USTHB

Faculté d'Informatique

Licence L3 Académique

Compilation : Les mécanismes de base

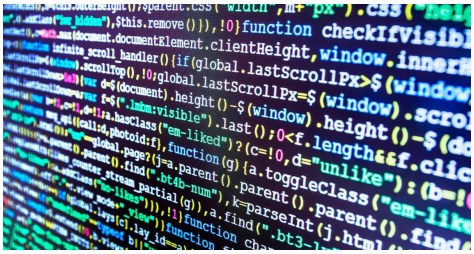
MALIKA BOUKALA-IOUALALEN HADJA FAIZA KHELLAF-HANED ABDENOUR HIMRANE

Année Universitaire 2021-2022

USTHB
Université des Sciences et de la Technologie
Houari Boumediene

FACULTÉ D'INFORMATIQUE

Compilation : Mécanismes de base



Malika Ioualalen-Boukala
Hadja Faiza Khellaf-Haned
Abdenour Himrane

Si debugger, c'est supprimer des bugs,
alors programmer ne peut être que les ajouter
- Edsger Dijkstra (1930-2002)

Table des matières

Introduction générale

1 Introduction

Au début de l'ère de l'informatique, les premiers programmes étaient écrits en langage machine ou langage binaire composé d'une suite de chiffres de 0 et de 1. Cette méthode n'était pas pratique, vu que la mise au point et l'écriture des programmes étaient assez complexes et fastidieuse. Les programmes étaient ensuite écrits en langage assembleur, ce qui consistait en l'utilisation de codes mnémoniques. Le code assembleur était proche de la machine, mais il est impératif de connaître la structure interne de celle-ci (les modes d'adressage, les registres,...) pour pouvoir écrire un programme. Les inconvénients que présentaient le langage machine et le langage assembleur ont conduit au développement de langages, appelés langages évolués, tels que le FORTRAN, ALGOL, qui sont proches du langage naturel, et qui sont aussi indépendants de la structure interne de la machine. Les différents niveaux des langages de programmations peuvent être illustrés par la Figure 1.

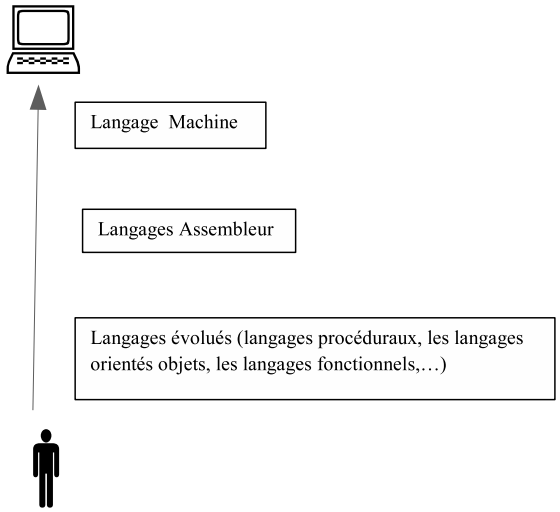


FIGURE 1 – les différents niveaux de langage de programmation

Le problème qui se pose alors est le suivant :

— un langage évolué est-il directement compris par la machine ?

— comment implémenter un langage évolué sur machine ?

L'idée consiste à transformer un programme écrit en langage évolué en un programme ou code directement exécutable par la machine. Le processus de transformation est appelé compilation.

Définition 0.1 *La compilation est un processus de traduction d'un programme source écrit dans un langage évolué vers un programme cible formé par des instructions machine.*

Ainsi, à partir d'un programme source écrit dans un langage évolué, le compilateur produit donc un programme cible qui sera exécutable par une machine et qui qui va fournir des résultats à l'utilisateur (voir Figure 2).

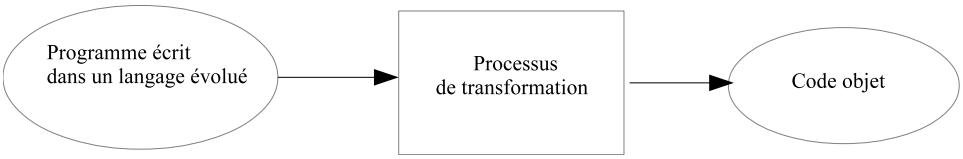


FIGURE 2 – le processus de compilation

Il existe d'autres transformations de programmes sources telles que la traduction et l'interprétation.

— La traduction consiste à traduire un programme écrit dans un langage L_1, appelé programme source, en un programme écrit en langage L_2, appelé code objet.

— L'interprétation consiste à traduire et à exécuter un programme écrit en langage source, instruction par instruction,

— La semi-interprétation consiste non pas à traduire complètement le code source à chaque exécution mais le code source est préalablement traduit dans un langage intermédiaire proche du langage machine.

Remarque 0.1 *Si le langage L_2 est le langage machine ou l'assembleur, alors le processus de transformation est celui du compilateur. Par contre, si les langages L_1 et L_2 sont tous les deux des langages évolués, le processus de transformation correspond au préprocesseur ou au pré-compilateur.*

Le compilateur est plus rapide, mais il est plus difficile à écrire. L'interpréteur traite les instructions au vol. Il ne produit pas de code objet.

Remarque 0.2 *Il existe aussi des compilateurs mono-passe et des compilateurs multi-passes. Dans le premier cas, la compilation s'effectue en un seul passage. Le compilateur transmet le code source à chaque entité de la compilation une seule fois. Dans le second cas, la compilation s'effectue en plusieurs passes successives.*

Il existe différents types de langages évolués :

1. les langages procéduraux dont le principe consiste à décomposer une tâche de programmation en un ensemble de variables, de structures de données et de sous-programmes. Les procédure opèrent directement sur les structures de données. Il existe une grande panoplie de langage procéduraux tels que le langage Pascal, le langage Fortran, le langage C,...

2. les langages orientés objets où il s'agit de décomposer une tâche de programmation en objets qui exposent le comportement ou les méthodes et les données ou attributs à l'aide d'interfaces. La programmation orientée objet regroupe les procédures et les structures de données de telle de sorte qu'un objet (une instance d'une classe) , fonctionne sur sa structure de données. Le langage C++, le langage Java, le langage Python figurent parmi les langages orientés objets les plus utilisés,
3. les langages fonctionnels dont le principe général consiste à concevoir des programmes comme des fonctions mathématiques. Ils se basent sur des expressions dont la valeur est le résultat du programme. Les langages fonctionnels les plus connus sont le Lisp, ML et Ocaml.
4. les langages Logiques, dont le paradigme repose sur la logique mathématique, considèrent un programme non pas comme une suite d'instructions mais plutôt comme une liste de faits et hypothèses. Chaque requête adressée au programme est traitée par un interpréteur qui applique des règles d'inférence afin d'atteindre le résultat. Ce type de langage est utilisé dans les applications dédiées à l'intelligence artificielle. Le langage PROLOG est la référence des langage logiques.
5. Les autres formes de langages : la notion de langage ne se limite pas uniquement aux langages de programmation. En effet, le processus de compilation peut être étendue à d'autres types de langage :
   - les langages de marquage de texte tel que HTML,
   - les langages de description de texte tel que LATEX,
   - les langages de modélisation universelle tel que UML,
   - les langages de description de scènes 3D,
   - les langages de description d'architecture tels que Darwin, Rapid,
   - les langages pour les protocoles dans les réseaux tel que ASN1,
   - les langages de script tel que Perl.

2 Les étapes de la compilation

Le processus de compilation est complexe. Il est ainsi nécessaire de décomposer la traduction en plusieurs étapes élémentaires. La Figure 3 illustre idéalement le schéma de la compilation. Il se décompose essentiellement de deux phases : la phase d'analyse et la phase de génération de code.

1. la phase d'analyse englobe l'analyse lexicale, l'analyse syntaxique et l'analyse sémantique,

2. la phase de génération de code consiste à la génération du code intermédiaire et à la génération du code objet.

Une phase parallèle est également enclenchée au début du processus de compilation dont les étapes sont invoquées tout le long du fonctionnement du compilateur. Cette phase englobe :

— la gestion de la table des symboles,

— la gestion des erreurs.

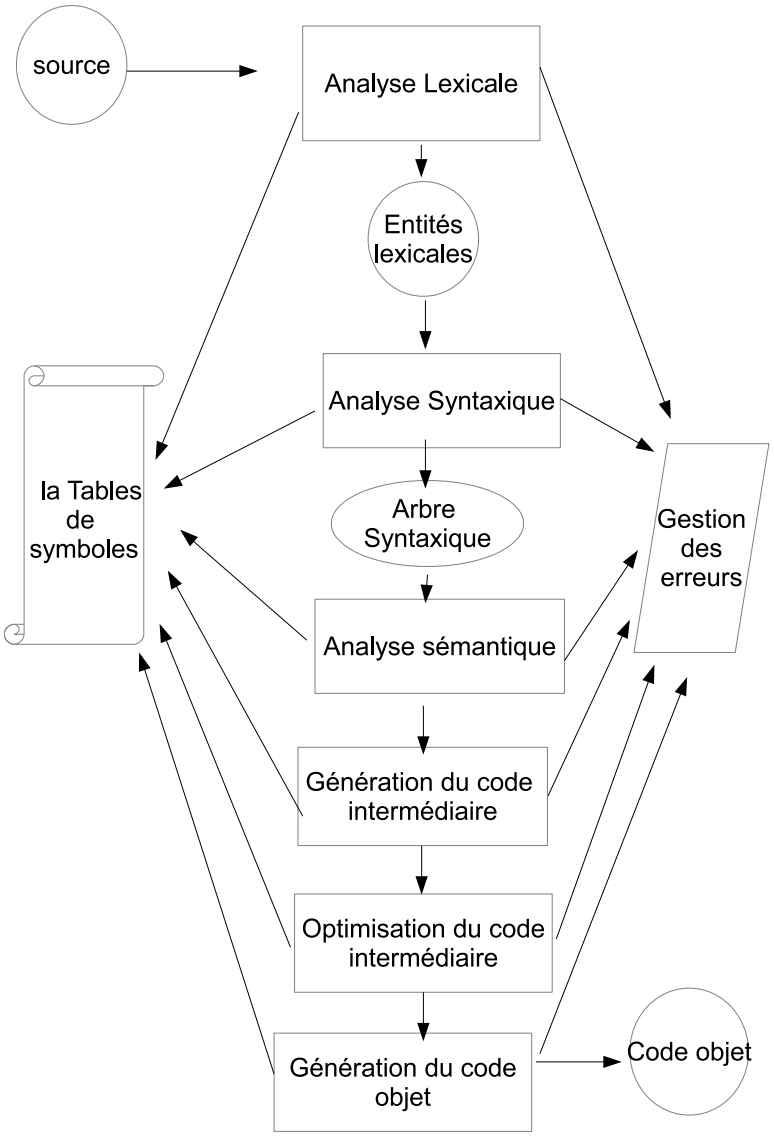


FIGURE 3 – les étapes de la compilation

2.1 Analyse lexicale

Le rôle de cette analyse est de déterminer les mots ou entités appartenant au langage à compiler et de construire le dictionnaire des entités appelé table des symboles. Le résultat de cette analyse est une succession d'entités lexicales appelée

chaîne codée (voir Figure 4).

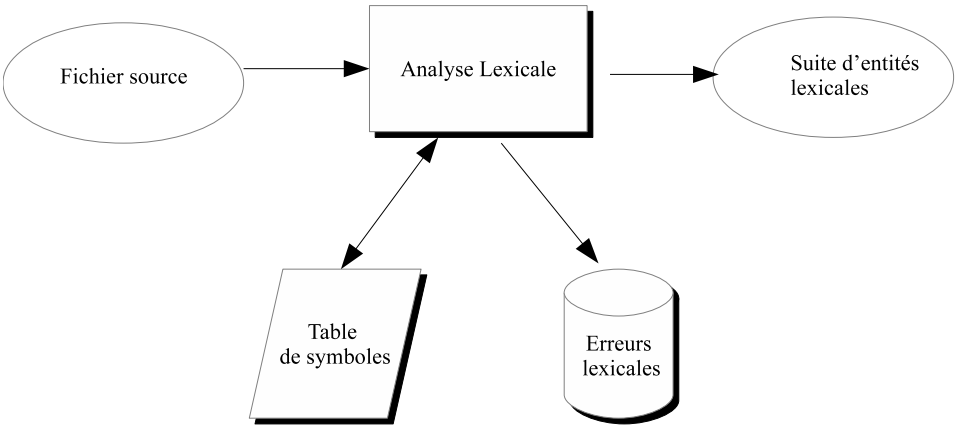


FIGURE 4 – Analyse lexicale

Exemple 0.1 *Considérons l'instruction conditionnelle if (delta > 0) then x := 0. Les différentes entités lexicales sont :* ***if delta > 0 ( ) then, x :=****. La chaîne codée obtenue : mot-clé, sep, identificateur, opérateur-comparaison, constante,séparateur, mot-clé, identificateur, opérateur-affectation, constante.*

2.2 Analyse Syntaxique

Le rôle de l'analyseur syntaxique est de vérifier si la structure de la chaîne codée obtenue de l'analyse lexicale est conforme à la grammaire syntaxique du langage considéré (voir Figure 5).

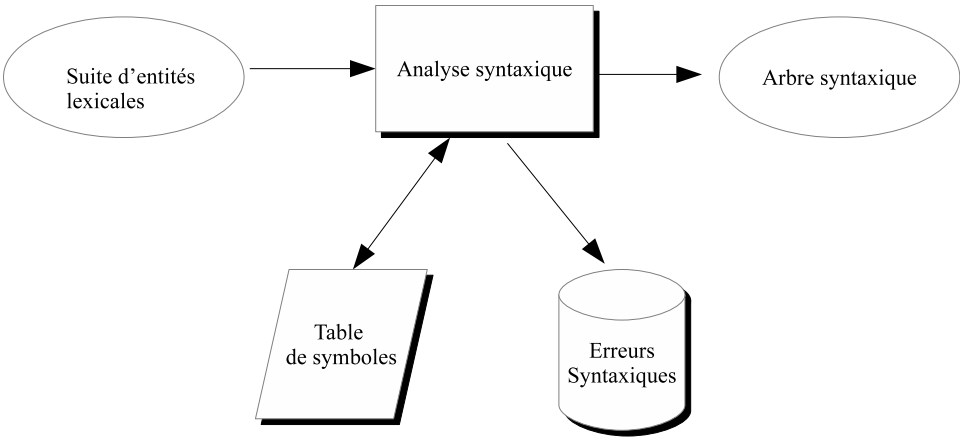


FIGURE 5 – Analyse syntaxique

2.3 Analyse sémantique

L'analyse sémantique permet de vérifier si la chaîne correcte syntaxiquement est aussi correcte du point de vue sens.

2.4 Génération du code intermédiaire

La génération de code intermédiaire permet de générer, à partir d'une entité syntaxique, un code simplifié de bas niveau dédié à une machine virtuelle ou abstraite.

Remarque 0.3 *En pratique, les phases d'analyse syntaxique, d'analyse sémantique et de génération du code intermédiaire se font en une seule étape constituant ainsi la phase de la traduction dirigée par la syntaxe durant qui en se basant sur une grammaire syntaxique la chaine des entités lexicales est analysée syntaxiquement et des routines sémantiques seront alors exécutées afin d'opérer des tests sémantiques et de générer le code intermédiaire correspondant (voir Figure 6).*

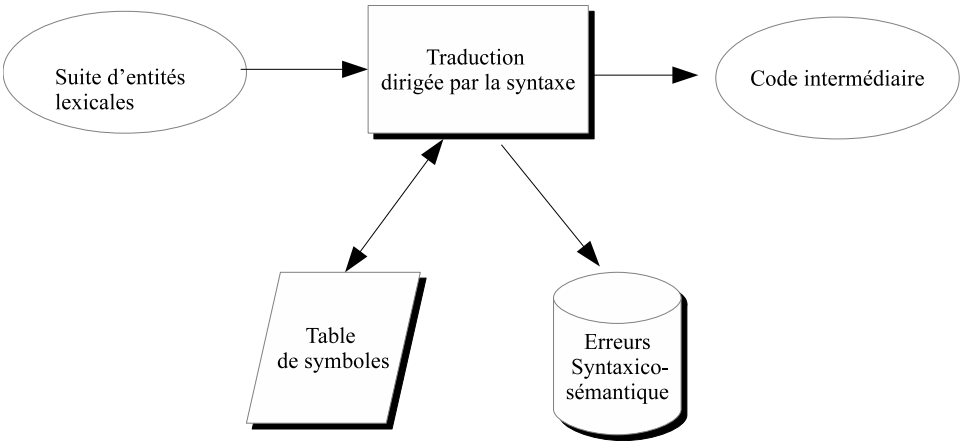


FIGURE 6 – Traduction dirigée par la syntaxe

2.5 Optimisation du code intermédiaire

Cette phase vise à optimiser le code du point de vue place mémoire et rapidité d'exécution. C'est une étape optionnelle dans les compilateurs.

