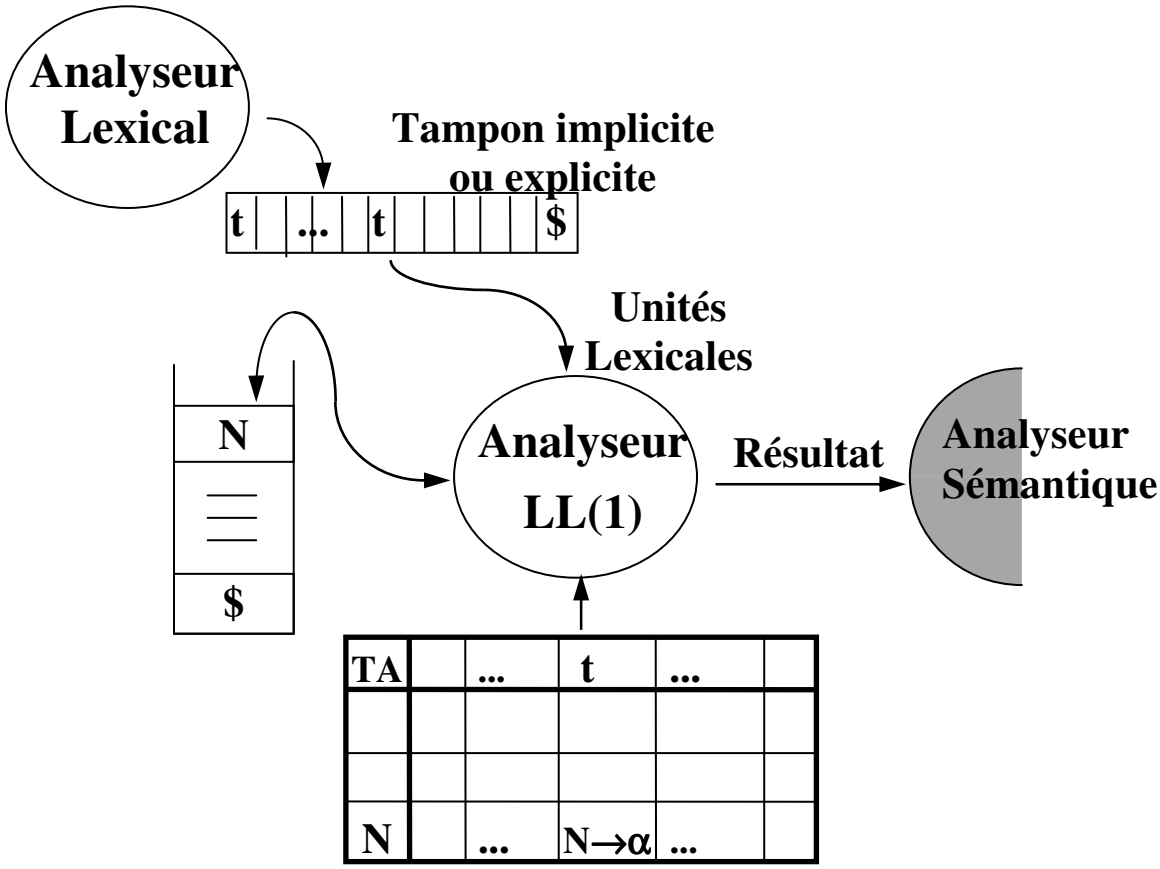
- Pour être déterministe on utilise la notion de **Symbole de Prévision**.

⇨ L'analyse prédictive s'appuie sur la connaissance des **premiers symboles** qui peuvent être engendrés par les parties droites des productions.

⇨ Définir les ensembles **First** et **Follow.**

⇨ Construire une table de décision appelée **Table d'Analyse** dans laquelle sont répertoriées les décisions de dérivation possibles.

⇨ L'analyseur utilise la table d'analyse pour prendre les décisions de dérivation adéquates et gère une **Pile d'Analyse** explicite pour y sauvegarder l'ensemble des noeuds de **l'arbre syntaxique**, qui sont en cours de développement. Cette pile remplace la pile implicite des appels récursifs d'un analyseur avec rebroussement.



Exemple :

Soit la grammaire des expressions arithmétiques suivante :

```
1.   E   →  T  E'
2.   E'  →  +  T  E'
3.   E'  →  -  T  E'
4.   E'  →  ε
5.   T   →  F  T'
6.   T'  →  *  F  T'
7.   T'  →  /  F  T'
8.   T'  →  ε
9.   F   →  id
10.  F   →  nb
11.  F   →  (  E  )
```

On note VT : le Vocabulaire Terminal. Pour la grammaire précédente :

$VT = \{ +\ ,\ -\ ,\ *\ ,\ /\ ,\ id\ ,\ nb\ ,\ (\ ,\ ) \}$

On note VN : le Vocabulaire Non Terminal. Pour la grammaire précédente :

$VN = \{ E\ ,\ E'\ ,\ T\ ,\ T'\ ,\ F \}$

Le vocabulaire du langage noté V est = VT ∪ VN.

1. Calcul de First :

L'algorithme First peut être appliqué à une chaîne χ constituée des éléments du vocabulaire V du langage.

```
First (χ) {
     if ( χ ∈ VT ) {
          F = { χ } ;
     }
     else if ( χ ∈ VN ) {
     F = {} ;
     Pour toute production [ χ → α ] {
          F = F ∪ First(α) ;
     }
     }
     else if (χ == X1 X2 X3 ... Xn) {
     i = 1;
     while ( (i < n) && (ε ∈ First(Xi)) )
                F = F ∪ First(Xi) \ {ε};
                i++;
          }
          F = F ∪ First(Xi);
     }
     return F;
}
```

2. Calcul de Follow :

L'algorithme Follow ne peut être appliqué qu'à un élément X du vocabulaire non terminal VN.

```
Follow (X) {
     F = {} ;
     if ( X est = à l'axiome de la grammaire ) {
          F = { $ } ;
     }

     Pour toute production [ A → α X β ] {
     F = F ∪ First(β) \ {ε} ;
     if ( (β == ε) || (ε ∈ First(β)) ) {
          F = F ∪ Follow(A) ;
     }
     }
}
```

Exemple :

	First	Follow
E	id , nb , (	$,)
E'	+ , - , ε	$,)
T	id , nb , (	+ , - , $,)
T'	* , / , ε	+ , - , $,)
F	id , nb , (	* , / , + , - , $,)