2.6 Génération du code objet

Dans cette étape, il est nécessaire de connaître la forme du code objet à générer, ainsi que l'architecture de la machine réelle sur laquelle s'exécutera le programme (jeu d'instructions, registres, modes d'adressage, ...).

2.7 Gestion de la table des symboles

La table des symboles enregistre les identificateurs et les leurs attributs (emplacement mémoire, type, portée, valeur,...). La table des symbole est créée lors de la phase de l'analyse lexicale Chaque identificateur,qui représente une variable, a une entrée dans la table des symboles. Les attributs seront mis à jour au fur et à mesure de l'évolution du processus de compilation.

2.8 Gestion des erreurs

La détection des erreurs se fait lors des différentes phases de l'analyse. Il s'agit d'afficher au programmeur le type d'erreur d'une manière la plus précise possible. Le traitement d'erreur consiste :

— en cas d'erreurs mineurs, le compilateur procède au recouvrement afin de permettre la poursuite du processus de compilation,

— en cas d'erreurs fatales ou lorsque le nombre d'erreurs devient important, le processus de compilation est interrompu.

Exemple 0.2 *La Figure 7 illustre les différentes phases de la compilation appliquées à l'instruction d'affectation : delta :=b*b-4*a*c.*

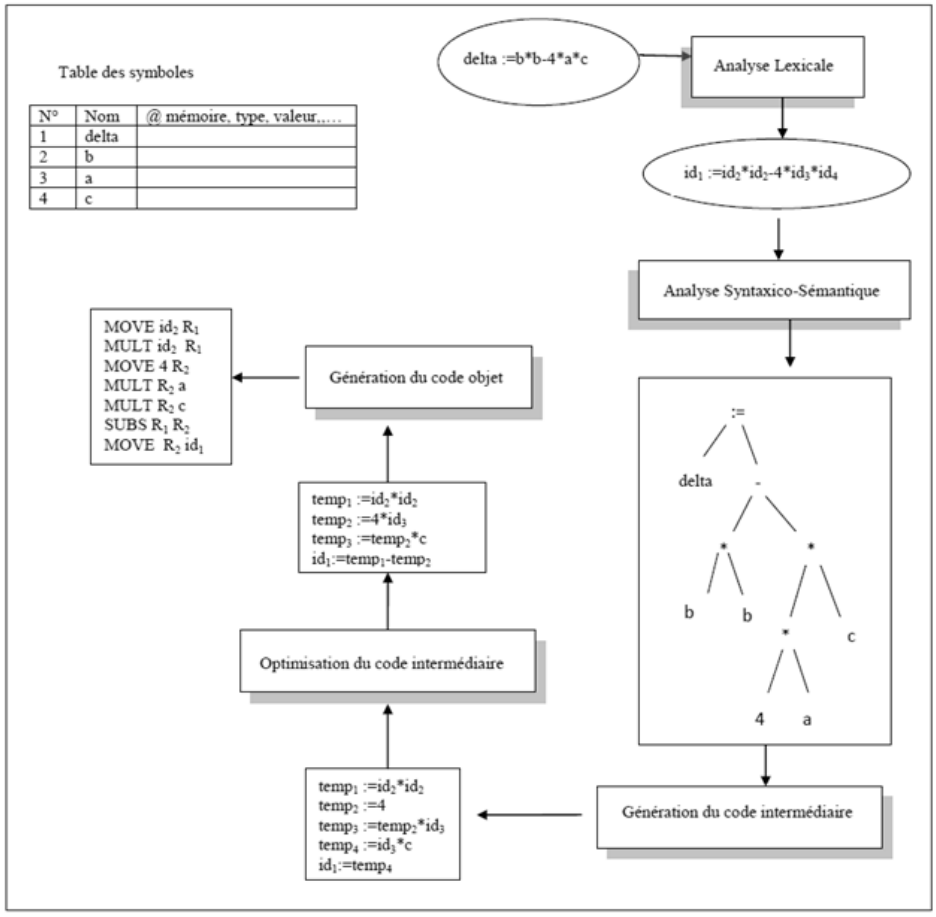


FIGURE 7 – Déroulement du processus de Compilation pour la chaine "delta :=b*b-4*a*c"

3 Compilateur, pré-compilateur, traducteur et interpréteur

Il existe d'autres transformations de programmes sources telles que la pré-compilation, la traduction et l'interprétation.

1. La traduction consiste à traduire un programme écrit dans un langage (L_1) en un programme écrit en langage (L_2). Le programme écrit dans (L_1) est appelé le source, celui écrit en (L_2) est appelé l'objet.
2. L'interprétation consiste à traduire et à exécuter un programme écrit en langage source, instruction par instruction. C'est à dire à partir du source, le résultat est donné directement. L'interpréteur traite les instructions au vol. Il ne produit pas de code objet. Il n'est pas alors possible de tracer l'exécution.
3. La pré-compilation consiste à traduire un programme écrit dans un langage (L_1) en un programme écrit dans un langage (L_2) tels que les langages (L_1) et (L_2)sont des langages évolués.

   Remarque 0.4 *Si le langage (L_1) est un langage évolué et le langage (L_2) est le langage machine ou l'assembleur, alors il s'agit d'un compilateur. Le compilateur est plus rapide, mais il est plus difficile à mettre en œuvre.*

4 Outils pour la construction de compilateurs

Il existe plusieurs outils d'aide à la réalisation de compilateurs. Ces outils sont en général adaptés aux différentes étapes du processus de compilation :

1. des générateurs d'analyseurs lexicaux ou syntaxiques tels que Flex et Bison,
2. des moteurs de traduction dirigée par la syntaxe,
3. des générateurs automatiques de code,
4. des logiciels intervenant dans le processus de traduction de programmes tels que les pré-processeurs, les assembleurs, les éditeurs de liens et les chargeurs (Voir Figure 8).
   - — un pré-processeur est un programme qui effectue des modifications textuelles (souvent relativement simples) sur le texte source ; en général le résultat du travail d'un pré-processeur doit être ultérieurement compilé.
   - — un assembleur est un compilateur pour un type de langage très proche du langage machine.

— un éditeur de liens a pour but de réunir plusieurs modules afin de produire un fichier exécutable. Ces modules sont des programmes objets compilés séparément ainsi que des bibliothèques (graphiques, mathématiques,...).

— un chargeur a pour fonction d'implanter le code exécutable dans la mémoire centrale et de lancer son exécution. L'édition de liens et le chargement sont deux étapes incontournables dans le processus conduisant à l'exécution d'un programme.

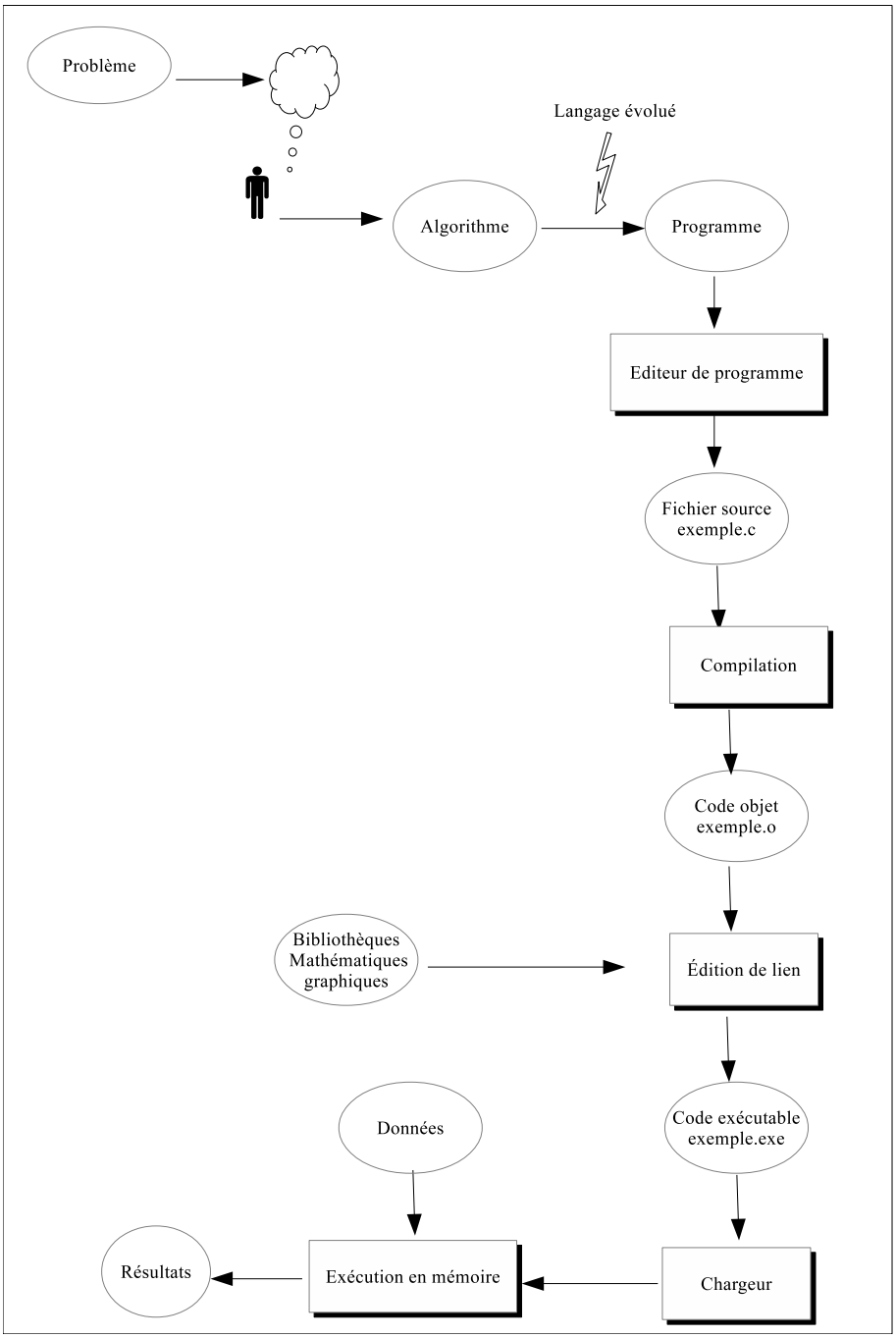


FIGURE 8 – La chaine de production d'un programme

Chapitre 1

Notions de Théorie des langages

1 introduction

Les langues naturelles permettent aux humains de communiquer entre eux. Elles ne sont pas forcément formelles elle peuvent être ambiguës . Par contre, les langages formels ont été définis par l'Homme afin de communiquer avec la machine. Ils doivent être non ambigus afin qu'ils puissent être interpréter par une machine. Ainsi, il est nécessaire de développer une passerelle entre le langage machine et les langages formels. Cette passerelle correspond au processus de compilation qui englobe des phases d'analyse (lexicale, syntaxique et sémantique) et des phases de génération de code (code intermédiaire et code objet).
L'analyse lexicale consiste essentiellement à lire le fichier source caractère par caractère et constitue en sortie une suite d'entités lexicales. L'analyse syntaxique a pour tâche de vérifier si la combinaison des entités lexicales obéit aux règles syntaxiques décrites par une grammaire qui génère le langage. Les notions de base relatives aux différentes étapes d'analyse se réfèrent à des notions de Théorie des Langages.
En théorie des langages, l'ensemble des entités élémentaires correspond à l'alphabet et la combinaison des entités élémentaires génère un mot. Un ensemble de mots définit un langage. Ce dernier est alors engendré par une grammaire et les entités du langage sont reconnues par un automate.
Dans la littérature, il existe différentes classes de langage qui correspondent à des différentes classes de grammaires et d'automates. Ainsi, les entités lexicales de la grammaire sont décrites par des grammaires régulières et sont reconnues par des automates à états finis déterministes et la description syntaxique du langage est réalisée grâce à l'utilisation des grammaires hors contextes et aux automates à pile.
Ce chapitre a pour objectif d'introduire les principales notions de la théorie des

langages afin de pouvoir aborder les différentes phases du processus de compilation.

2 les notions préliminaires

les éléments qui composent un langage se basent sur la notion de l'alphabet.

Définition 1.1 *Un alphabet est un ensemble non vide* Σ *de symboles.*

2.1 Mot

Définition 1.2 *Un mot est une séquence finie d'éléments du langage* Σ

Plusieurs caractéristiques sont associées à la notion de mot.

- $| x |$ spécifie la longueur du mot x exprimée par le nombre de symboles qui le composent.
- ϵ désigne le mot vide dont la longueur est nulle.
- x.Y correspond à la concaténation des deux mots X et Y.
- X^n désigne le mot X concaténé n fois.
- Le miroir d'un mot m , noté m^R est obtenu en inversant les symboles du mot m. Ainsi, si $m = a_1, a_2, ..., a_n$ alors $m^R = a_n, a_n - 1, ..., a_2, a_1$.
- Σ^+ correspond à l'ensemble des mots que l'on peut construire à partir de Σ.
- $\Sigma^* = \Sigma \cup \{\epsilon\}$.

Exemple 1.1 *Soit* $\Sigma = \{a, b, c, d\}$ *aabbc,cdda,b sont des mots du langage de* Σ^* *de longueur respectives égales à 5, 4 ,1.*

2.2 Langage

Définition 1.3 *Un langage sur un alphabet* Σ *est tout sous ensemble de* Σ^*.

Exemple 1.2 *Soit* L_1 *un langage défini à partir de l'alphabet* $\Sigma = \{a, b, c, d\}$ *représentant l'ensemble de mots qui commencent par a et se terminent par b et contenant un nombre pair de d. Il correspond à la liste infinie de mots du type abcddab,addbddab,addb,.... Le mot adb* $\notin L_1$.

2.2.1 Opérations définies sur les langages

Soient $L_1 et L_2$ deux langages définis sur deux alphabets $\Sigma_1 et \Sigma_2$.

— L'union des langages $L_1 et L_2$ est le langage défini sur $\Sigma_1 \cup \Sigma_2$ contenant tous les mots qui appartiennent soit à L_1 soit à L_2. Ainsi,

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

.

— L'intersection des langages $L_1 et L_2$ est le langage défini sur $\Sigma_1 \cap \Sigma_2$ contenant tous les mots qui appartiennent à L_1 et à L_2. Ainsi,

$$L_1 \cap L_2 = x : x \in L_1 et x \in L_2$$

.

— Le complément d'un langage L_1 est le langage défini sur Σ_1 contenant tous les mots qui n'appartiennent pas à L_1 . ainsi,

$$C5L_1) = x : x \notin L_1$$

.

— La différence de deux langages $L_1 et L_2$ est le langage défini sur Σ_1 contenant tous les mots qui appartiennent à L_1 et qui n'appartiennent pas à L_2. Ainsi,

$$L_1 - L_2 = x : x \in L_1 ou x \notin L_2$$

.

— La concaténation des langages $L_1 et L_2$ est le langage défini sur $\Sigma_1 \cup \Sigma_2$ contenant tous les mots formés d'un mot de L_1 suivi d'un mot de L_2. Ainsi,

$$L_1.L_2 = xY : x \in L_1 et y \in L_2$$

.

— La fermeture itérative d'un langage L_1 et l'ensemble des mots formés par une concaténation finie de mots du langage L_1. Ainsi,

$$L^* = x : \exists k \geq 0 et x_1, ..., x_k \in L_1 tel que x = x_1 x_2 ... x_k$$

.

Définition 1.4 Clôture de Kleen
Soit L un langage défini sur un alphabet Σ. La clôture de Kleen du langage L, notée L^*, est l'ensemble des chaînes qui sont formées à partir d'un nombre quelconque (et éventuellement nul) de chaînes du langage L en les concaténant.

Ainsi :

— $L^0 = \epsilon$,

— $L^1 = L$,

— $L^2 = LL$,

— $L^i = LL...L : concaténation i fois de L$,

Définition 1.5 Langage régulier (Kleen 1956)
Un langage régulier L sur un alphabet Σ est défini d'une manière récursive comme suit :

1. **$\emptyset$ est un langage régulier sur Σ**
2. **ϵ est un langage régulier sur Σ,**
3. **si a $\in \Sigma$, alors, a est un langage régulier sur Σ,**
4. **si R est un langage régulier sur Σ alors R^* est un langage régulier sur Σ,**
5. **si R et Q sont deux langages réguliers sur Σ alors $R \cup Q$ et $P.Q$ sont des langages réguliers sur Σ.**

Un langage régulier est décrit par une expression régulière.

2.2.2 Expressions régulières

Une expression régulière sur un alphabet Σ est définie comme suit :

1. ϵ est une expression régulière qui décrit le langage ϵ,
2. si a $\in \Sigma$ alors a est une expression régulière qui décrit le langage a,
3. si r est une expression régulière qui décrit le langage R alors :
   — $(r)^*$ est une expression régulière qui décrit le langage R^*,
   — $(r)^+$ est une expression régulière qui décrit le langage R^+.
4. si r et s sont des expressions régulières qui décrivent respectivement les langages R et S alors :
   — $(r) \mid (s)$ est une expression régulière qui décrit le langage $R \cup S$,
   — $(r)(s)$ est une expression régulière qui décrit le langage RS,

Exemple 1.3 *L'expression régulière* $(a^n b^* c^n)^*$ *dénote :*

- *l'ensemble des mots, définis sur l'alphabet* $\Sigma = a, b, c$*, qui commencent par* n *occurrences du symbole* a*, suivies par* o *ou plusieurs occurrences du symbole* b *et se terminent par* n *occurrences du symbole* c *ou*
- *la chaîne vide*

Ainsi, cette expression régulière décrit une liste infinie de mots tels que $\epsilon, abc, aacc, ...$

Extensions des notations

3 conclusion

Chapitre 2

Analyse Lexicale

1 introduction

L'analyse lexicale constitue la première étape du processus de compilation. Sa principale tâche consiste à lire le fichier source caractère par caractère et de produire comme résultat une suite d'éléments lexicaux ou mots appelés entités lexicales afin de faciliter l'analyse des prochaines étapes.