3. Calcul de la table d'analyse LL(1) :

L'algorithme consiste à répartir les différentes productions sur les cellules de la table d'analyse que nous notons TA.

```
RemplirTableD'Analyse() {
     Pour toute production [ A → α ]) {
     Pour tout élément a ∈ (First(α) \ {ε}) {
          TA[A][a] = [ A → α ];
     }
     if (α == ε || ε ∈ First(α)) {
          Pour tout élément b ∈ (Follow(A)) {
               TA[A][b] = [ A → α ];
          }
     }
     }
}
```

Exemple :

TA	+	-	*	/	id	nb	(	)	$
E					1	1	1		
E'	2	3						4	4
T					5	5	5		
T'	8	8	6	7				8	8
F					9	10	11		

4. Analyseur LL(1) :

L'analyseur LL(1) utilise comme entrée :

- La table d'analyse que nous allons noter : **TA**
- La pile d'analyse que nous allons noter : **PA**
- Un objet d'analyse lexicale noté : **scanner** la fonction extract() sera utiliser pour récupérer une unité lexicale noté **p (Symbole de Prévision)**.

```
boolean LL1(Scanner scanner, TA) {
        PA = | S |
             | $ | ; // où S est l'axiome de la grammaire
        Token p = scanner.extract() ;
        Symbol A = PA.tete() ;
        do {
                if (A ∈ VT) {
                        if (p.equals(A)) {
                                if (p.equals("$")) return true;
                                else   {
                                        PA.depiler() ;
                                        p = scanner.extract() ;
                                }
                        }
                        else {
                                if (p.equals("$"))  error("Unexpected EOF ");
                                else if (A.equals("$"))  error("Expected EOF ");
                                else error("Expected Symbol : " + A);

                                return false;
                        }
                }
                else { // A ∈  VN
                        if ( TA[A][p] == [ A → X1 X2 X3 ... Xn ] ) {
                                PA.depiler() ;
                                for (i = N ; i >= 1 ; i--) PA.empiler(Xi) ;
                                A = PA.tete();
                        }
                        else {// TA[A][p] == null
                                if (p.equals("$"))  error("Unexpected EOF ");
                                else error("Unexpected Symbol " + p);
                                return false;
                        }
                }
        }
        while (true) ;
}
```

3.6 Analyseur LR(1)

3.6.1 principe

- Le problème est transformé à la recherche suivant les états d'un automate fini non déterministe.
- Un état de l'automate correspond à une position dans la partie droite d'une production de la grammaire noté : A $\rightarrow \alpha \, . \, \beta$, et appelé *Item*. Le point indique la position de l'analyseur.
- Les décisions sont prises en consultant la table d'analyse composée de deux partie : Action et successeur .

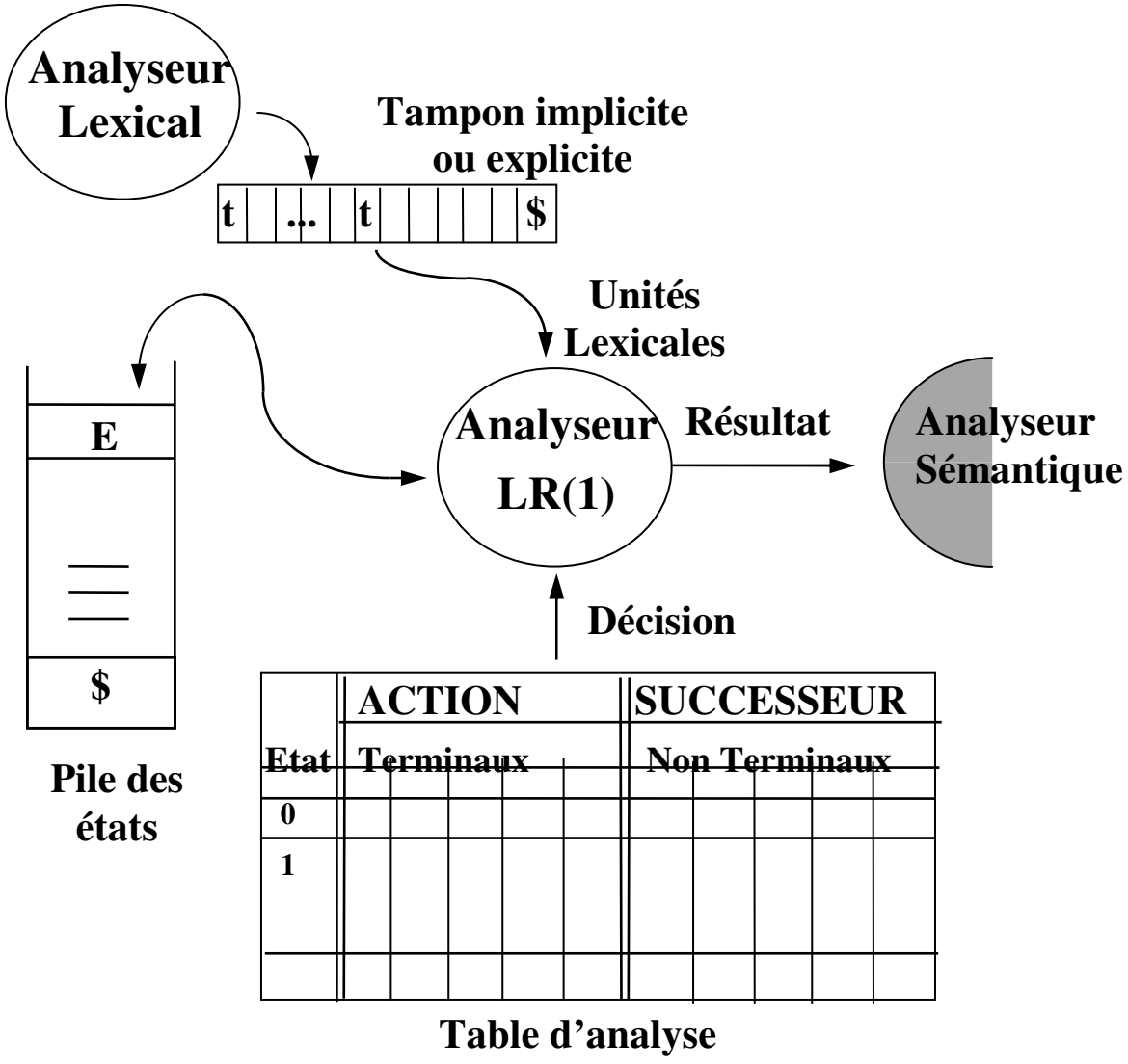


Table d'analyse

L'analyseur LR(1) utilise un tampon d'entrée, une pile d'états et une table d'analyse divisée en deux parties : Action et Successeur. Chaque état E en sommet de pile combiné avec le symbole d'entrée a possède une entrée dans la table de décision : Action[E][a], qui détermine l'action à effectuer :

- Décaler E' : avancer dans le tampon d'entrée et empiler un nouvel état E' en sommet de pile. Nous représentons cette décision par le couple {'D', E'}.