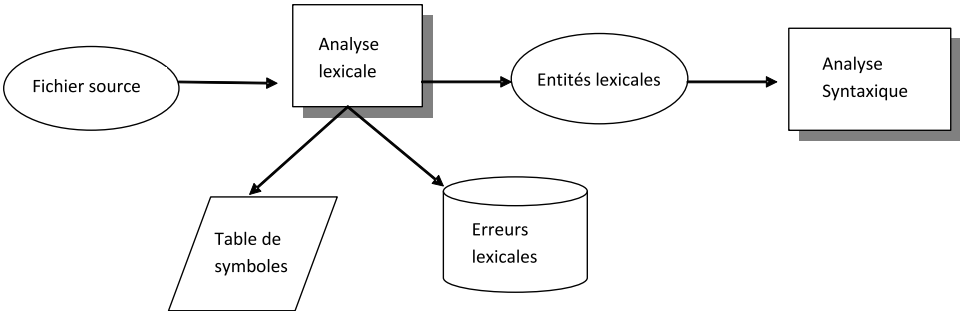


FIGURE 2.1 – les étapes d'un analyseur lexical

En plus, l'analyseur lexicale réalise des tâches secondaires comme :

— Élimination des caractères superflus tels que les commentaires, les tabulations, les fins de ligne,

— Gestion des numéros de ligne du programme source afin de faciliter la gestion des erreurs qui seront générées lors des différentes phases d'analyse,

— Construction de la table de symboles qui regroupe touts les informations relatives aux identificateurs déclarés et utilisés dans le programme source.

Avant d'entamer la mise en œuvre d'un analyseur lexical, il est important de faire un rappel sur quelques notions de la théorie des langages.

2 Quelques définitions de base

Définition 2.1 *Une entité lexicale est une suite de caractères qui possède une signification collective.*

Exemple 2.1 *En langage C, un identificateur est une suite non vide de caractères qui commence par une lettre suivie de chiffres ou de lettres ou du symbole _.*
Un entier est une suite non vide de chiffres précédée éventuellement par un signe+, −.

Définition 2.2 *Grammaire Une grammaire G définie par le quadruplet <T,N,S,P> est telle que :*
T :représente les terminaux de la grammaire G
N : constitue l'ensemble des non terminaux de la grammaire G
S : est l'axiome de la grammaire G
P : représente l'ensemble des règles de production de la forme $A \rightarrow \alpha$ *où* $A \in N$ *et* $\alpha \in (T \cup N)^*$.

Définition 2.3 *Grammaire régulière droite Une grammaire G est dite régulière à droite si toutes les règles de production sont du type :* $A \rightarrow aB \mid a$ *où* $a \in T$ *et* $B \in N$.

grammaire régulière gauche Une grammaire G est dite régulière à gauche si toutes les règles de production sont du type : $A \rightarrow Ba \mid a$ *où* $a \in T$ *et* $B \in N$.

Définition 2.4 *Automate Un automate A est défini par <T,N,S,F,I> où :*

- *T est l'ensemble des terminaux,*
- *N est l'ensemble des états de l'automate,*
- *S est l'état initial de l'automate,*
- *F est l'ensemble des états finaux,*
- *I représente l'ensemble des transitions.*

3 Mise en œuvre d'un analyseur lexical

Les entités lexicales dont décrites à l'aide de grammaires régulières qui sont reconnues à l'aide des automates d'états finis. Ainsi, la mise en œuvre d'un analyseur lexical nécessite la construction d'un automate à états finis déterministe permettant de reconnaitre toutes les entités du langage. L'automate est obtenu en :

1. définissant l'ensemble des entités lexicales du langage,
2. en associant à chaque entité lexicale une expression régulière,
3. en associant à chaque expression régulière l'automate de reconnaissance correspondant,
4. en regroupant l'ensemble des automates en un automate à états finis,
5. en transformant l'automate obtenu en un automate à états finis déterministe.

3.1 Spécification des entités lexicales

Les entités lexicales sont décrites par des expressions régulières.

Exemple 2.2 *la description d'un identificateur en langage C devient :*
Lettre $= A-Z|a-z$
Chiffre$= 0-9$
Sep$=-$
Identificateur$=(lettre|sep)(lettre|chiffre|sep)^*$.

D'une manière générale, un langage regroupe les entités lexicales suivantes :

— les mots clés qui pour certains langages sont réservés,

— les identificateurs,

— les séparateurs,

— les opérateurs (numériques, booléens, de comparaison,...),

— les constantes (numériques, booléennes, chaines,...).

3.2 l'analyseur lexical

Le rôle de l'analyseur lexical est de reconnaitre les différentes entités lexicales. En théorie des langages, les automates sont des machines théoriques permettant de reconnaître des mots. En particulier, un langage régulier est reconnu par un automate déterministe à états finis. Ainsi, pour la mise en place d'un analyseur lexicale, les différentes entités lexicales composant un langage seront reconnues par un automate à états finis déterministe.
La représentation d'un automate en mémoire se fait à l'aide d'une matrice(m,n), appelée matrice de transition où m représente le nombre des différents états de l'automate et n représente le nombre des terminaux.

Exemple 2.3 *L'automate représenté par la Figure 1.2 reconnait l'entité lexicale représentant les* ***identificateur*** *décrites par l'expression régulière de l'Exemple 1.1.*

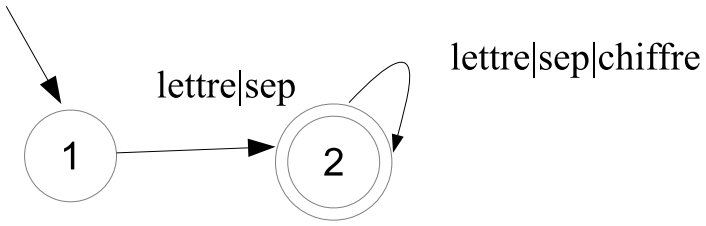


FIGURE 2.2 – Automate correspondant à l'expression régulière de l'exemple 1.1

Le fonctionnement de l'analyseur lexical est décrit par l'algorithme suivant :

```
Begin
      Lire(entite);
       Tc → première entité de la chaine;
       Ec → état initial;
while Ec ≠ ∅ do
   Ec → I(Ec,Tc);
   Tc → Ts;
end while
if Ec = ∅ then
   ecrire ('chaine incorrecte lexicalement');
else if Ec ∉ ETAT FINAUX then
   ecrire ('chaine incorrecte lexicalement');
else
   ecrire ('chaine correcte);
   coder l'entité;
   if code='IDF' then
      insérer dans la Table des symboles;
   end if
end if
End
```

Algorithme 1 : Reconnaissance des entités lexicales

4 Gestion de la table de symboles

Généralement, la table de symboles est créée dynamiquement lors de la phase lexicale. Puis, elle est complétée au fur et à mesure de l'avancement des phases d'analyse (syntaxico-sémantique). La table des symboles réunit des informations sur les identificateur : nature, type, valeur initiale, propriétés d'allocation : Pour toute variable, elle garde les informations suivantes : son nom, son type, sa portée, sa visibilité et son adresse en mémoire. Les propriétés d'un identificateur peuvent être de structures complexes (arbres, tables, listes, champs...). Pour tout identificateur désignant un nom de fonction ou de procédure, des information spécifiques des paramètres sont fournis telles que les types des arguments, leurs modes de passage, les type des résultat fournis, etc. Pour permettre un accès rapide, la table des symboles est indexée par le nom de l'identificateur (chaîne de caractères), d'où l'utilisation d'une table de hachage.
Lors de la définition d'un identificateur, l'analyseur lexical crée une entrée dans la table de symboles. Cette déclaration est généralement unique, à moins d'une surcharge. Les attributs associés à chaque identificateur seront insérés lors de l'analyse syntaxique. L'analyseur sémantique se servira, par exemple, de la table de symboles pour vérifier les concordances de types. La table des symboles est également utilisée lors de la phase génération de code. Dans certains compilateurs, les mots-clés du langage ne font pas partie de la syntaxe mais sont insérés dans la table des symboles.
D'une manière générale, afin de gérer la table des symboles, deux procédures élémentaires doivent être définies : la recherche d'un identificateur dans la TS et l'insertion dans la TS. Ainsi, à la rencontre d'une entité lexicale qui correspond à la description d'un identificateur, l'analyseur lance la procédure de recherche de l'entité dans la TS Recherche(TS,indice) ou lookup(entité, indice). Si celle ci est présente alors l'entrée dans la TS est restituée sinon, la procédure insert(entité, indice) est lancée afin d'insérer l'entité dans la TS et son indice sera alors retourné pour la codification.

Exemple 2.4 *Soit la partie d'un programme C suivante :*
int a,b = 10 ;
float taux ;
double cout ;
Main()

fscanf(count) ;

Une structure simple de la table de symbole engendrée lors de l'analyse lexicale

est représentée par la Table 1.1.

TABLE 2.1 – Table de symboles

Numéro	Nom	Type	adresse	valeur
1	a			
2	b			
3	taux			
4	count			

5 Générateur de l'analyse lexicale lex

Un analyseur lexical, lexer ou encore scanner peut être développé :

1. manuellement
2. en utilisant un automate décrit par une table de transition et un programme l'exploitant
3. en utilisant un générateur d'analyseur lexical tel que Flex. Le générateur de l'analyseur lexical facilite considérablement sa mise en œuvre. En effet, dans le fichier lex, il suffit principalement de décrire les entités lexicales par des expressions réguliers, puis d'associer à chaque expression régulière les actions à exécuter en langage C.

Exemple 2.5 *Soient les entités lexicales suivantes : <>,<=,<<.*

1. *manuellement : L'algorithme permet d'affecter à chaque opérateur un code différent afin de les distinguer. les variables tc et ts désignent respectivement le terme courant et le terme suivant.*

```
Debut
      SWITCH(tc)
        CASE '<' ; tc :=ts ;
        SWITCH(tc)
STATE          CASE '>' :code :=1 ; tc :=ts ;BREAK ;
          CASE '=' :code :=2 ; tc :=ts ;BREAK ;
          CASE '<' : code :=3 ; tc :=ts ;BREAK ;
          DEFAULT : erreur ;
End
```

Algorithme 2 : Reconnaissance des entités lexicales manuellement

2. *les entités lexicales <>, <=, « peuvent être reconnues à l'aide de l'automate illustré par la Figure 1.3 et décrit par la table de transition 1.2 :*

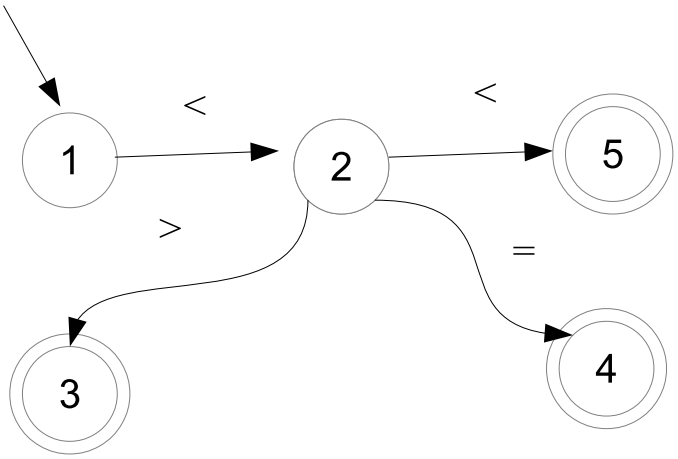


FIGURE 2.3 – Automate de reconnaissances des entités <>, <=, <<

TABLE 2.2 – Table de transitions associée à l'automate décrivant les entités <> ,<=, <<

	<	>	=
1	2	0	0
2	5	3	4
3	0	0	0
4	0	0	0
5	0	0	0

L'algorithme de reconnaissance utilisant l'automate décrit précédemment est défini comme suit :

```
Debut
      Lire(entité);
       Tc ← première entité de la chaine;
       Ec ← 1;
       EtatsFinaux ← {3,4,5};
while (Ec ≠ 0)(Ec ≠ #) do
   Ec ← I(Ec,Tc);
   Tc ← Ts;
end while
if (Ec = 0) then
   ecrire ('chaine incorrecte lexicalement');
else if (Ec ∉ ETAT FINAUX) then
   écrire ('chaine incomplète');
else
   écrire ('chaine correcte);
   if (Ec = 3) then
      coder (opérateur,"<>");
   else if (Ec = 4) then
      coder (opérateur,"<=");
   else
      coder (opérateur,"<<");;
   end if
end if
End
```

Algorithme 3 : Reconnaissance des entités lexicales

3. *En utilisant Flex :*
   La reconnaissances des entités lexicales <>,<=,« se traduit par la description Flex suivante :

```
"<>" return 1;
"<=" return 2;
"«" return 3;
tc=yylex(); *lancement de l'analyseur lexical
```

Algorithme 4 : Reconnaissance des entités lexicales à l'aide de l'outil flex

6 Générations des erreurs lexicales

Peu d'erreurs sont détectables lors de l'analyse lexicale car ce dernier a une vision locale du programme source. Les erreurs se produisent lorsque une entité

lexicale ne correspond à aucune des descriptions définies par l'analyseur. Après avoir détecté une erreur, l'analyse se poursuit afin que d'autres erreurs puissent être détectées. Néanmoins, il y a un seuil à ne pas dépasser. Un nombre important d'erreurs peut entrainer l'arrêt de l'exécution du compilateur. A titre d'exemple, en Pascal,

— L'entité **1gamma** n'est pas une entité reconnue.

— La constante **1223445555555554** est un entier trop long.

Lorsque l'analyseur est confronté à une suite de caractères qui ne correspond à aucune description, il adopte différentes stratégies :

1. mode panique : les caractères non reconnus seront ignorés jusqu'à la rencontre d'un séparateur,
2. mode recouvrement : le texte sera transformé en insérant ou en remplaçant des caractères.

Exemple 2.6 *Soit l'instruction en Pascal : a :=1while(a>10)do x :=x+1 ; do*
Ainsi, en mode panique, les caractères seront ignorés jusqu'à la rencontre du séparateur ;. Par contre, en mode de recouvrement d'erreurs, un espace sera inséré après l'entité lexical 1 qui correspond à la description d'un nombre.

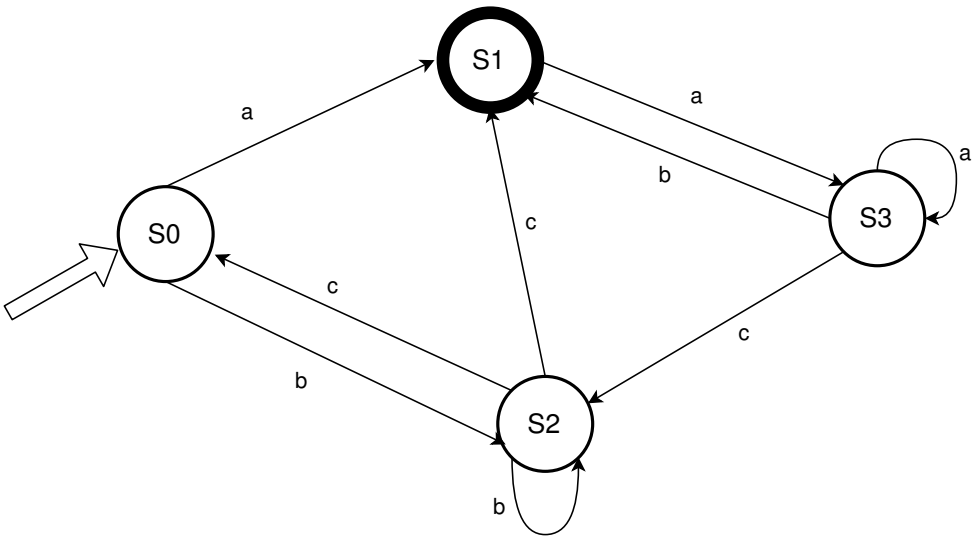
7 conclusion

Le processus de l'analyse lexicale est de complexité linéaire par rapport au nombre des entités lexicales qui composent le programme source présenté en entrée du compilateur. Néanmoins, lors de la mise en œuvre d'un analyseur lexicale, il n'est pas pratique de l'implémenter en utilisant un automate. Il existe pour cela des outils permettant d'écrire des programmes simulant le fonctionnement des automates à partir de simples expressions régulières. Lex est un générateur d'analyseur lexical. Flex est la version libre de Lex. Il prend en entrée un ensemble de règles lexicales définissant des descriptions et les actions à exécuter lorsque l'analyseur rencontre les description. Ces descriptions sont enregistrées sous forme d'un fichier.l. Flex transforme par la suite les règles en un code C correspondant à l'analyseur lexicale en générant ainsi le fichier lex.yy.c. il suffit par la suite de le compiler afin d'obtenir l'analyseur lexical exécutable.

8 Exercices

Exercice 1 :

Soit l'automate suivant :



Donnez l'automate simple déterministe correspondant.

Exercice 2 :

On veut analyser le langage L définis sur $\{0,1\}^*$ et présentant les caractéristiques suivantes :

- Le langage est formé des entités de type X,Y et Z (chaines binaires) ;
- Une entité X commence par '0' et ne contient pas deux '0' consécutifs ;
- Une entité Y commence par un '1' qui est suivi d'un nombre pair de '0' (il y a au moins deux) ;
- Une entité Z commence par deux '0' ou bien deux '1' suivis par une séquence de '0' et de '1' ne contenant la sous chaine '101'.

1. Construire un automate déterministe reconnaissant les entités X,Y et Z.
2. Écrire un algorithme qui fait l'analyse lexicale du langage.

Exercice 3 :

On souhaite écrire un analyseur lexicale pour le langage de commande suivant :

- Une commande est composée d'un nom de commande, suivi d'une liste optionnelle d'arguments, suivie d'une liste facultative d'options ;

— Une liste d'arguments est une suite d'argument ;
— Une liste d'options est une suite non vide d'options encadrée par '[' et ']', à l'intérieur de laquelle les options sont séparées par ',' ;
— Une option est un caractère précédé d'un tiret ;
— Un argument est un identificateur, de même qu'un nom de commande.
— par exemple :
  - macom arg1 arg2 [-a -b]
  - macom [-f]
  - macom

1. Quelles sont les entité lexicales nécessaires à la description d'une commande ?
2. Donner pour chacune d'entre elle une description régulière qui la définit.

Chapitre 3

L'outil Flex

1 Introduction

Lors de la mise en œuvre d'un analyseur lexicale, il n'est pas pratique de l'implémenter. Il existe pour cela des outils permettant d'écrire des programmes simulant le fonctionnement des automates à partir de simples expressions régulières. Lex est un générateur d'analyseur lexical. Flex est un freeware (la version libre de lex) disponible à partir de http ://flex.sourceforge.net. Il prend en entrée un ensemble de règles lexicales définissant des descriptions et les actions à exécuter lorsque l'analyseur rencontre les description. Ces descriptions sont enregistrées sous forme d'un fichier.l. lex transforme par la suite les règles en un code C correspondant à l'analyseur lexicale en générant ainsi le fichier lex.yy.c. il suffit par la suite de le compiler afin d'obtenir l'analyseur lexical exécutable.

Flex et Bison sont des utilitaires de compilation définis sur les systèmes Unix. Ils sont très utiles pour tout processus de compilation relatif à tout les langages de communication avec la machine tels que les langages de programmation de haut nivau (C,Pascal,Fortran) les langages orientés objets(C++,Java,PHP,Phyton), les langages de description (XML,HTML) PHP, les protocoles (FTP, HTTP).
Flex (version GNU de Lex) est un générateur d'analyseur lexical défini sur les systèmes d'exploitation UNIX. Il construit un analyseur lexical à partir d'un ensemble de règles et d'actions décrites sous formes d'expressions régulières. Il est agencé (voir fig 1) avec le compilateur des compilateur Bison (version GNU de YACC acronyme de Yet Another Compiler of Compilers). Ce dernier construit un compilateur d'un langage décrit par un ensemble de règles et d'action d'une grammaire LALR. Le fonctionnement de flex est résumé par la figure suivante :

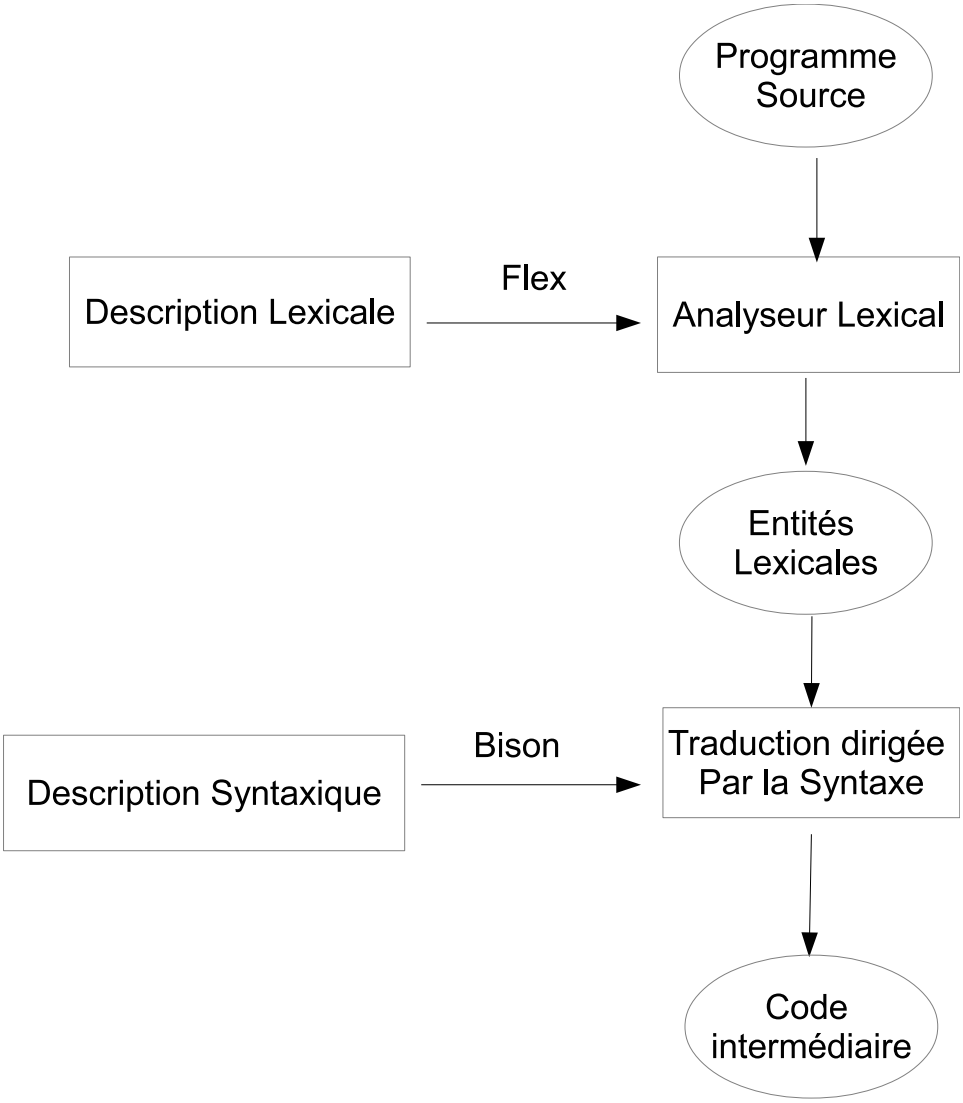


FIGURE 3.1 – Processus de compilation utilisant Flex et Bison

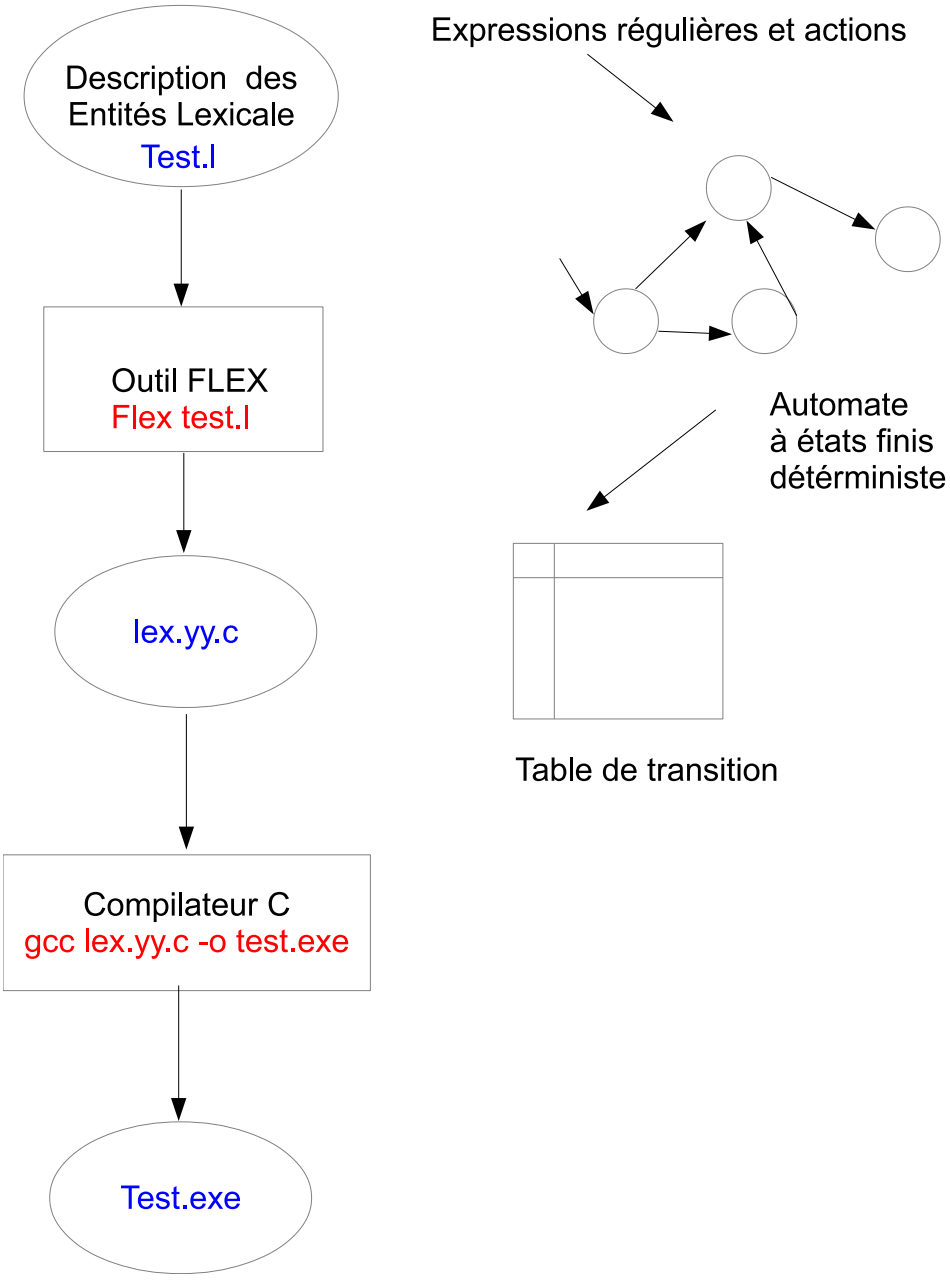


FIGURE 3.2 – Fonctionnement de Flex

2 structure d'un fichier en flex

Le fichier de règles pour LEX se divise en trois parties séparées par les balises %{, %}, %%, %%. La première partie du fichier contient les déclarations C ainsi que les déclaration des éléments du lexique décrits par des expressions régulières. La seconde partie du fichier décrit les actions à effectuer lors de la détection des différents éléments lexicaux décrits dans la partie précédente. Chacune des entrées contient deux blocs entre accolades : Le premier bloc entre accolades contient le nom de l'élément lexical considéré, Le second bloc entre accolades contient le code C de l'action à exécuter. La troisième partie contient le code principal de l'analyseur lexical ainsi que les procédures si nécessaire. Ainsi, un programme Flex se présente comme suit :
% Déclarations en C
%
Définitions des éléments lexicaux
%%
Associations des éléments lexicaux et des actions
%%
Définition des procédures et du code principal

2.1 Les déclarations

Cette partie d'un fichier Flex est constituée :

— du code écrit dans le langage cible, encadré par %{ et %}. Les fichiers à inclure sont spécifiés dans cette partie, tels que <stdio.h>, <stdlib.h>.

— des expressions régulières définissant des notions non terminales, telles que lettre, chiffre et nombre. Ces spécifications permettent d'associer à chaque notion une expression régulière.

Exemple 3.1 *blancs* [
t
n]+
lettre $[A-Za-z]$
chiffre $[0-9]$

2.1.1 Les expressions régulières

Une expression régulière en Flex se compose de caractères et de méta-caractères ayant une spécification spéciale tels que : $$,

TABLE 3.1 – Les expressions régulières

Expression	Signification	Exemple
c	tout caractère c qui n'est pas un méta-caractère	a
c	si c n'est pas une minuscule, ça désigne c littéralement	
+		
"s"	la chaine de caractères s littéralement	"abc*
$r_1 r-2$	r_1 suivie de r_2	ab
.	n'importe quel caractère excepté le retour ligne	aab*
∧	comme premier caractère de l'expression, il signifie le début de ligne	$\wedge abc$
$$	comme dernier caractère de l'expression, il signifie la fin de ligne	
[s]	n'importe des caractères constituant la chaine s	[abc]
[∧s]	n'importe des caractères à l'exception de ceux constituant la chaine	[∧abc]
r^*	0 ou plusieurs occurrences de r	b^*
r^+	1 ou plusieurs occurrences de r	b^+
r?	0 ou 1 occurrence de r	a?
r{m}	m occurrences de r	b{3}
r{m,n}	m à n occurrences de r	b{3,6}
$r_1 \mid r_2$	r_1 ou r_2	a|b
$r_1 \mid r_2$	r_1 si elle est suivie de r_2	ab|cd
(r)	r	(a|b)
n	caractère retour ligne	
‿	tabulation	
{}	pour faire référence à une définition régulière	{nombre}
"EOF"	fin de fichier	

, [], ,, +, /, <, >, −, ∗, |, ?. Le tableau précèdent spécifie la liste des expressions régulières de Flex. Il est à noter que Flex fait la différence entre les minuscules et les majuscules.

2.2 les définitions des symboles

Cette partie sert à indiquer à Flex l'action à exécuter à la rencontre d'une expression. Elle peut contenir :

— des spécifications écrites dans le langage cible, encadrées par de ligne), qui seront placées au début de la fonction yylex(), la fonction chargée de consommer les terminaux, et qui renvoie un entier.

— des productions associant à chaque expression régulière l'action à exécuter.

Si aucune action n'est spécifiée, Flex recopiera les caractères tels quels sur la sortie standard.

Les commentaires tels que /* ... */ ne peuvent être présents dans la deuxième partie d'un fichier Flex que s'il sont placés dans les actions parenthésées. Dans le cas contraire, ils seront considérés par Flex comme des expressions régulières ou des actions, ce qui engendrer des messages d'erreur.
La compilation d'une source Flex produit une fonction *yylex()*. Un appel de *yylex()* déclenche une analyse lexicale du flux *yyin* (par défaut, il représente stdin). Durant le processus de reconnaissance des entités lexicales, l'analyseur teste les descriptions une à une jusqu'à rencontrer celle qui correspond à l'entité. Cette dernière sera alors chargée dans la variable *yytext*, représentée par un tableau de caractères et sa longueur sera affectée à *yyleng*. La variable *yyout* est une variable prédéfinie qui représente le fichier d'écriture (stdout par défaut). La fonction *yywrap()* est appelée à partir de la fonction *yylex()*. Elle peut être utilisée en la définissant dans la partie réservée aux fonctions auxiliaires.

2.3 Le code additionnel

Cette section du fichier flex contient les implémentations C des fonctions nécessaires. main()
yylex() ;

La fonction *main()* contient par défaut juste l'appel à la fonction **yylex()** mais l'utilisateur pet la redéfinir dans la section des fonctions auxiliaires.
yylex() est une fonction qui lance l'analyseur lexicale. Elle appelle la fonction **yywrap()**. Celle ci est toujours appelée en fin de flot d'entrée. Par défaut, elle retourne 0 si l'analyse doit se poursuivre et 1 sinon. L'utilisateur peut néanmoins la redéfinir dans la section des fonctions auxiliaires.
L'analyse peut être stoppée par la fonction *yyterminate()*.