- Réduire $A \rightarrow \alpha$: qui signifie dépiler un nombre d'états égal au nombre de symboles qui composent α. Et empiler Successeur[F][A], avec F est le nouvel état en sommet de pile. Cette entrée sera représentée par le couple {'R', $A \rightarrow \alpha$}.

- Echec : l'analyseur détecte une erreur et peut appeler une procédure de récupération sur erreur. Pour garder la même notation nous représentons cette entrée par : {'E', 0}.

- Succès : signale la fin de l'analyse. Pour garder la même notation nous représentons cette entrée par : {'S', 0}.

3.6.2 Algorithme LR(1)

```
#define Succes 1
#define Echec 0

Initialement Pile ={$}
```

Algorithme LR(1)

```
{
  GetLex();
  Do {
        E = Tete(Pile);
        a = UL;
        if (Action[E][a] = {'S', 0 }) return Succes
        else if (Action[E][a] = {'E', 0}) return Echec
        else if (Action[E][a] = {'D', E'}) {
                GetLex();
                Empiler(E', Pile);
        }
        else if (Action[E][a] = {'R', A → α }) {
                L = |α| ; // cardinal de α : nombre de symboles
                For ( I=1; I<=L; I++) Pop(Pile);
                E=Tete(Pile);
                Empiler(Successeur[E][A], Pile);
        }
  }
  while (1);
}
```

3.6.3 Construction de la table d'analyse

trois méthodes existent pour la construction de la table d'analyse: Simple LR, LALR, et la meilleure : LR canonique.

3.6.3.1 Construction des tables d'analyse SLR

Soit G la grammaire utilisée d'axiome S.

On considère la grammaire augmentée G' de G, càd : la grammaire G avec une nouvel production : S' $\rightarrow$ S, et donc un nouvel axiome S'.

Algorithme SLR

On note VT le vocabulaire terminal, et VN le vocabulaire non terminal.

Algorithme SLR

{

1- Calculer la collection canonique : C = {I_0, I_1, …, I_n} de la grammaire G';

2- Pour chaque $I_i \in$ C associer un état i;

3- for (i = 1; i $\leq$ n ; i++) {

for ($\forall a \in$ VT) {

if (*Transition*(I_i, a) = I_j) Action[i][a] = {'D', j};

}

for ($\forall$ A $\rightarrow \alpha. \in I_i$, tq A $\neq$ S') {

for ($\forall$ a $\in$ Follow(A)) Action[i][a] = {'R', A $\rightarrow \alpha$};

}

if (S' $\rightarrow$ S. $\in I_i$) Action [i][$] = {'S', 0};

for ($\forall$X $\in$ VN) {

if (*Transition*(I_i, X) = I_j) Successeur[i][X] = j ;

}

}

}

Calcul de la collection canonique

On considère le vocabulaire V = VT ∪ VN.

Algorithme Calcul_Collection_Canonique

```
{
        I0 = Fermeture( { S' → .S});
        C = { I0}
        Do {
                Soit ( I ∈ C ) {
                        for (∀X ∈ V) {
                                if (Transition(I, X) ∉ C)  C = C + Transition(I, X) ;
                        }
                }
        }
        while (! Saturation);
}
```

Calcul de la Fermeture d'un ensemble d'Items

Soit un ensemble d'Items I.

Algorithme Fermeture (I)

```
{
        F = I ;
        Do {
                for (∀ A → α . B γ ∈ F) {
                        for (∀ B → φ , tq B → .φ ∉ F)  F = F ∪ {B → .φ};
                }
        }
        while (! Saturation);
        return F;
}
```

Opération de transition

Soit un ensemble d'Items I et soit X ∈ V.

Transition(I, X) = Fermeture ($A \rightarrow \alpha B . \gamma$ *tq* $A \rightarrow \alpha . B \gamma \in I$ }

Algorithme Transition (I, X)

```
{
    E = {};
    for (∀ A → α . B γ ∈ I ) {
        E = E ∪ { A → α B . γ};
    }
    T = Fermeture( E );
    Return T;
}
```

3.6.3.2 Construction des tables d'analyse LR canonique

ici un item sera noté [$A \rightarrow \alpha . \beta$, a] . où le a désigne le symbole terminal qui vient juste après la partie gauche de la production $A \rightarrow \alpha \beta$. $a \in VT \cup \{\$\}$. Il faut alors redéfinir les algorithmes Fermeture, Transition, construction de C et construction des tables d'analyse.

Fermeture d'un ensemble d'Items

Algorithme Fermeture (I)

```
{
    F = I ;
    Do {
        for (∀ [A → α . B γ , a] ∈ F) {
            for (∀ B → φ )
                for (∀ b ∈ First(γ a) ) {
                    if ( [B → .φ, b] ∉ F )   F = F ∪ { [B → .φ, b] };
                }
        }
    }
    while (! Saturation);
    return F;
}
```

Opération Transition

Transition(I, X) = Fermeture ([A → α B **.** γ, a] *tq* [A → α **.** B γ, a] ∈ I }

Algorithme Transition (I, X) {

```
        E = {};
        for (∀ [A → α . B γ, a] ∈ I ) {
                E = E ∪ { [A → α B . γ, a] };
        }
        T = Fermeture( E );
        Return T;
}
```

Construction de la collection canonique

Algorithme Calcul_Collection_Canonique {

```
        I0 = Fermeture( { [S' → .S, $]});
        C = { I0}
        Do {
                Soit ( I ∈ C ) {
                        for (∀X ∈ V) {
                                if (Transition(I, X) ∉ C)  C = C + Transition(I, X) ;
                        }
                }
        }
        while (! Saturation);
}
```

Construction des tables d'analyse

Algorithme LR_Canonique {

```
        1- Calculer la collection canonique : C = {I0, I1, …, In} de la grammaire G';
        2- Pour chaque Ii ∈ C associer un état i;
        3- for (i = 1; i ≤ n ; i++) {
                for (∀a ∈ VT) {
                        if (Transition(Ii, a) = Ij)  Action[i][a] = {'D', j};
                }
                for (∀ [A → α. , a] ∈ Ii) {
                        Action[i][a] = {'R', A → α};
                }
                if ([S' → S. , $] ∈ Ii ) Action [i][$] = {'S', 0};

                for (∀X ∈ VN) {
                        if (Transition(Ii, X) = Ij)  Successeur[i][X] = j ;
                }
        }
}
```

Chapitre 4

Analyse sémantique

4.1 Objectif

L'analyse sémantique consiste à contrôler la vérification des règles concernant le sens logique des expressions syntaxiquement justes.

⇨ Contrôle de type.
⇨ Contrôle d'unicité.
⇨ Contrôle du flot d'exécution (instruction englobante).

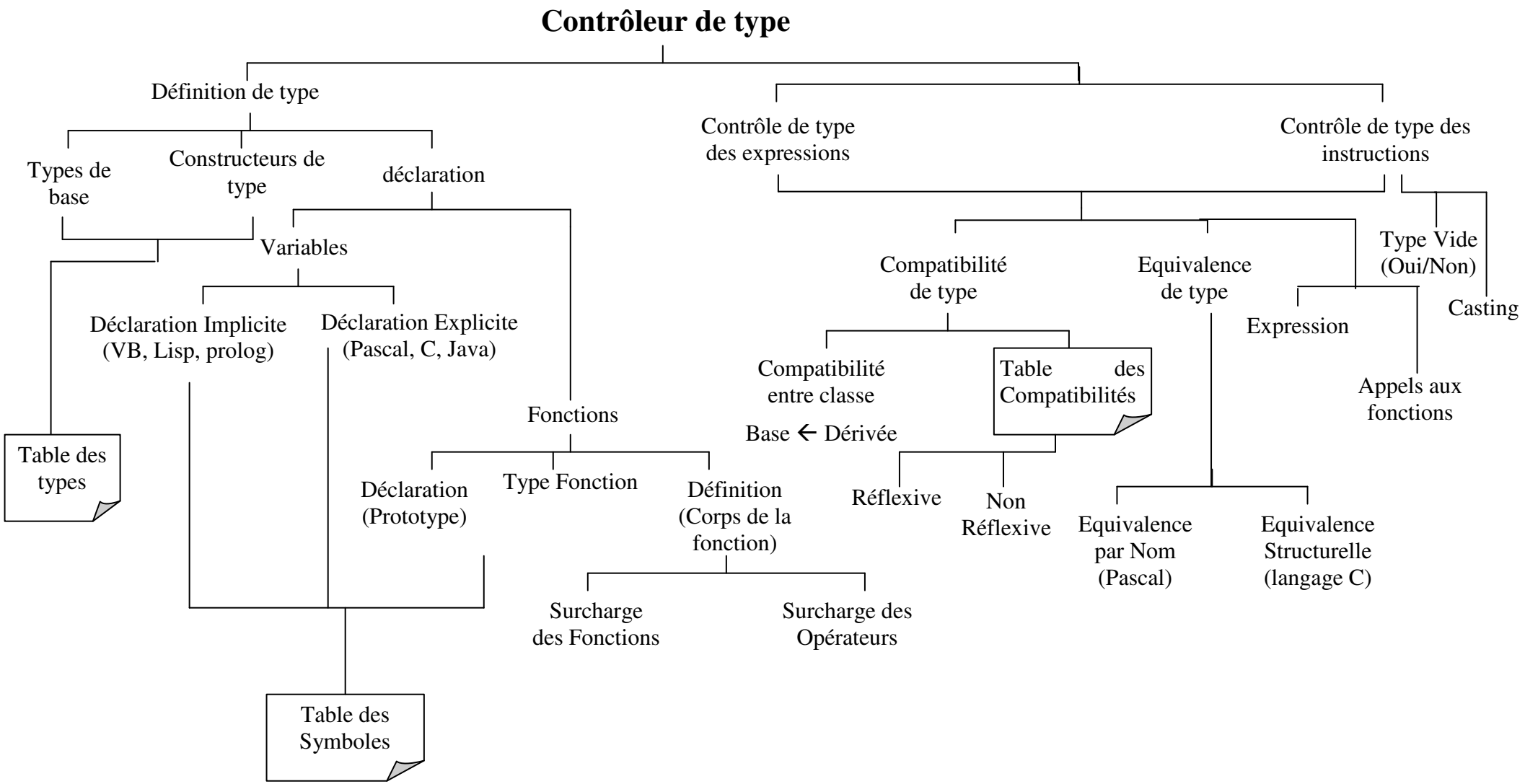
4.2 Traduction dirigée par la syntaxe

- L'analyse sémantique est effectuée à l'aide d'une traduction dirigé par la syntaxe
- La grammaire est transformée en une **grammaire de traduction avec attributs**.
  ⇨ Des **actions sémantiques** sont associées aux productions de la grammaire.
  ⇨ Des **attributs** sont associés aux symboles de la grammaire.
  ⇨ Après la vérification d'une production on exécute l'action sémantique correspondante.

  $N_1 \quad \rightarrow \alpha_1 \ \{ AS1 \}$
  $N_2 \quad \rightarrow \alpha_2 \ \{ AS2 \}$

4.3 Contrôle de type

- Définition d'un système de typage (déclaration)
- Contrôle de type { des expressions. / des instructions.
- Equivalence de type { Equivalence structurelle (C). / Equivalence par nom (Pascal).


Contrôleur de type
Définition de type
Contrôle de type des expressions
Contrôle de type des instructions
Types de base
Constructeurs de type
déclaration
Variables
Déclaration Implicite (VB, Lisp, prolog)
Déclaration Explicite (Pascal, C, Java)
Table des types
Fonctions
Déclaration (Prototype)
Type Fonction
Définition (Corps de la fonction)
Surcharge des Fonctions
Surcharge des Opérateurs
Table des Symboles
Compatibilité de type
Equivalence de type
Type Vide (Oui/Non)
Expression
Casting
Appels aux fonctions
Compatibilité entre classe
Base ← Dérivée
Table des Compatibilités
Réflexive
Non Réflexive
Equivalence par Nom (Pascal)
Equivalence Structurelle (langage C)

Chapitre 5

Génération du code intermédiaire

5.1 Objectif

- Produire à partir du code source un code intermédiaire indépendant de toute exigence matérielle.
- La phase de gestion du code finale sera indépendante des particularité du code source.
- Facilité de changement du code final sans avoir à toucher la phase d'analyse

5.2 Réalisation

⇨ Choix du code intermédiaire.

⇨ Manière d'implantation.

5.3 Code à 3 adresses

- suite d'instructions faisant intervenir au maximum 3 adresses:

  Résultat := **Opérande**$_1$ Opération **Opérande**$_2$

- Généralisation de cette forme pour englober tout type de construction du langage.

Structure de donné utilisée : enregistrement

⇨ Notion des **quadruplets (~ unité syntaxique ~)**.