Exemple 3.2 *Considérons un premier exemple qui compte le nombre de voyelles, de consonnes, de caractères, de caractères de ponctuation, de lignes, de lettres en majuscules et de lettres en minuscules. %*
int nbVoyelles, nbConsonnes, nbPonctuation, nbCar, nbLignes, nbMajuscules, nbMinuscules ; % consonne [b-d f-h j-n p-x z]
ponctuation [, ; : ? ! .]
voyelle [a e i o u y]
majuscule [A-Z]
minuscule [a-z]
%% nbVoyelles++ ;

```
consonne nbConsonnes++;
ponctuation nbPonctuation++;
voyelle nbVoyelles++;
majuscule nbMajuscules++;
minuscule nbMinuscules++;

n nbLignes++;
. nbCar++;
%% main()

nbVoyelles=nbConsonns=nbPonctuation=nbCar=nbLignes=nbMajuscules=nbMinuscules=0;
yylex();
printf( nbVoyelles, nbConsonnes, nbPonctuation, nbCar, nbLignes, nbMajuscules,
nbMinuscules;)
```

Exemple 3.3 *Considérons un deuxième exemple de fichier Flex qui reconnait certaines catégories lexicales :*

```
%
#include <stdio.h>
#include "y.tab.h>
%
%%
"=" return EQ;
"!=" return NE;
"<" return LT;
">" return GT;
">=" return GE;
"<=" return LE;
" :=" return ASSIGN;
"+" return PLUS;
"*" return MULT;
"IF" return IF;
"THEN" return THEN;
"ELSE" return ELSE;
"WHILE" return WHILE;
"PRINT" return PRINT;
[0-9]+ yyval=atoi(yytext); return NUMBER;
[a-z]+ yyval=yytext; return NAME;
```

n nextline() ;
. yyerror("illegal token") ;
%% #ifndef yywrap
yywrap() return 1 ;
#endif

Exemple 3.4 *Considérons un troisième fichier Flex permettant de reconnaitre quelques entités lexicales du langage JAVA :*
%
#include <stdio.h>
#include <string.h>
%
CHIFFRE [0-9]
LETTRE [a-z][A-Z]
MOTRESERVE abstract|double|int|static|class|while|boolean|else|if|public|long|void
*OPERATEUR <|=|+|<=|**
%%
MOTRESERVE printf «MR%S »,yytext) ;
OPERATEUR printf « OP%S »,yytext) ;
LETTRE(LETTRE|CHIFFRE) printf (« id %S »,yytext) ;*
CHIFFRE printf("CONST-STRING ") ;*
%%
Main()
printf ("Taper CTRL+D pour stopper."
n) ;
yylex() ;

Afin se simuler le fonctionnement de l'analyseur précédent, considérons le fichier source JAVA suivant : Class factorielle
Public static void main()
Long x = 1 ;
Int n=0 ;
Int max=30 ;
While(n<max)
n=n+1 ; incrementation
*x=x*n ; multiplication*

Le programme sera transformé comme suit par le programme lex :
MRclass id
Mr public MR static MRvoid id-fonction OP
MR long id OP= CONST-INT
MR int id OP= CONST-INT
MR int id OP= CONST-INT
MRwhile (id OP< id)
Id OP= id OP+ CONST-INT ; COMMENTAIRE
Id OP= id OP CONST-INT ; COMMENTAIRE*

3 Compilation avec Flex

le nom d'un fichier Flex doit avoir le suffixe **.lex**. Comme le fichier source contient des instructions en Flex et du code C, la compilation se fait en deux étapes :

1. la compilation Flex avec la commande : **flex fichier.lex**. Dans le cas où aucune erreur n'est détectée, Flex construit un fichier source en C **yy.lex.c**,
2. la compilation C du fichier obtenu en output après la première étape avec la commande : **gcc lex.yy.c -o fichier.exe**.

L'exécution du fichier output se fait comme suit : **./fichier.exe**. Le fichier yy.lex.c
Les options de la commande de compilation Flex sont les suivantes :

— -d pour invoquer le mode debug

— -i pour ne pas faire la différence entre les majuscules et les minuscules

— -l Pour avoir le comportement de Flex

— -s pour supprimer l'action par défaut

Exemple 3.5 *Le fichier Flex décrit ci-après correspond à un mini interprète des expressions arithmétiques formées par des nombres réels et des opérateurs arithmétique classiques* $(+, *, -, /)$.
#include "global.h" #include "calc.tab.h" #include <stdlib.h> blancs [ ]+ chiffre [0-9] entier chiffre+ exposant [eE][+-]?entier reel entier("."entier)?exposant? blancs / ignoré */ reel yylval=atof(yytext) ; return(NOMBRE) ; "+" return(PLUS) ;*
"-" return(MOINS) ;
"" return(FOIS) ;*
"" return(DIVISE) ; %% int yywrap() return 1 ;

4 Création de la table des symboles

5 Conclusion

Chapitre 4

Analyse syntaxique

1 introduction

L'analyse syntaxique a pour rôle de vérifier si la suite des entités lexicales, fournie par l'analyseur lexical, respecte un ordre précis de succession décrit par la syntaxe du langage. Pour se faire, la description de la syntaxe est matérialisée par une grammaire syntaxique permettant d'engendrer toutes les constructions possibles du langage (voir Figure 4.17). D'ailleurs, l'analyse syntaxique est l'étape la mieux formalisé du processus de compilation.

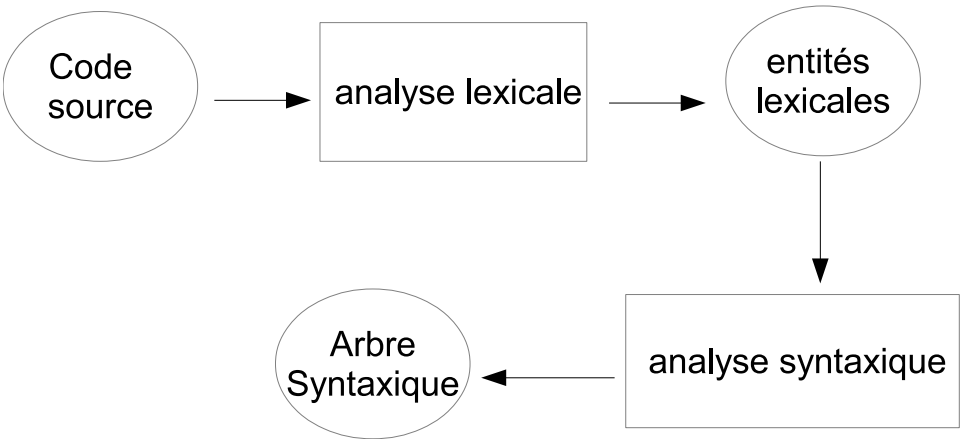


FIGURE 4.1 – Le processus de l'analyse syntaxique

2 Les concepts de base

Afin d'entamer les différentes étapes de la phase de l'analyse syntaxique, il est nécessaire de définir un certain nombre de concepts.

définition 2.1 (Grammaire syntaxique)
Une grammaire syntaxique G est formée de règles de production permettant d'engendrer tous les programmes écrits dans un langage donné. Elle est définie par un quadruplet <T,ℕ,S,P> tels que :

- *T représente un ensemble non vide des symboles terminaux de G,*
- *ℕ représente un ensemble non vide des symboles non terminaux,*
- *S est l'axiome,*
- *P est constitué par l'ensemble des règles de production du type $A \rightarrow \alpha$ avec $A \in \mathbb{N}$ et $\alpha \in (T \cup \mathbb{N})^*$. A représente le membre gauche de la production (MGP) et α représente le membre droit de la production (MDP).*
  Une règle de production précise que A peut être remplacé par la séquence de symboles α.

Exemple 1 *Soit la grammaire G définie par <T,ℕ,PROGRAMME, P> tels que T={+ ID * ( )}, ℕ={<programme>, <expression>, <terme>, <fact>} et des expressions arithmétiques suivante :*

$$P : \begin{cases} < programme > \rightarrow < expression > \\ < expression > \rightarrow < terme > | < expression > + < terme > \\ < terme > \rightarrow < fact > | < term > * < fact > \\ < fact > \rightarrow ID \mid (< expression >) \end{cases}$$

Cette grammaire permet de décrire les expressions arithmétiques.

définition 2.2 (Notion de dérivation)
La dérivation est une opération qui consiste, en utilisant une règle de production du type $A \rightarrow \alpha$, à remplacer le membre gauche A par le membre droit α.

définition 2.3 (Notion de réduction)
L'opération réduction est l'opération inverse de la dérivation. Elle consiste à remplacer, une fois que l'on reconnaît un membre droit de production, par son membre gauche.

définition 2.4 (Notion d'arbre syntaxique)
Un arbre syntaxique associé à une chaîne d'entités lexicales est un arbre dont la racine est l'axiome de la grammaire syntaxique, les nœuds représentent les membres

droits de production et les feuilles correspondent à la chaîne à analyser syntaxiquement.

Exemple 2 *L'arbre syntaxique de l'expression $id + id * id$ générée à partir de la grammaire G de l'exemple 1 est représenté par la figure 7.4.*

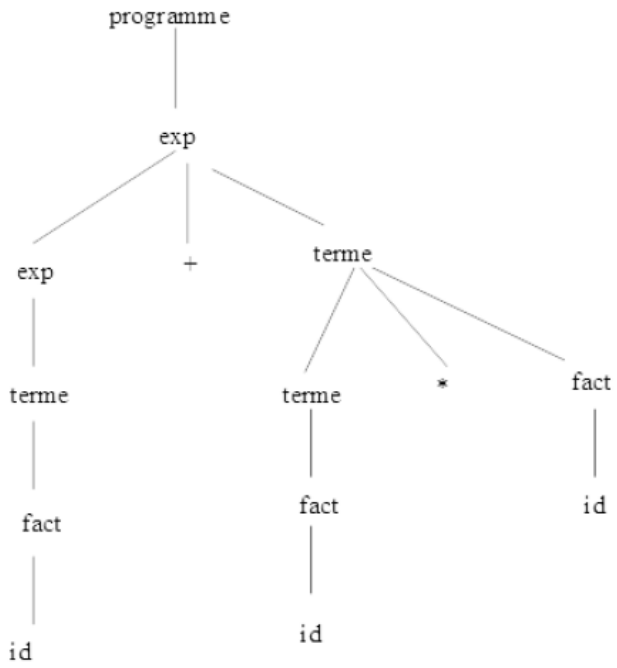


FIGURE 4.2 – Arbre syntaxique associé à la chaîne $id + id * id$

Deux méthodes sont utilisées pour la construction d'un arbre syntaxique :

1. les méthodes descendantes : en partant de l'axiome, une série de dérivations sont effectuées, jusqu'à atteindre la chaîne à analyser.

   Exemple 3 *La figure 3 illustre l'arbre syntaxique associé à la chaîne $id + id * id$ générée par la grammaire de l'exemple 1 en appliquant une méthode descendante.*

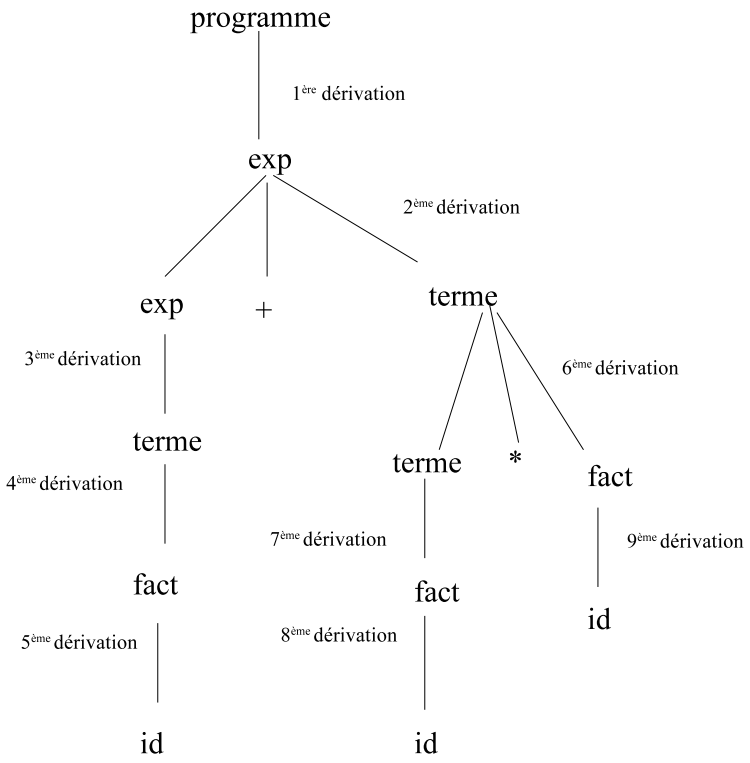


FIGURE 4.3 – Arbre syntaxique associé à la chaîne $id + id * id$ généré d'une manière descendante

2. les méthodes ascendantes : à partir de la chaîne à analyser, une série de réductions sont effectuées, jusqu'à arriver à l'axiome.

Exemple 4 *La figure 4 illustre l'arbre syntaxique associé à la chaîne* $id + id * id$ *générée par la grammaire de l'exemple 1 en appliquant une méthode ascendante.*

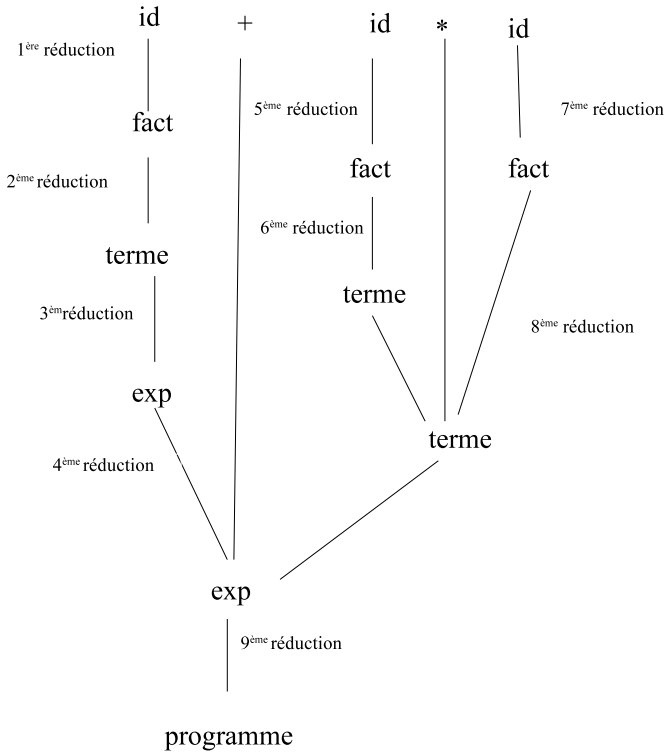


FIGURE 4.4 – Arbre syntaxique associé à la chaîne $id + id * id$ généré d'une manière ascendante

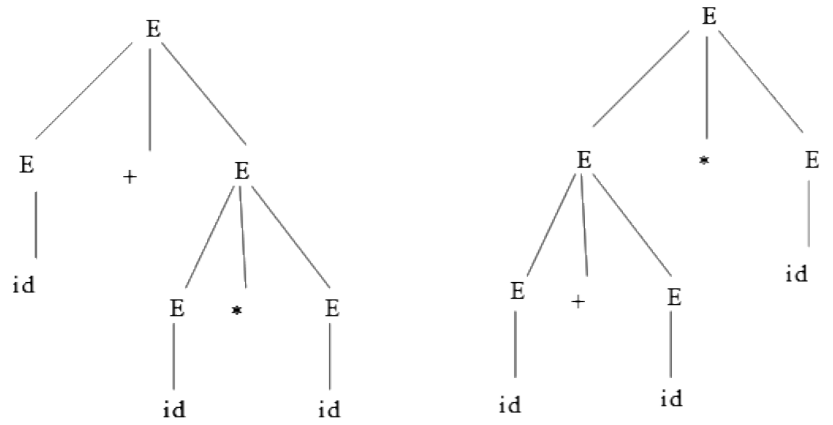
Remarque 1 *Un arbre qui a deux fils est dit binaire.*

définition 2.5 (Notion d'ambiguïté) *Une grammaire est dite ambiguë, s'il existe au moins deux arbres syntaxiques correspondants à une même chaîne.*

Exemple 5 *Soit G la grammaire définie <T,ℕ,E,P> tels que T={+ * ( ) id} ; ℕ={E} et*

$$P : \left\{ E \rightarrow E + E \mid E * E \mid (E) \mid id \right.$$

La grammaire G est ambiguë car à titre d'exemple, deux arbres syntaxiques sont associés à la chaîne $id + id * id$ comme l'illustre la Figure 5. La figure 5 illustre l'arbre syntaxique associé à la chaîne $id + id * id$ générée par la grammaire de l'exemple 1 en appliquant une méthode descendante.



Arbre 1 Arbre 2

FIGURE 4.5 – Arbres syntaxiques associés à la chaîne $id + id * id$

définition 2.6 (Récursivité gauche directe) *Une grammaire est dite récursive à gauche directe si elle possède un non terminal $A \in \mathbb{N}$ tel que $A \rightarrow A\ \alpha \mid \beta$ avec $\alpha \in N$ et $\alpha, \beta \in (T \cup \mathbb{N})^*$.*

définition 2.7 (Récursivité gauche indirecte) *Une grammaire est récursive indirectement si elle possède un non terminal $A \in \mathbb{N}$ tel que $A \xrightarrow{*} A\alpha$*

2.1 Élimination de la récursivité gauche dans une grammaire

Une analyse descendante ne peut se faire sur une grammaire récursive gauche directe ou indirecte. Si dans une grammaire, il existe une production du type A $\rightarrow$ A α avec $\alpha \in (T \cup N)^+$, l'identification de la chaîne à analyser avec un des membres droits de la production sera bloquée (boucle infinie). Il est ainsi nécessaire d'éliminer les récursivités gauches directes et indirectes dans une grammaire.