5.4 Implémentation

- Sous forme d'une **traduction dirigée par la syntaxe** au sein de l'analyseur sémantique.
- création d'un **temporaire** pour chaque opérateur.

Exemple:

x := a + b * c ⇔ t1 = b * c , t2 = a + t1 , x = t2

6. Optimisation de code

- Phase optionnelle.
- Optimisation de point de vue temps d'exécution
- Utilisation optimale des registres.
- Traduction des multiplications, si possible, en additions ou décalages.
- Suppression des calculs invariants dans les boucles
- nombre d'instructions.

⇨ Nécessite de parcourir plusieurs fois le fichier d'entrée, en modifiant à chaque fois l'ordre d'évaluation des quadruplets.

⇨ Phase qui augmente le temps de compilation.

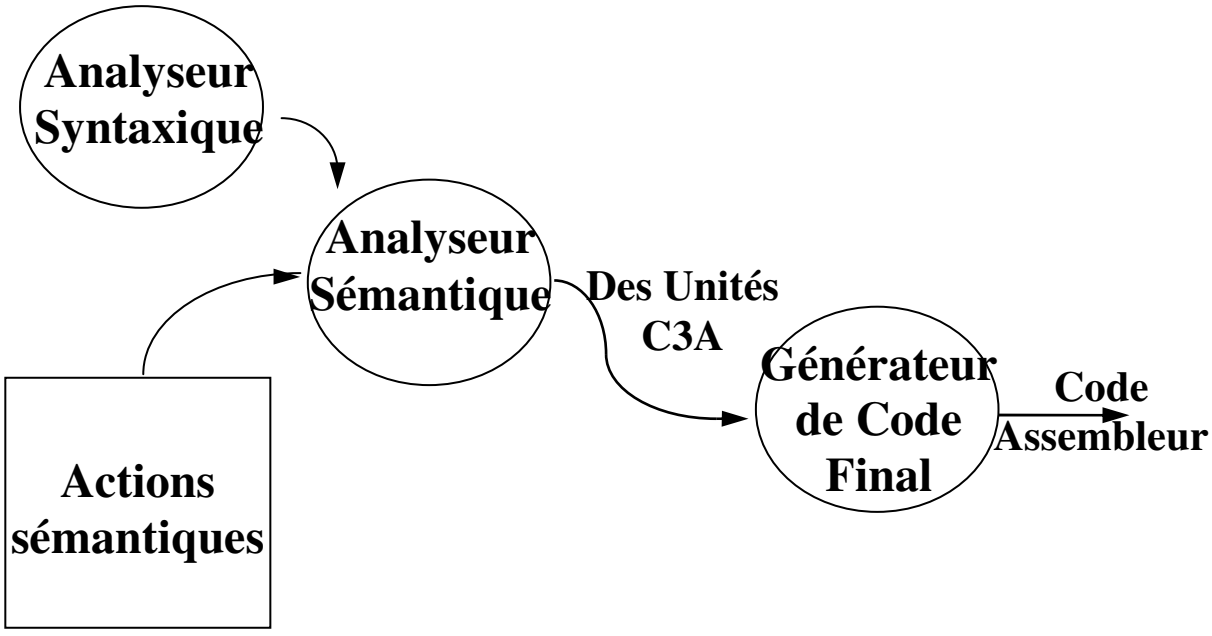
7. Génération de code final

7.1 Choix du code final

- **Code machine (.obj) :**
  - ⇨ Plus compliqué.
  - ⇨ Gestion des adresses.
  - ⇨ Connaissance du matériel.
  - ⇨ Dépendance du matériel.

- **Code assembleur**
  - ⇨ Plus facile à générer.
  - ⇨ Plus proche du code à 3 adresses.
  - ⇨ Existence de l'Assembleur.

7.2 Choix de la technique d'implémentation



Conclusion

- Une très bonne organisation des compilateurs.

- Plusieurs techniques permettant l'implantation efficace d'un très grand nombre de compilateurs, et encourageant la conception est le développement de plusieurs langages évolués.

- Un ensemble d'outils permettant l'implantation facile et immédiat des compilateurs.

- Des techniques pouvant être appliquées à différents domaines.

- Tout produit informatique, faisant intervenir des notions de traduction peut bénéficier des techniques de compilation et être configuré suivant le schéma d'un compilateur.

Exercices et Travaux Pratiques

Exercices d'analyse :

1- Donner une comparaison entre les différentes méthodes d'analyse syntaxique.

2- Quelle sont les fonctions des deux utilitaires LEX et YACC, leur principe et sur quelle notion se base chacun des deux utilitaires ?

3- Qu'est ce qu'on veut dire par les deux notions : prédictif et sans rebroussement, et est ce qu'il y a une différence entre ces deux notions ?

4- Est-ce qu'on peut écrire un générateur de code intermédiaire séparé de l'analyseur syntaxique ? Faites une comparaison avec le cas d'intégration dans celui-ci.

5- Qu'est ce qu'une grammaire LR(1) ? Décrire l'analyseur LR(1).

6- Est-ce qu'une grammaire LL(1) et forcément LR(1) ?

7- Est-ce qu'on peut envisager l'optimisation de code après la phase de génération de code final ?

8- Dans quelle phase a-t-on réellement besoin de la table de symbole ? et quelle peut être la structure ?

9- Qu'est ce que le préprocesseur ? L'éditeur de lien ? et quelle différence existe t-elle entre un programme .obj et son .exe correspondant ?

10- Est-ce qu'on peut avoir une ambiguïté pour l'extraction des unités lexicales lors de l'analyse lexicale. Expliquer.

11- Etant donné un langage de programmation. Quelles sont les étapes qu'on doit entreprendre avant de commencer la réalisation d'un compilateur ?

12- Y a t-il une technique pour le traitement des erreurs syntaxiques ? sémantique ?

13- En quoi consiste l'analyse ascendante ? Qu'est-ce qu'un manche ? Qu'est-ce qu'une réduction ? C'est quoi 'Elagage du manche' ?

14- Quels sont les inconvénients de l'analyse par précédence d'opérateur ? Qu'est-ce qu'une grammaire d'opérateur ?

15- Qu'est-ce que le conflit Décaler/Réduire ? Réduire/Réduire ?

16- Quels sont les conflits qu'on peut avoir lors de l'analyse syntaxique ascendante ? est-ce qu'on peut avoir les mêmes problèmes en analyse descendante ? Quels sont les problèmes équivalents ?

17- Donner les inconvénients des différents analyseurs que vous connaissez.

Exercice :

Donner les automates à états finis déterministes engendrés par les expressions régulières suivantes :

1- (a | b)* a b b

2- 0 (0 | 1)* 0

3- ((ε | 0) 1*)* 0 0

4- (0 0 | 1 1)* 0 1 1

5- 0* 1 0* 1 0* 1 0*

6- ((ε | a) b*)*

7- (a | b)* a (a (a |b) (a | b)

8- (a | b)* a b (a | b)*

9- (a | b)* a b b (a | b)* a b b

10- (a | b)* a b a b

11- $(0\ 1^+ \mid 0\ 1^+ (0 \mid 1)^+ \mid (0 \mid 1)^+ 0\ 1$

12- $0^+ \mid 0^+ 1\ 0^+ \mid 0^+ (0 \mid (0\ 1)\ )^+$

TP N° 1 :

On demande la réalisation d'un analyseur Lexical Turbo Pascal. L'analyseur doit savoir extraire toutes les unités lexicales d'un programme Pascal. Celles-ci seront rangées dans une liste chaînée dont les éléments sont caractérisés par : le lexème, l'unité lexicale, la ligne sur laquelle se trouve le lexème. A la fin du programme, le contenu de la liste est enregistré dans un fichier texte organisé en 3 colonnes séparé par des espaces. L'analyseur doit signaler toute présence d'erreur tout en affichant la ligne sur laquelle l'erreur est rencontré. En fin l'analyseur affiche le nombre d'erreurs lexicales trouvées.

Les unités lexicales à identifier sont :

- Les identificateurs
- Les nombres
- Les commentaires (* …*) et {…} qui sont ignorés
- Les Blanc qui sont aussi ignorés
- Chaînes de caractères
- Opérateurs de relation
- Opérateurs arithmétiques
- L'opérateur d'affectation
- Les blancs (espace, tabulation et retour chariot) sont ignorés
- Les séparateurs : , ; ( ) [ ] …
- Les opérateurs logiques
- Les mots clés

On utilisera la fonction suivante :

```
int  AFD(Transition T, Etat A, Ulex NU)
{
       int e=0;
       char s = Tampon[ReadHead];
       int save = ReadHead;
       while (T[e][s]!=-1)
       {
                   e=T[e][s];
                   ReadHead++;
                   s=Tampon[ReadHead];
       }
       if (strchr(A,e) && e!=0) {
             free(Lexeme);
             Lexeme = (char *) malloc(ReadHead - save + 1);
             strncpy(Lexeme, &Tampon[save], ReadHead - save);
             Lexeme[ReadHead - save ] = 0;
             UL = NU;
             return 1;
       }
       else {
             ReadHead = save;
             return 0;
       }
}
```

avec :

```
typedef int Transition [MaxEtat][256];
typedef char Etats [MaxEtat];
enum Ulex {id, nb, oprel, ...};
```

TP N° 2 :

Ajouter à l'analyseur lexical les fonctionnalités suivantes :

1- Les mots clés sont lus à partir d'un fichier Texte.

2- Tous les identificateurs seront stockés dans une structure de liste chaînée.

3- Les autres lexèmes sont stockés dans une autre structure de liste chaînée à deux champs : (Lexème et Unité lexicale).

4- Toutes les erreurs sont stockées dans un fichier texte « lex.err », on précisant l'erreur et sa ligne d'occurrence.

5- Les différentes unités lexicales sont réimprimées à l'écran avec des couleurs différentes

TP N° 3 :

Soit un fichier texte contenant des séquences de chiffres et des mots clés brouillés avec des caractères quelconques.

Exemple :

dgdg_-$e**begin**gte**1254787547**h*heg

Ecrire un analyseur lexical qui fait la recherche et l'extraction de toutes les séquences de chiffres dont le nombre est fixe (10 par exemple) ainsi que tous les mots clés. On dispose d'un fichier texte de mots clés acceptés. Le programme génère un fichier de sortie sur lequel il reprend uniquement les mots clés et les nombres acceptés séparés par des espaces.

TP N° 4 :

Ecrire un programme qui lit un fichier texte (en Turbo C++) et transforme toute occurrence de '{' en 'Begin', '}' en 'end', 'printf' en 'write' et 'scanf' en 'read', dans un nouveau fichier.

TP N° 5 :

Ecrire un programme qui reçoit en entrée le nom d'un fichier texte (programme C) et fait la suppression de tous les commentaires du fichier et met le résultat dans un deuxième fichier.

TP N° 6 :

Ecrire une fonction qui reçoit en paramètre une chaîne de caractères et vérifie si elle appartient au langage engendré par la grammaire suivante :

$S \rightarrow 0\ S\ 1 \mid 01$

TP N° 7 : **Réalisation d'un Préprocesseur C**

1- Ecrire une procédure StrSubst(char *S1, char *S2, char *S) qui remplace toute occurrence de la chaîne S1 dans la chaîne S par la chaîne S2.

2- Ecrire une procédure FileSubst(char *S1, char *S2, char *Fichier) qui remplace toute occurrence de S1 dans Fichier par S2.

3- Ecrire une procédure qui reçoit en entrée un fichier texte (programme C) contenant des Macros (#define S1 S2) et en construit un nouveau fichier en remplaçant toute occurrence de S1 par S2.

4- Ecrire un programme qui reçoit en entrée un fichier texte contenant des instructions (#include "…") et en construit un nouveau fichier contenant le programme d'entrée concaténé avec les fichiers spécifiés par #include.

5- Déduire la réalisation d'un préprocesseur C.

6- Refaite la réalisation du préprocesseur en utilisant un analyseur lexical.

TP N° 8 :

Ecrire un programme qui fait l'indentation d'un programme C. c.à.d organise le programme à l'aide des tabulations de manière à obtenir un programme sous la forme :

```
Main()
{
→     …
→     …
→     If (…)
→     {
      →     …
      →     …
      }
→     …
}
```

le symbole → indique des tabulations.

TP N° 9 :

Le problème suivant se compose de deux parties :

1- La première partie consiste à choisir des codes pour les différentes lettres de l'alphabet. Le code (appelé code morse) sera constitué d'une séquence formée par les symboles **.** et **–** à condition que le code ne soit pas uniforme (on n'associe pas le même nombre de symboles à toutes les lette) et de ne pas avoir d'ambiguïté pour les reconnaissance des codes.