2.1.1 Élimination de la récursivité gauche directe

L'élimination de la récursivité gauche directe revient à transformer les règles de production du non terminal A du type :

$$\mathrm{A} \rightarrow \mathrm{A}\alpha_1 \mid ... \mid A\alpha_n \mid \beta_1 \mid ... \mid \beta_m \text{ avec } \mathrm{A} \in \mathbb{N} \text{ et } \beta_i, \alpha_i \in (T \cup N)^* \Leftrightarrow$$

$$P' : \begin{cases} A \rightarrow \beta_1 A' \mid ... \mid \beta_m A' \mid \beta_1 \mid ... \mid \beta_m \\ A' \rightarrow \alpha A' \mid ... \mid \alpha_n A' \mid \alpha_1 \mid ... \mid \alpha_n \end{cases}$$

ou encore

$$P' : \begin{cases} A \rightarrow \beta_1 A' \mid ... \mid \beta_m A' \\ A' \rightarrow \alpha_1 A' \mid ... \mid \alpha_n A' \mid \epsilon \end{cases}$$

Exemple 6 *Soit G la grammaire définie <T,ℕ,S,P> tels que T={a,b,c} ; ℕ={S,A} et*

$$P : \begin{cases} S \rightarrow Sa \mid SAb \mid Ac \mid a \\ A \rightarrow aA \mid c \end{cases}$$

G est récursive gauche directe en S. L'élimination directe de la récursivité directe en S transformera la grammaire G en une grammaire équivalente G' définie par <T, ℕ',S,P'> tels que : T={a,b,c} ; ℕ'={S,S',A} et

$$P' : \begin{cases} S \rightarrow AcS' \mid aS' \mid bSS' \mid Ac \mid a \mid bS \\ S' \rightarrow aS' \mid AbS' \mid a \mid Ab \\ A \rightarrow aA \mid c \end{cases}$$

ou encore

$$P' : \begin{cases} S \rightarrow AcS' \mid aS' \mid bSS' \\ S' \rightarrow aS' \mid AbS' \mid \epsilon \\ A \rightarrow aA \mid c \end{cases}$$

2.1.2 Élimination de la récursivité gauche indirecte

Afin d'éliminer le cas de la récursivité gauche indirecte d'un non terminal A, il faut effectuer des substitutions de telle manière à faire apparaître une récursivité gauche directe, et appliquer les règles de transformation précédentes. Ces étapes sont décrites par l'algorithme suivant :

début
  Ordonner les non terminaux $A_1 < A_2 < ... < A_n$ tels que
  $A_i < A_j$ si $A_i \rightarrow A_j\alpha$ avec $A_i, A_j \in \mathbb{N}$ et $\alpha \in (T \cup \mathbb{N})^*$
  pour *i :=1 to n* **faire**
    pour *j :=1 to i-1* **faire**
      Remplacer chaque production de la forme $A_i \rightarrow A_j\alpha \mid \beta$
      par les productions :
      $A_i \rightarrow \gamma_1\alpha \mid \gamma_2\alpha... \mid \gamma_k\alpha \mid \beta$ où $A_j \rightarrow \gamma_1 \mid \gamma_2... \mid \gamma_k$
    fin
    Éliminer la récursivité gauche directe en A_i ;
  fin
fin

Algorithme 5 : Élimination de la récursivité gauche indirecte

Exemple 7 *Soit G la grammaire définie <T,ℕ,S,P> tels que T={1,0} ; ℕ={S,A,B} et*

$$P : \begin{cases} S \rightarrow SA10 \mid AB \\ A \rightarrow B1 \mid 0 \\ B \rightarrow S0 \mid 1 \end{cases}$$

1. *Ordonnancement des non terminaux : S < A < B < S et S < S. La grammaire G présente une récursivité gauche indirecte en S et une récursivité gauche directe en S.*

2. *Substitutions successives :*

   — *Substitution de B dans A :*
   $A \rightarrow S01 \mid 11 \mid 0$

   — *Substitution de A dans S :*
   $S \rightarrow A10 \mid S01B \mid 11B \mid 0B$

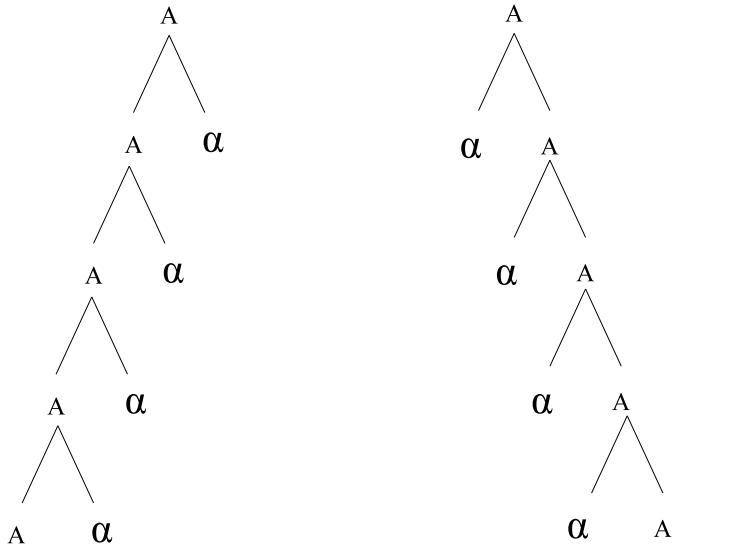
3. *Élimination de la récursivité gauche directe en S :*
   $S \rightarrow 11BS' \mid 0BS'$
   $S' \rightarrow A10S' \mid 01BS' \mid \epsilon$

Ainsi, la grammaire G est transformée en une grammaire équivalente G' définie

par <T,ℕ′, S, P'> tels que : T={1,0} ; ℕ′={S,S',A,B} et

$$P' : \begin{cases} S \rightarrow 11BS' \mid 0BS' \\ S' \rightarrow A10S' \mid 01BS' \mid \epsilon \\ A \rightarrow B1 \mid 0 \\ B \rightarrow S0 \mid 1 \end{cases}$$

Remarque 2 *Si la grammaire comporte une règle de la forme $A \rightarrow \alpha A$ avec $\alpha \in (T \cup N)^+$, elle est dite récursive droite.*
La récursivité droite n'entrave pas le déroulement d'un analyseur descendant car la partie récursive (α) est générée en premier lieu contrairement au cas de la récursivité gauche (voir Figure 2).



Récursivité gauche Récursivité droite

FIGURE 4.6 – Les cas de récursivité gauche et droite

2.2 Grammaires ϵ libres et sans cycles

L'algorithme d'élimination de la récursivité gauche s'applique aux grammaires ϵ libres et sans cycles.

2.2.1 Grammaires ϵ libres

définition 2.8 *Une grammaire G définie par <T,ℕ,S,P> est dite ϵ libre :*

— *Si aucune règle de production ne contient ϵ,*

— *Si l'axiome S → ϵ, alors S ne doit pas apparaître dans aucun membre droit des règles de production de P.*

Exemple 8 *Soit G la grammaire définie par <T,ℕ,S,P> où T={a,b} ; ℕ={S,A,B} et*

$$P : \begin{cases} S \rightarrow Ab \mid Bb \mid \epsilon \\ A \rightarrow aB \mid bSB \\ B \rightarrow AB \mid bAS \end{cases}$$

La grammaire G n'est pas ϵ libre car S→ ϵ et S apparaît dans deux membres droits de deux règles de production.

L'algorithme de transformation d'une grammaire G, définie par <𝕋,ℕ,S,P>, non ϵ libre en une grammaire ϵ libre est comme suit :

début
L ← ∅ ;
pour *chaque* $A \in \mathbb{N}$ **faire**
si $(A \rightarrow \epsilon)$ **alors**
Insérer A dans L ;
fin
fin
tant que $(L \neq \emptyset)$ **faire**
Retirer le non terminal A de L ;
pour *Chaque règle qui fait apparaître A dans un de ses membres droits* **faire**
Soit $B \rightarrow \alpha A \beta A \gamma ... A \delta$;
Rajouter de nouveaux membres droits à B en remplaçant toutes les occurrences de A par ϵ :
$B \rightarrow \alpha \beta A \gamma ... A \delta$
$B \rightarrow \alpha A \beta \gamma ... A \delta$
$B \rightarrow \alpha A \beta A \gamma ... \delta$
$B \rightarrow \alpha A \beta \gamma ... \delta$
$B \rightarrow \alpha \beta A \gamma ... \delta$
$B \rightarrow \alpha \beta \gamma ... A \delta$
$B \rightarrow \alpha \beta \gamma ... \delta$
si $(B \rightarrow B)$ **alors**
Supprimer la règle B → B ;
sinon
si *(en remplaçant toutes les occurrences de A par* ϵ*,* $B \rightarrow \epsilon$*)* **alors**
Insérer B dans L ;
si $(A \neq S)$ **alors**
Supprimer la règle A → ϵ de P ;
fin
sinon
fin
si *(S apparaît dans un des membres droits des règles de production de P)* **alors**
Rajouter S' à $\mathbb{N}$;
rajouter les règles suivantes à P :
S' → *S#* | ϵ
Remplacer l'axiome S par S' ;
fin
fin
fin
fin
fin
fin
fin

Algorithme 6 : Transformation d'une grammaire en $\epsilon - libre$

Exemple 9 *Soit G la grammaire définie par <𝕋,ℕ,S,P> où 𝕋={a,b} ; ℕ={S,A,B} et*

$$P : \begin{cases} S \rightarrow Ab \mid Bb \mid \epsilon \\ A \rightarrow aB \mid bSB\epsilon \\ B \rightarrow AB \mid bS \end{cases}$$

La grammaire G n'est pas ϵ libre car S→ ϵ et S apparaît dans deux membres droits de deux règles de production. De plus, A → ϵ. La grammaire ϵ libre G' équivalente à G est obtenue en appliquant l'algorithme 6. Elle est définie par <𝕋,ℕ,S,P'> tels que : 𝕋={a,b} ; ℕ={S,A,B}

$$P' : \begin{cases} S \rightarrow Ab \mid Bb \mid b \\ A \rightarrow aB \mid bSB \mid bB \\ B \rightarrow AB \mid bS \mid B \mid b \end{cases}$$

Comme la règle de production B →B est inutile, elle sera supprimée de l'ensemble des règles de production. Ainsi :

$$P' : \begin{cases} S \rightarrow Ab \mid Bb \mid b \\ A \rightarrow aB \mid bSB \mid bB \\ B \rightarrow AB \mid bS \mid b \end{cases}$$

2.2.2 Grammaire sans cycle

définition 2.9 *Une grammaire G définie par <𝕋,ℕ,S,P> est dite sans cycle si elle n'admet aucune dérivation de la forme $A \xrightarrow{*} A$.*

2.3 Factorisation d'une grammaire

Une des conditions permettant de faire une analyse syntaxique descendante déterministe est la factorisation.
Une grammaire G est dite factorisée si tous les membres droits des productions associés à l'ensemble des non terminaux de la grammaire qui débutent par des terminaux, ces derniers doivent être distincts. Dans le cas contraire, il faudra transformer les règles de la manière suivante :
Des règle du type A $\rightarrow a\alpha \mid a\beta \mid a\gamma$ avec a $\in \mathbb{T}$
deviennent :

$$\begin{cases} A \rightarrow aA' \\ A' \rightarrow \alpha \mid \beta \mid \gamma \end{cases}$$

L'algorithme suivant décrit le procédure de la factorisation.

début
  pour *chaque non terminal* $A \in \mathbb{N}$ **faire**
    Trouver le plus long préfixe $\alpha \in (\mathbb{T} \cup \mathbb{N})^*$ commun à au moins deux productions affiliées à A ;
    si $(\alpha \neq \epsilon)$ **alors**
      remplacer A $\rightarrow \alpha\beta_1 \mid ...\alpha\beta_n \mid \gamma_1 \mid ... \mid \gamma_m$ avec $\beta_i, \gamma_j \in (\mathbb{T} \cup \mathbb{N})^*$ par
      A $\rightarrow \alpha A' \mid \gamma_1 \mid ... \mid \gamma_m$
      A' $\rightarrow \beta_1 \mid ... \mid \beta_n$
    fin
  fin
fin

Algorithme 7 : Factorisation d'une grammaire

Remarque 3 *La factorisation assure que lors de l'analyse syntaxique, lorsque le sommet de pile est un non terminal A et que l'entité courante est a, un seul membre droit pourra être appliqué pour reconnaître le terminal. La factorisation contribue à garantir le déterminisme de la méthode d'analyse.*

Exemple 10 *Soit G la grammaire définie par <T,ℕ,S,P> où T={a,b} ; ℕ={S,A} et*

$$P : \begin{cases} S \rightarrow aS \mid abAS \mid Ab \\ A \rightarrow bA \mid bS \mid b \end{cases}$$

La factorisation va transformer la grammaire G en une grammaire G' équivalente définie par <T,ℕ′,S,P'> où T={a,b} ; ℕ′={S,S',A,A'} et

$$P' : \begin{cases} S \rightarrow aA' \mid Ab \\ S' \rightarrow S \mid bAS \\ A \rightarrow bA' \\ A' \rightarrow A \mid S \mid \epsilon \end{cases}$$

D'une manière générale, il est préférable de trouver le plus long préfixe qui est en commun entre les membres droits affiliés à un non terminal A pour la factorisation.

Exemple 11 *Soient les règles de production relatives à l'instruction conditionnelle :*
<instif> → IF <condition> THEN <instruction> ELSE <instruction> | IF <condition> THEN <instruction>

La factorisation va ainsi transformer ces règles comme suit :
<instif> → IF <condition> THEN <instruction> <suite>
<suite> → ELSE <instruction> | ϵ
Le fait d'avoir factoriser les quatre premiers symboles des deux règles va permettre de rendre le processus d'analyse déterministe dans le cas de l'instruction conditionnelle.

2.4 Les formes normales des grammaires

2.4.1 La forme Normale de Chomsky

définition 2.10 (Grammaires sous forme normale de Chomsky (FNC)) *Une grammaire est sous forme normale de Chomsky, si chaque production de G est de la forme :*

- *$A \rightarrow B\ C$ avec $A, B, C \in N$,*
- *$A \rightarrow a$ avec $a \in T$,*
- *G est ϵ-libre : Si $\epsilon \in L(G)$, alors seul $S \rightarrow \epsilon$ et S ne doit pas apparaître comme membre droit d'une production.*

Lemme 1 *A toute grammaire de type 2, une grammaire G' équivalente, sous FNC peut lui être associée.*

Preuve 1 *Soit G = <T, N, S, P> une grammaire de type 2. La grammaire G' = <T', N', S', P'> équivalente sous la forme FNC est définie par l'algorithme 8.*

début
- T' := T ;
- N' := N ∪{ensemble de tous les non-terminaux rajoutés afin d'obtenir une FNC} ;
- S' := S ;
- P' est tel que :
- **si** *($A \rightarrow a \in P$ où $A \in N$ et $a \in T$)* **alors**
  - A → a ∈ P' ;
- **fin**
- **si** *($A \rightarrow B\ C \in P$ où $A,B \in N$)* **alors**
  - A → B C ∈ P' ;
- **fin**
- **si** *($S \rightarrow \epsilon \in P$)* **alors**
  - S → ϵ ∈ P' ;
- **fin**
- **si** *($A \rightarrow X_1X_2...X_n$, avec $X_i \in (T \cup N)^*$ et $n \geq 2$)* **alors**
  - ajouter dans P' les productions suivantes :
  - A → $X_1 < X_2.....X_n >$;
  - $< X_2.....X_n > \rightarrow X_2 < X_3.....X_n >$;
  - réitérer le procédé jusqu'à la fin ;
- **fin**
- **si** *($A \rightarrow X_1X_2$)* **alors**
  - **si** *($X_1, X_2 \in N$)* **alors**
    - A → $X_1X_2 \in P'$;
  - **fin**
  - **sinon**
    - **si** *($X_1X_2 \in T$)* **alors**
      - A → Y Z avec Y, Z ∈ N, Y → X_1, Z → X_2
    - **fin**
    - **sinon**
      - **si** *(l'un des deux est un terminal)* **alors**
        - introduire un nouveau non terminal se dérivant en ce terminal ;
      - **fin**
    - **fin**
  - **fin**
- **fin**

fin

Algorithme 8 : Transformation d'une grammaire sous la forme FNC

L'avantage d'une grammaire sous forme normale de Chomsky est que son arbre

syntaxique est binaire ce qui induit que son parcours devient rapide et l'analyse est alors accélérée.