Exemple : `'A' = ".-", 'B' = "-..-", etc.`

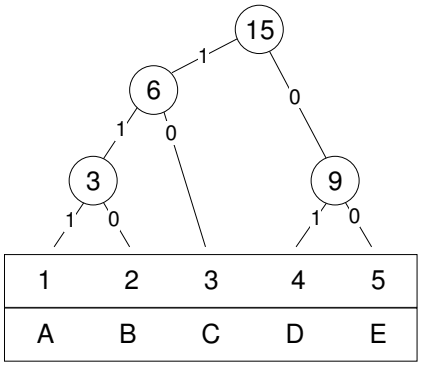
Pour déterminer les codes de manière non ambiguë, on propose alors un algorithme qui découle de la méthode de Huffman :

Cela consiste à réaliser les étapes suivantes :

- Etape 1 : Associer à chaque lettre de l'alphabet une valeur entre 1 et 26.
  Exemple : 'A' → 1, 'B' → 2, …
- Etape 2 : regrouper séquentiellement par paires de lettres de plus faibles valeurs.

Exemple :

On prend par exemple les lettres : A, B, C, D et E. et on leur associe successivement les valeur de 1 à 5.



- Etape 3 : on obtient un arbre binaire, on code alors chaque fils gauche par 1 et chaque fils droit par 0. Le code lue à partir de la racine jusqu'à une feuille donne le code binaire de la feuille. Dans l'exemple on obtient :

  A = 111, B = 110, C = 10, D = 01, E = 00

- Etape 4 : On remplace alors chaque 1 par − et chaque 0 par `.`, on obtient :

  `A = ---, B = --., C = -., D = .-, E = ..`

On demande alors de déterminer le code pour les 26 lettres de l'alphabet on appliquant la méthode précédente. Transformer l'arbre de huffman en un AFD à 26 états accepteurs.

2- La deuxième partie consiste à implémenter un analyseur lexical basé sur l'AFD obtenu en 1. L'analyseur aura pour objectif de lire un fichier texte codé et de le décoder dans un deuxième fichier. la fonction AFD devra retourner à chaque appel l'état accepteur sur lequel on a aboutit. Cet état sera utilisé pour déterminer la lettre correspondante.

TP N° 10 :

L'objectif de ce TP est l'implémentation d'un analyseur lexical.

Le travail sera alors divisé en deux parties :

Partie N° 1 :

1. Classe Scanner

Il s'agit d'une classe générale qui regroupe les fonctionnalités, générales et indépendantes du langage, d'un analyseur lexical. La classe Lex peut être définie à l'aide de la structure suivante :

```
Class Scanner {
      Unsigned char Tampon[MaxLen];
      Unsigned char *Lexeme;
      int UL;
      int  PtrDeb,PtrFin,Ltamp;
      Fonction *TU ;
      FILE *Source;
      Public:
            Lex();
            Lex(char *NomFichier);
            Int  Refresh(); //Maj Tampon
            int  AFD(Transition T, Etats A, int idUL)
            int  Open(char *NomFichier);
            void Load(Fonction u1,...);
            int  GetNext(char *Lexeme, int &UL);
            void ErreurLexicale();
            int  NoMore();  // fin de fichier : UL = '$'
            ~Lex();
};
```

La classe Scanner utilise 3 types : Transition, Etats et Fonction qui sont définis comme suit :

```
#define MaxEtat 25
typedef int Transition[MaxEtat][256];
typedef unsigned char Etats[MaxEtat];
typedef int (*Fonction)();
```

2. Principales méthodes de la classe

2.1 Méthode AFD

Il s'agit de la méthode de base qui permet l'analyse du tampon d'entrée en fonction d'un automate fini déterministe associé à l'unité lexicale à analyser

2.2 Méthode Refresh

Cette méthode est appelée avant chaque tentative d'extraction d'unité lexicale pour mettre à jour le tampon d'entrée.

2.3 Méthode Open

C'est la méthode qui ouvre le fichier à analyser. Elle doit être appelé avant toutes les autre fonctions.

2.4 Méthode Load

Cette méthode permet de remplir le tableau des fonction d'extractions d'unités lexicales **TU**. Elle reçoit donc en paramètre une liste de noms de fonctions qui se termine par NULL. Par exemple : identificateur, nombre, …

Les fonctions ainsi obtenu seront utilisées dans la fonction suivante.

2.5 Méthode GetNext

Il s'agit de la fonction d'analyse lexicale proprement dit. Elle effectue une série d'appels au différentes fonctions d'analyse lexicale (chargées dans TU) pour extraire la prochaine unité lexicale. Celle-ci sera alors retournée en paramètre. Dans le cas d'erreur la méthode **ErreurLexicale** sera alors appelée.

2.6 Méthode NoMore

C'est une fonction qui teste s'il n'y a plus rien à analyser.

Partie N° 2 :

Afin de tester les fonctionnalités de la classe Lex, on se propose de construire une nouvelle classe qui se dérive de la classe Lex et qui la complète. On remarque en effet que la classe Lex ne dispose d'aucune des fonctions lexicales (identificateur, nombre, etc..). Pour cela on réalise une classe Pascal qui définie les principales unités lexicales du langage Pascal.

La classe Pascal se présente de la manière suivante :

```
class Pascal:public Lex {
      public:
            Pascal() {Load(Id,Nb,Oprel, ..., NULL);}
            static int Id()
            {
           Etat A = {1} ;
           Transition T ;
           ...
           return  AFD(T,A,id) ;
            }
            static int Nb()
            {
           ...
           return  AFD(T,A,nb) ;
            }
            static int Oprel()
            {
           ...
           return  AFD(T,A,oprel) ;
            }
      ...
};
```

Les constantes id, nb, oprel sont les unités lexicale associées au différentes fonctions et peuvent être définies comme des constantes dans un type énuméré :

enum {id, nb, oprel, …} ;

Comme exemple d'application, on peut écrire le programme suivant :

```
void main()
{
      char *Lexeme ;
      int UL ;
      Pascal P ;
      P.Open("prog01.pas") ;
      while (!P.NoMore()) {
                  if (P.GetNext(Lexeme, UL) ==1) {
                        Printf("%s : %d\n",Lexeme,UL) ;
                  }
      }
}
```

TP N° 11 :

L'objectif de ce TP est l'implémentation de l'algorithme LL(1) pour la réalisation d'un analyseur syntaxique descendant prédictif.

Le travail sera alors divisé en deux parties :

Partie N° 1 :

1. Classe LL1

Dans une première partie du travail, on se propose d'implémenter une classe **LL1** définie par l'intermédiaire de trois propriétés de base qui sont la grammaire, la table d'analyse ainsi qu'un analyseur lexical. Ces 3 données vont constituer l'entrée de l'analyseur (fournies donc via le constructeur). Une méthode **Parse** pouvant être appelée par l'intermédiaire d'une instance de la classe LL1 pour analyser le fichier fourni en paramètre. Elle retourne 0 ou 1 en fonction du résultat de l'analyse. La méthode Parse constitue une implémentation de l'algorithme LL(1). La classe LL1 aura alors la structure suivante :

```
Class LL1 {
      Scanner *L;
      Grammaire *G;
      TabledAanalyse *TA;
      Public:
            LL1(Lex &L, Grammaire &G, TabledAanalyse
&TA);
            int Parse(char *Fichier);
            ~LL1();
};
```

On remarque alors que la classe LL1 a besoin de 3 autres classes : Lex, Grammaire et TabledAnalyse.

2. Classe Scanner

La classe **Scanner** étant la classe réalisée dans le TP N° 1.

3. Classe Grammaire

La classe **Grammaire** est une classe qui permet la construction des grammaires. Elle est définie alors par un Attribut principal : Liste de produits. Elle contient aussi d'autres attributs tels que : l'axiome, VT, VN, nombre de production ainsi que la désignation littérale de chaque symbole du vocabulaire. La classe Grammaire fournie aussi une méthode principale : **Production** qui permet, à chaque appel, d'ajouter une nouvelle production à la grammaire. L'utilisateur de la classe pourra définir un type énuméré pour le vocabulaire et appeler la méthode par 'intermédiaire des nom de constante qu'il a définie.

Exemple :

Soit la grammaire :

E → T E1

E1 → + T E1 | - T E1 | ε

…

l'utilisateur définira alors :

```
enum {E, E1, T, …};
```

Pour construire la grammaire, il réalise les opérations suivantes :

```
Grammaire G;
G.production(E, T, E1, -1);
G.production(E1, '+', T, E1, -1);
G.production(E1, '-', T, E1, -1);
G.production(E1, -1);
```

La classe grammaire peut finalement avoir la structure suivante :

```
class Grammaire {
      int Axiome, NbProd;
      ListProd Prod;
      int *VT, *VN;
      char *Desig[];
      public :
            Grammaire();
            int production(unsigned int pg,...);
              //retourne le numéro de production
            void setAxiome(int S);
            void setDesig(int S, char *D);
            void afficher();
            ~Grammaire();
};
```

La structure ListProd (Liste des productions) est une structure de liste chaînée définie par l'intermédiaire des structure suivantes :

```
Typedef struct PtrListe {
      int symbole;
      PtrListe *suc;
} *Chaine;
```

```
typedef struct {
      int pg;
      Chaine pd;
} Production;
```

```
typedef struct Ptr {
      int Num;
      Production p;
      Ptr *suc;
} *ListProd;
```

4. Classe TabledAnalyse

La classe **TabledAnalyse** est utilisée pour introduire la table d'analyse qui sera utilisée (ainsi que la grammaire) par l'analyseur LL1. Celle-ci contient donc une donnée T qui va recevoir le détail de la table d'analyse. Ainsi une nouvelle entrée dans la table d'analyse sera ajoutée par l'intermédiaire de la méthode **ajouter** qui reçoit 3 paramètres : le non terminal, le terminal et le numéro de la production. Le remplissage de la table d'analyse pourra être fait parallèlement à celui de la grammaire.

Exemple :

Pour introduire :

```
TA[E]['('] = E → T E1
TA[E][id]  = E → T E1
```

On peut écrire :

```
Grammaire G;
TabledAnalayse TA;
int N = G.production(E, T, E1, -1);
TA.ajouter(E, '(',N);
TA.ajouter(E, id, N);
```

Finalement la classe TabledAnalyse pourra avoir la structure minimale suivante :

```
class TabledAanalyse {
      int *T;
      public:
            TabledAanalyse();
            void ajouter(int NT,int T,int Num);
            void afficher();
            ~TabledAanalyse();
};
```

5. Utilisation de la classe LL1

Etant donnée maintenant que l'on dispose des 4 classes précédantes l'écriture d'un programme d'analyse pourra être réalisée comme suit :

```
Main() {
      Lex L ;
      Grammaire G ;
      TabledAnalyse TA ;
      Initialiser (G, TA) ;

      LL1 A(L, G, TA);
      char NF[80] ;
      printf("Entrer le nom du fichier à analyser : ") ;
      gets(NF) ;
      if (A.Parse(NF)) printf("aucune erreur") ;
      else printf("Erreurs :") ;
}
```

Remarques :

Pour accéder aux données des autres classes à partir de l'analyseur LL1, il pourrait être nécessaire d'ajouter d'autres méthodes aux différentes classes.

6. Test de l'analyseur LL1

Pour tester l'analyseur LL1 obtenu, on demande l'implémentation d'un analyseur d'expressions arithmétiques.

Partie N° 2 :

Afin de minimiser les efforts de frappe et la nécessité se recompilation des modules on propose d'ajouter à chacune des deux classes : Grammaire et TabledAnalyse un constructeur permettant de charger les données à partir de fichiers.

On ajoute alors à la classe Grammaire le constructeur suivant :

```
Grammaire(char *NF);
```

Celui-ci va permettre de charger une grammaire quelconque à partir d'un fichier texte donné par son nom. La structure du fichier pourra être la suivante :

```
4
E    →  T  E1
E1   →  +  T  E1
E1   →  -   T  E1
E1   →
```

La première ligne contient le nombre de productions, les lignes suivantes la liste des productions.

La classe TabledAnalyse aura aussi un nouveau constructeur :

```
TabledAanalyse(char *NF);
```

Ce qui offre la possibilité de charger une table d'analyse à partir d'un fichier texte. On utilisera alors le même fichier que celui des productions ; seulement les données de la table d'analyse viendront juste après la dernière production.

Exemple :

Pour écrire la grammaire précédente et représenter par exemple :

```
TA[E]['('] = E → T E1
TA[E][id]  = E → T E1
TA[E1]['+']  = E1 → + T E1
TA[E1]['-']  = E1 → - T E1
```

Le fichier aura la structure suivante (la ligne qui sépare la grammaire de la table d'analyse ne doit pas figurer dans le fichier, elle est ajoutée ici juste l'explication) :

```
4
E  →  T  E1
E1 →  +  T  E1
E1 →  -   T  E1
E1 →
-------------------
1  2
E  '('
E  id
2  1
E1 '+'
3  1
E1 '-'
```

La deuxième partie du fichier concernant la table d'analyse se lit :

La production n° 1 se trouve dans 2 cases :

```
TA[E]['('] et
TA[E][id]
```

La production n° 2 se trouve dans 1 case :

```
TA[E1]['+']
```

La production n° 3 se trouve dans 1 case :

```
TA[E1]['-']
```

Réaliser les deux constructeurs et écrire un nouveau programme de test avec la grammaire des expressions arithmétiques.