Exemple 12 *Soit G la grammaire définie par <T,ℕ,S,P> où T={a,c,d} ; ℕ={S,A,B} et*

$$P : \begin{cases} S \rightarrow a \mid A \\ A \rightarrow BA \mid c \mid \epsilon \\ B \rightarrow dB \end{cases}$$

G n'est pas sous la forme FNC car elle n'est pas ϵ libre. La transformation de la grammaire sous FNC nécessite plusieurs étape :

1. *rendre G $\epsilon - libre$:*

$$\begin{cases} S \rightarrow a \mid A \mid \epsilon \\ A \rightarrow BA \mid c \mid B \\ B \rightarrow dB \end{cases}$$

2. *remplacer dans S le non terminal A par ses membres droits :*

$$\begin{cases} S \rightarrow a \mid BA \mid c \mid B \mid \epsilon \\ A \rightarrow BA \mid c \mid B \\ B \rightarrow dB \end{cases}$$

3. *remplacer dans S, A et B le terminal d par un non terminal C :*

$$\begin{cases} S \rightarrow a \mid BA \mid c \mid dB \mid \epsilon \\ A \rightarrow BA \mid c \mid dB \\ B \rightarrow dB \end{cases}$$

4. *remplacer B par ses membres droits dans S et A :*

$$\begin{cases} S \rightarrow a \mid BA \mid c \mid CB \mid \epsilon \\ A \rightarrow BA \mid c \mid CB \\ B \rightarrow CBC \rightarrow d \end{cases}$$

Ainsi, la grammaire G' mise sous forme Normale de Chomsky équivalente à G est définie par <𝕋,ℕ′,S,P'> tels que T={a,c,d} ; ℕ′={S,A,B,C} et P' :

$$\begin{cases} S \rightarrow a \mid BA \mid c \mid CB \mid \epsilon \\ A \rightarrow BA \mid c \mid CB \\ B \rightarrow CBC \rightarrow d \end{cases}$$

2.4.2 La forme Normale de Greibach

Définition 4.1 (Grammaires sous forme normale de Greibach (FNG)) *Une grammaire G est sous forme normale de Greibach, si G est ϵ libre de type 2, et chaque production est de la forme $A \rightarrow a\alpha$ avec $A \in N$, $a \in T, \alpha \in N^*$*

L'algorithme 9 dresse les étapes de transformation d'une grammaire G sous la forme normale de Greibach.

début
  si *(G n'est pas ϵ-libre)* **alors**
    rendre G ϵ-libre en appliquant l'algorithme6
  fin
  si *(G est récursive gauche directe)* **alors**
    supprimer la récursivité gauche directe
  fin
  si *(G est récursive gauche indirecte* **alors**
    Supprimer la récursivité gauche indirecte en appliquant l'algorithme 5
  fin
  Définir un ordre partiel entre les non terminaux :
  (A <B, si A→ B α avec A,B ∈ N et $\alpha \in (T \cup N)^*$).
  En partant de la dernière production du plus grand non terminal ordonné vers le plus petit,
  répéter
    remplacer chaque production de la forme :
    $A_i \rightarrow A_j\alpha$ avec
    $A_j \rightarrow \beta_1 \mid \beta_2 \cdots \mid \beta_n$ avec $\beta_i \in (T \cup N)^*$ par
    $A_i \rightarrow \beta_1\alpha \mid \beta_2\alpha \cdots \mid \beta_n\alpha$;
  jusqu'à *ce que chaque $\beta_{i,i\in[1,n]}$ commence par un terminal*;
fin

Algorithme 9 : Transformation d'une grammaire sous la forme FNG

L'avantage de la forme Greibach est que chaque membre droit des productions commence par un terminal. Ainsi, l'analyse est accélérée, vu que le terme courant est comparé directement au terminal du membre droit de production.

Exemple 4.1 *Soit à transformer la grammaire G sous FNG telle que G est définie par <T,N,A,P> où T={a,,b,c} ; N={A,B,C} ; et P :*

$$\begin{cases} A \rightarrow AB \mid a \\ B \rightarrow CA \mid b \\ C \rightarrow c \end{cases}$$

1. *G est ϵ-libre ;*
2. *G est récursive directe en A. Après élimination de la récursivité directe en A, la grammaire G est transformée en une grammaire G' équivalente définie par <T,N',A,P'> où T={a,b,c} ; N'={A,A',B,C} ; et P' :*

$$\begin{cases} A \rightarrow aA' \mid a \\ A' \rightarrow BA' \mid B \\ B \rightarrow CA \mid b \\ C \rightarrow c \end{cases}$$

3. *Établissement d'un ordre partiel entre les non terminaux : A' < B < C (car A' appelle B et B appelle C)*
4. *Remplacement de C par ses membres droits dans B*

$$\begin{cases} A \rightarrow aA' \mid a \\ A' \rightarrow BA' \mid B \\ B \rightarrow cA \mid b \\ C \rightarrow c \end{cases}$$

5. *Remplacement de B par ses membres droits dans A'*

$$\begin{cases} A \rightarrow aA' \mid a \\ A' \rightarrow cAA' \mid bA' \mid cA \mid b \\ B \rightarrow cA \mid b \\ C \rightarrow c \end{cases}$$

Le but de l'analyseur syntaxique est de construire ou d'associer un arbre syntaxique à la suite d'entités lexicales produite par l'analyseur lexicale. Si le processus de construction de l'arbre réussi alors la chaîne est acceptée du point de vue syntaxique. Dans le cas contraire, la chaîne est erronée et l'analyseur va générer des erreurs syntaxiques.
La méthode de construction de l'arbre syntaxique induit l'existence de deux type de méthodes d'analyse :

1. les méthodes descendantes, qui consistent, à partir de l'axiome à dériver jusqu'à l'obtention de la chaîne à analyser en commençant par le non terminal le plus à gauche,
2. les méthodes ascendante qui consistent à partir de la chaîne à opérer un certain nombre de réductions jusqu'à aboutir à l'axiome.

3 Les méthodes descendantes

Les méthodes d'analyse syntaxiques descendantes ont pour rôle de générer un arbre syntaxique à partir de l'axiome jusqu'à ce que les feuilles de l'arbre correspondent à la chaîne à analyser. Pour cela, deux familles d'analyseurs ont été développées : les méthodes déterministes et les méthodes non déterministes. Ces derniers ne sont pas implémentées car elles ne sont pas efficaces.

3.1 Les méthodes d'analyse syntaxiques descendantes non déterministes

3.1.1 Analyse descendante parallèle

L'opération consiste à remplacer un non-terminal par un ou plusieurs membres droits de production. Contrairement à un analyseur déterministe, toutes les alternatives sont abordées. Les remplacements sont retenus, en parallèle, jusqu'à possibilité d'abandon. Les étapes de la méthode sont résumées par l'algorithme 10.

```
BEGIN
cible ← S# ;
if (tête-cible = non-terminal) then
   remplacer le non-terminal par
   tous ses membres droits des productions,
   jusqu'à obtention d'un terminal
end if
comparer le terme courant avec la tête de la cible ;
if (tête de la cible = terme courant de la chaine) then
   avancer() ;
   dépiler() ;
else
   échec() ;
end if
if (tête-cible = #)  (terme courant=#) then
   (écrire "La chaîne est correcte syntaxiquement") ;
   abandonner les autres cibles ;
else
   échec() ;
end if
END
```

Algorithme 10 : Algorithme de la descente parallèle

Exemple 13 *Soit la grammaire G définie par <T,N,Bloc,P> où T={debut ; d i} ; N={BLOC, LD, LI} ; et P :*

$$\begin{cases} BLOC \rightarrow debut LD ; LI fin \\ LD \rightarrow d \mid d ; LD \\ LI \rightarrow i \mid i ; LI \end{cases}$$

L'analyse de la chaîne "debut d ;i fin #" selon la méthode de la descente parallèle se déroule comme suit :

TABLE 4.1 – Analyse de la chaîne *debut d ;i fin #*

Pas	Cible	Chaîne	Action
1	BLOC #	debut d ;i fin #	remplacer BLOC par debut LD ;LI fin
2	**debut** LD ;LI fin #	**debut** d ;i fin #	égalité : dépiler() ; avancer() ;
3	LD ;LI fin #	d ;i fin #	remplacer LD par ses MDP
4	**d** ; LI fin #	**d** ;i fin #	égalité : dépiler() ; avancer() ;
	d ;LD LI fin #	**d** ;i fin #	égalité : dépiler() ; avancer() ;
5	**;** LI fin #	**;** i fin #	égalité : dépiler() ; avancer() ;
	; LD LI fin #	**;** i fin #	égalité : dépiler() ; avancer() ;
6	LI fin #	i fin #	remplacer LI par ses deux MDP
	LD LI fin #	i fin #	remplacer LD par ses deux MDP
7	**i** fin #	**i** fin #	égalité : dépiler() ; avancer() ;
	i ; LI fin #	**i** fin #	égalité : dépiler() ; avancer() ;
	d LI fin #	i fin #	blocage() ;
	d ; LD LI fin #	i fin #	blocage() ;
8	**fin** #	**fin** #	égalité : dépiler() ; avancer() ;
	; LI fin #	fin #	blocage() ;
9	#	#	La chaîne est correcte syntaxiquement ;

Au niveau de chaque étape, l'analyseur indique toutes les futures possibilités. Cette méthode n'est pas performante car elle nécessite l'examen de plusieurs arbres en parallèle ce qui couteux en temps et en espace.

3.1.2 La méthode avec Retour Arrière

Le processus de fonctionnement de la méthode avec retour arrière est séquentiel. Elle consiste à exploiter une alternative à fond. Si cette dernière aboutit à un échec alors l'algorithme effectue un retour arrière afin d'explorer une autre alternative. A chaque fois que l'algorithme entame un parcours, l'alternative est em-

pilée. Si le choix s'avère infructueux, le retour arrière s'effectue selon le contenu de la pile. Techniquement, les productions de la grammaire sont numérotées. L'algorithme explore les alternatives une à une jusqu'à aboutir à l'acceptation de la chaîne à analyser. Si aucune alternative ne fonctionne, alors la chaîne est erronée syntaxiquement.
Le fonctionnement de la méthode est décrit par l'algorithme 11.

```
début
    Pour un non terminal donné, les alternatives sont explorées dans un
      ordre préalablement défini
    si (terme courant= terminal droit de la production) alors
        dépiler() ;
        avancer() ;
    fin
    sinon
        pour le même non terminal, empiler l'alternative suivante ;
        si (aucune alternative ne fonctionne) alors
            revenir au sous arbre précédent ;
        fin
    fin
fin
```

Algorithme 11 : Algorithme de la méthode avec retour arrière

Remarque 4 *Dans cette méthode, il est nécessaire de mémoriser les différentes alternatives d'un non terminal donné ainsi que le terme en cours de traitement. L'algorithme n'est performant car en pratique, il est indispensable d'effectuer plusieurs retours arrières. De ce fait, il faut associer à chaque règle les informations suivantes : le numéro de la règle, le numéro de l'alternative, le nombre des alternatives, la position dans la chaîne.*

Exemple 14 *L'analyse de la chaîne "debut d ;i fin #" selon la méthode avec retour arrière se déroule comme suit :*

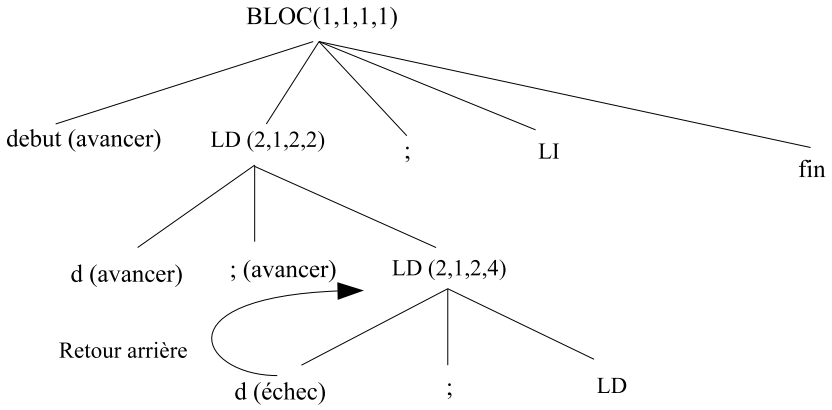


FIGURE 4.7 – Méthode avec retour arrière : étape 1

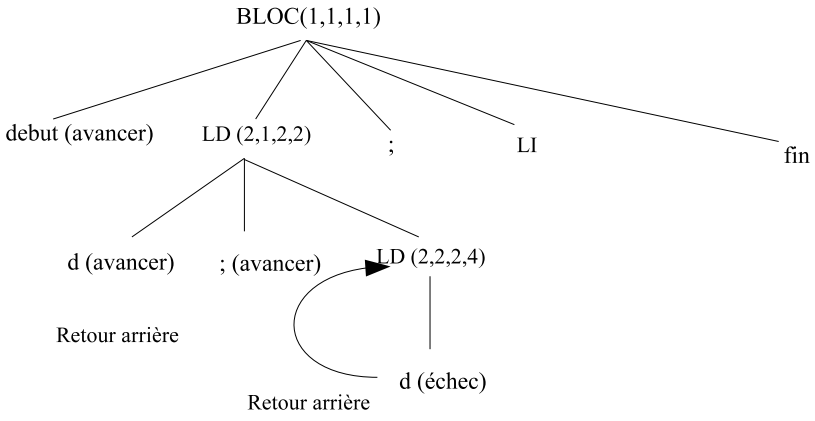


FIGURE 4.8 – Méthode avec retour arrière : étape 2

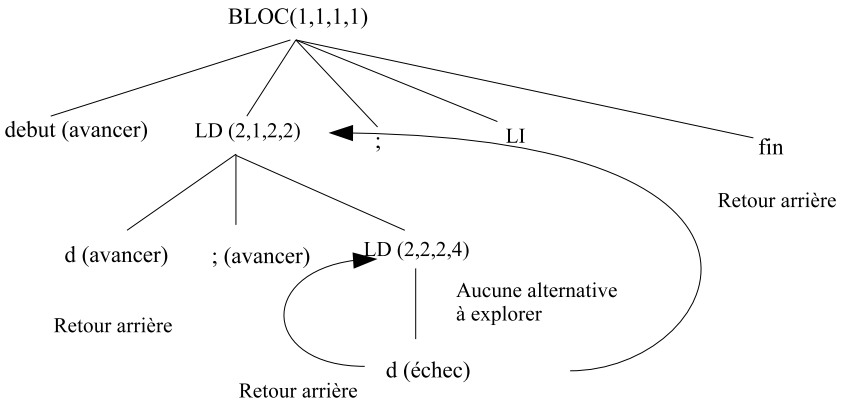


FIGURE 4.9 – Méthode avec retour arrière : étape 3

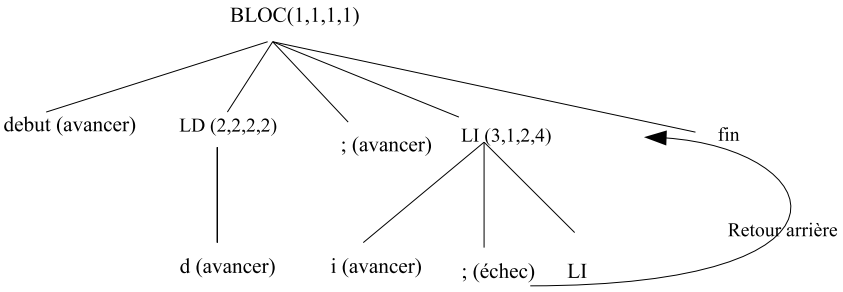


FIGURE 4.10 – Méthode avec retour arrière : étape 4

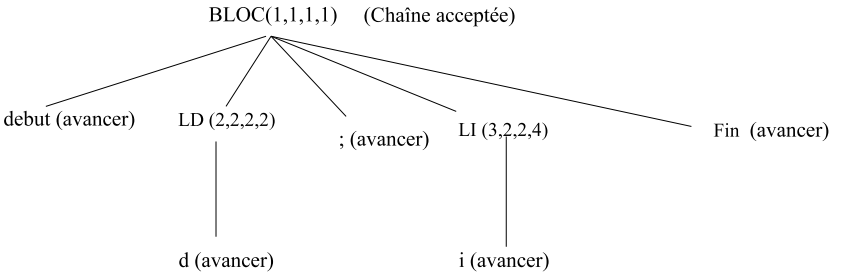


FIGURE 4.11 – Méthode avec retour arrière : étape 5

3.2 Les méthodes d'analyse déterministes

Contrairement aux méthodes non déterministes, les méthodes déterministes sont sans retour arrière. A chaque étape, seule l'alternative susceptible de correspondre est sélectionnée.

4 Analyse descendante LL(1)

La méthode d'analyse LL(1) est une analyse prédictive. A partir d'une grammaire et d'une chaine en entrée, il s'agit de déterminer les productions à appliquer en partant de l'axiome afin d'arriver à la chaine. En effet, il est possible de construire un analyseur prédictif, non récursif en utilisant une pile de façon explicite.

Définition 4.2 (Définition informelle d'une grammaire LL(1)) *Une grammaire est dite LL(1), si en consultant une seule entité du programme à analyser, la règle à appliquer est sélectionnée.*

La signification de l'acronyme LL(1) est la suivante :
L : Left to right scanning (parcours de gauche à droite)
L : Left most derivation (dérivation à gauche)
(1) : désigne le nombre de symboles d'entrée en prévision utilisés à chaque étape afin de décider de l'action à exécuter.
Le fonctionnement de l'analyseur LL(1) est illustré par la Figure 4.

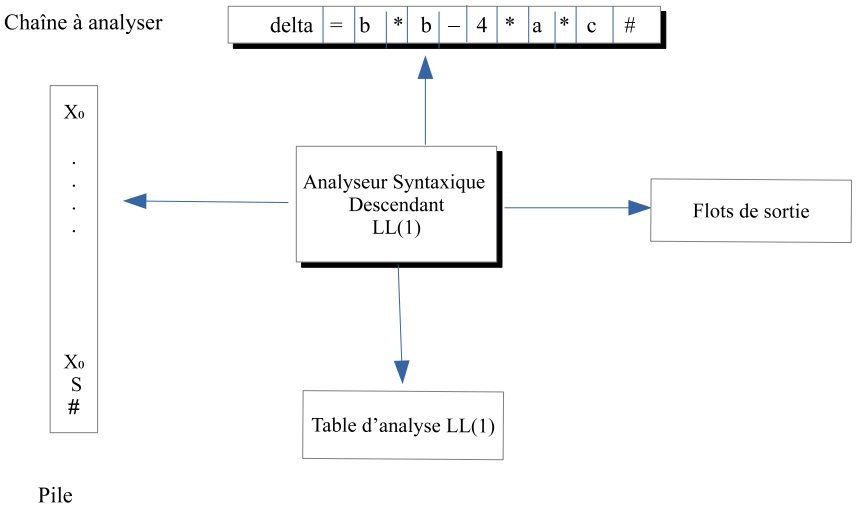


FIGURE 4.12 – Schéma de fonctionnement d'un analyseur LL(1)

L'analyseur dispose :

— de la chaîne d'entrée à analyser qui est composée par une suite d'entités lexicales suivie par le symbole fin de chaîne #,

— de la pile d'analyse qui contient les symboles de la grammaire et du symbole fin de chaîne #. Initialement, la pile contient # suivi de l'axiome de la grammaire,

— de la table d'analyse. C'est un tableau à deux dimensions telles que les ligne représentent les non terminaux de la grammaire et les colonnes représentent les terminaux et le symbole fin de chaîne. M(A,a), tels que A ∈ N et a ∈ T, indique la règle à appliquer lorsque le symbole de sommet de pile est **A** et l'entité courante de la chaîne est **a**.

Ainsi, l'analyseur traite la chaîne d'entrée entité par entité et décide à chaque étape de l'action à exécuter en fonction :

— du sommet de pile A,

— de l'entité courante a,

— du contenu de la case M(A,a).

Définition 4.3 *une Grammaire est LL(1) si :*

1. *G est non récursive gauche directe et G est non récursive gauche indirecte,*

2. *G est factorisée,*

3. *La table d'analyse est mono-définie.*

Remarque 5 *Afin de matérialiser l'arrêt de l'analyse syntaxique avec succès d'une grammaire G définie par <T,N,S,P>, cette dernière est augmentée d'un nouveau symbole non terminal Z, comme symbole de départ avec une règle de production supplémentaire Z → S# donnant ainsi naissance à une grammaire G' définie par <T,N,S,P'> tel que P' est obtenu à partir de P auquel la production Z → S# est rajouté.*

4.1 Construction de la table d'analyse LL(1)

La construction de la table d'analyse LL(1) associée à une grammaire G définie par <T,N,S,P> nécessite de définir l'ensemble des débuts et l'ensemble des suivants affiliés à chaque non terminal A ∈ N de la grammaire G.

4.1.1 Calcul des débuts

Cette étape consiste à définir l'ensemble des terminaux qui apparaissent directement ou indirectement au début des règles associées à A. Le calcul de l'ensemble des débuts d'un non terminal A se fait selon l'algorithme 12.

début
  si *($X \rightarrow x\beta$ tels que $X \in N$, $x \in T$ et $\beta \in (T \cup N)^*$)* **alors**
    x ∈ Débuts(X)
  fin
  si $X \rightarrow \epsilon$ **alors**
    $\epsilon \in$ Début(X)
  fin
  si *($X \rightarrow Y_1 ... Y_n$)* **alors**
    Débuts(Y_1)-{ϵ} ⊂ Débuts(X)
    si *($Y_1 \xrightarrow{*} \epsilon$)* **alors**
      Débuts(Y_2)-{ϵ } ⊂ Débuts(X)
    fin
    si *(pour tout $i < n$, $Y_i \xrightarrow{*} \epsilon$)* **alors**
      Débuts(Y_{i+1})-{ϵ } ⊂ Débuts(X)
      si *(pour tout $i \in [1, n]$, $Y_i \xrightarrow{*} \epsilon$)* **alors**
        $\epsilon \in$ Débuts(X)
      fin
    fin
  fin
fin

Algorithme 12 : Algorithme de calcul des débuts

4.1.2 Calcul des suivants

L'ensemble des suivants associé à un non terminal A ∈ N d'une grammaire G définie par <T,N,S,P>, est constitué des terminaux qui apparaissent directement ou indirectement à droite du non-terminal en MDP. Le calcul des suivants se fait selon l'algorithme 13.

début
  # ∈ Suivants(S) ;* car la règle Z → *S* # appartient à la grammaire augmentée associée à G *)
  Soit la règle A → αBβ A,B ∈ $\mathbb{N}$ α,β∈ $(T \cup \mathbb{N})^*$
  si *($\beta \in T$)* **alors**
    β ∈ Suivants(B)
  fin
  si *($\beta \in N$)* **alors**
    Débuts(β)-{ϵ} ⊂ Suivants(B)
  fin
  si *($\beta \xrightarrow{*} \epsilon$)* **alors**
    Suivants(A) ⊂ Suivants(B)
  fin
fin

Algorithme 13 : Algorithme de calcul des suivants

Exemple 15 *Soit la grammaire G définie par <T, N,E,P> où T={+ - * / ( ) nb}, N={E,E',T,T',F} et P :*

$$\begin{cases} Z \rightarrow E\# \\ E \rightarrow TE'(1) \\ E' \rightarrow +TE'(2) \mid -TE'(3) \mid \epsilon(4) \\ T \rightarrow FT'(5) \\ T' \rightarrow *FT'(6) \mid /FT'(7) \mid \epsilon(8) \\ F \rightarrow (E)(9) \mid nb(10) \end{cases}$$

Calcul des Débuts
Débuts(E) = Débuts(T) = Débuts(F) = { (, nb}
Débuts(E') = {+,-,ϵ }
Débuts(T) = Débuts(F) = { (, nb}
*Débuts(T') = { *, /, ϵ }*
Débuts(F) = { (, nb)}

Calcul des Suivants
Suivants(E) = { #,)}
suivant(E') = Suivants(E) = { #,)}
Suivants(T) = Débuts(E') - {ϵ } ∪ Suivants(E) = { +, -,), #}
Suivants(T') =Suivants(T)= { +, -,), #}
*Suivants(F) =Débuts(T') - {ϵ } ∪ Suivants(T)= { *, /, +, -,), #}.*

Les débuts et les suivants associés aux non terminaux de la grammaire G sont résumés par la table 15.

	Débuts	Suivants
E	(nb	#)
E'	+ - ϵ	#)
T	(nb	+ -) #
T'	* / ϵ	+ -) #
F	(nb	* / + -) #

TABLE 4.2 – Les débuts et les suivants de la grammaire G de l'Exemple 15

4.1.3 Construction de la table d'analyse LL(1)

Une table d'analyse, notée $\mathbb{M}$, est un tableau M à deux dimensions qui indique à chaque non-terminal A et chaque terminal a ou #, la règle de production à appliquer. La construction de la table d'analyse est décrite par l'algorithme 14.

début
  pour *chaque production* $A \rightarrow \alpha$ **faire**
    pour $a \in$ *débuts*(α) *et* $(a \neq \epsilon)$ **faire**
      rajouter la production $A \rightarrow \alpha$ dans la case M[A,a]
    fin
  fin
  si $(\epsilon \in$ *débuts*$(\alpha))$ **alors**
    pour *(chaque* $b \in$ *suivant(A))* **faire**
      ajouter la règle $A \rightarrow \epsilon$ dans M[A,b]
    fin
  fin
  Chaque case M[A,a] vide correspond à une erreur syntaxique
fin

Algorithme 14 : Algorithme de construction de la table d'analyse LL(1)

Exemple 16 *La table d'analyse qui correspond à la grammaire G de l'Exemple 15 est donnée par la Table 16.*

	nb	+	-	*	/	(	)	#
E	R_1					R_1		
E'		R_2	R_3				R_4	R_4
T	R_5					R_5		
T'		R_8	R_8	R_6	R_7		R_8	R_8
F	R_{10}					R_9		

TABLE 4.3 – Table LL(1) de la grammaire de l'exemple 16

La table d'analyse LL(1) est mono-définie $\Longrightarrow$ *G est LL(1).*

4.2 Algorithme d'analyse syntaxique LL(1)

Pour déterminer si une chaîne est dérivée d'une grammaire G définie par <T,$\mathbb{N}$,S,P> ($S \xrightarrow[?]{*} chaîne$), une pile est utilisée. En entrée, l'algorithme d'analyse reçoit une chaine d'entités lexicales ainsi qu'une table d'analyse $\mathbb{M}$. La chaine à analyser comporte un caractère de fin de chaine symbolisé par #. La pile contient les symboles de la grammaire $(T \cup \mathbb{N})^*$ et #. Initialement la pile contient # suivi de l'axiome S. Les étapes de l'analyse sont décrites par l'algorithme 15.

```
Données : Chaîne suivie de #
table d'analyse M
Initialisation :
Empiler (#);
Empiler (S);
PS := 1^ere entité de la chaîne;
début
    Soit X le symbol sommet de pile;
    Soit a l'entité pointée par PS;
    répéter
        si (X est un non-terminal) alors
            si M[X,a] = X → Y1 Y2 ... Yn alors
                dépiler(X);
                Empiler( Yn Yn-1 ... Y1);
                Écrire X → Y1 Y2 ... Yn;
            sinon
                Écrire("Erreur Syntaxique")
        sinon
            si (X = '#') alors
                si a = '#' alors
                    Accepter();
                sinon
                    Écrire("Erreur :# expected");
            sinon
                si (X = a) alors
                    dépiler();
                    Avancer();
                sinon
                    Écrire("Erreur :a expected");
    jusqu'à (Erreur() ou accepter());
fin
```

Algorithme 15 : Algorithme d'analyse LL(1)

Exemple 17 *Soit la grammaire G de l'exemple 15. Les étapes d'analyse de la chaine* nb+nb*nb# *sont données par la Table 17.*

Pile	Chaine	Action
#E	nb+nb*nb#	E → TE'
#E'T	nb+nb*nb#	T → FT'
#E'T'F	nb+nb*nb#	F → nb
#E'T'nb	nb+nb*nb#	avancer() ; dépiler() ;
#E'T'	+nb*nb#	T' → ϵ
#E'	+nb*nb#	E' → +TE'
#E'T+	+nb*nb#	avancer() ; dépiler() ;
#E'T	nb*nb#	T → FT'
#E'T'F	nb*nb#	F → nb
#E'T'nb	nb*nb#	avancer() ; dépiler() ;
#E'T	*nb#	T' → *FT'
#E'T'F*	*nb#	avancer() ; dépiler() ;
#E'T'F	nb#	F → nb
#E'T'nb	nb#	avancer() ; dépiler() ;
#E'T'	#	T' → ϵ
#E'	#	T' → ϵ
#	#	Accept

TABLE 4.4 – Analyse de la chaîne *i+i+3#*

Exemple 18 *Soit G la grammaire définie par <T,N,S,P> où T={a,b,d,e} ; N={S,A,B,D} ; et P :*

$$\begin{cases} Z \rightarrow S\# \\ S \rightarrow AB\textcircled{1} \mid Da\textcircled{2} \\ A \rightarrow aAb\textcircled{3} \mid \epsilon\textcircled{4} \\ B \rightarrow bB\textcircled{5} \mid \epsilon\textcircled{6} \\ D \rightarrow dD\textcircled{7} \mid e\textcircled{8} \end{cases}$$

Les débuts et les suivants associés aux non terminaux de la grammaire G sont donnés par la Table 18.

	Débuts	Suivants
S	a, b, d, e, ϵ	#
A	a, ϵ	b,#
B	b, ϵ	#
D	d,e	a

TABLE 4.5 – Les débuts et les suivants de la grammaire G de l'Exemple 18

La table d'analyse LL(1) associée à la grammaire G est donnée par la Table 18.

	a	b	d	e	#
S	$S \rightarrow AB$	$S \rightarrow AB$	$S \rightarrow Da$	$S \rightarrow Da$	$S \rightarrow AB$
A	$A \rightarrow aAb$	$A \rightarrow \epsilon$			$A \rightarrow \epsilon$
B		$B \rightarrow bB$			$B \rightarrow \epsilon$
D			$D \rightarrow dD$	$D \rightarrow e$	

TABLE 4.6 – Table LL(1) de la grammaire de l'exemple 18

La grammaire G est LL(1) car :

1. *La grammaire G est non récursive gauche (ni directe ni indirecte),*
2. *G est factorisée,*
3. *La table d'analyse de la grammaire G est mono-définie*

définition 4.1 *Une grammaire G est dite LL(1) si :*

1. *G est non récursive gauche directe et non récursive gauche indirecte,*
2. *G est factorisée,*
3. *Pour toutes productions du type* $A \rightarrow \alpha_1 \mid \alpha_2 ... \mid \alpha_n, \alpha_i \in (T \cup N)^+$ *alors*
   (a) *Débuts*$(\alpha_i) \cap$ *Débuts*$(\alpha_j) = \emptyset \forall i, j \in [1, n]$ *avec* $i \neq j$ *(retour arrière),*
   (b) *Si* $\alpha_i \xrightarrow{*} \epsilon$ *alors Débuts*$(\alpha_i) \cap Suivants(A) = \emptyset$ *(non déterminisme),*
   (c) *Si* $\alpha_i \xrightarrow{*} \epsilon$ *alors* $\alpha_j \overset{*}{\nrightarrow} \epsilon \forall j \neq i$ *(problème d'ambiguïté).*

Remarque 6 *Ainsi, il est possible de vérifier si une grammaire est LL(1) sans construire la table d'analyse en vérifiant les conditions citées dans la définition 4.1.*

Exemple 19 *18 Soit la grammaire G définie dans l'exemple 18. La grammaire G est LL(1) car :*

1. *La grammaire G est non récursive gauche (ni directe ni indirecte),*
2. *G est factorisée,*
3. *la table d'analyse est mono-définie car :*
   (a) *Pour les règles* $S \rightarrow AB \mid Da$
   $Débuts(AB) \cap Débuts(Da) = \emptyset$,
   (b) *Pour les règles* $A \rightarrow aAb \mid \epsilon$
   $Débuts(aAb) \cap Suivants(A) = \emptyset$,

(c) *Pour les règles* $B \rightarrow bB \mid \epsilon$
$D\acute{e}buts(bB) \cap Suivants(B) = \emptyset$,

(d) *Pour les règles* $D \rightarrow dD \mid e$
$D\acute{e}buts(dD) \cap D\acute{e}buts(e) = \emptyset$,

Les étapes d'analyse de la chaîne abdb#*sont données par la table 19.*

Pile	Chaine	Action
#S	abdb#	S → AB : Dépiler() ; Empiler(miroir(AB)) ;
#BA	abdb#	A → aAb :Dépiler() ; Empiler(miroir(aAb)) ;
#BbAa	abdb#	Avancer() ; Dépiler() ;
#BbA	bdb#	A → ϵ : Dépiler() ;
#Bb	bdb#	Avancer() ; Dépiler() ;
#B	db#	Erreur :b ou # expected ;

TABLE 4.7 – Analyse de la chaîne *abdb#*

Exemple 20 *Soit G la grammaire définie par <T,N,S,P> où T={i,t,a,e,b} ; N={S,S',E} ; et P :*

$$\begin{cases} Z \rightarrow S\# \\ S \rightarrow iEtSS' \mid a \\ S' \rightarrow eS \mid \epsilon \\ E \rightarrow b \end{cases}$$

Les débuts et les suivants associés aux non terminaux de la grammaire G sont donnés par la Table suivante :

	Débuts	Suivants
S	i,a	#,e
S'	e, ϵ	e,#
E	b	t

TABLE 4.8 – Les débuts et les suivants de la grammaire G de l'Exemple 20

1. *La grammaire G est non récursive gauche (ni directe ni indirecte),*
2. *G est factorisée,*
3. (a) *Pour les règles S → iEtSS' | a , Débuts(iEtSS') ∪ Débuts(a)=∅*
   (b) *Pour les règles S' → eS | ϵ Débuts(eS) ∩ Suivants(S')={e} ≠ ∅.*

D'où, la grammaire G n'est pas LL(1).

Remarque 7 *Si une grammaire G est LL(1) ⇒ G est LL(k) $\forall k \geq 1$.*
Une grammaire G est dite LL(k) si k entités lexicales de la chaîne à analyser sont nécessaires afin de rendre l'analyse syntaxique déterministe.

Exemple 21 *Soit G la grammaire définie par <T,N,S,P> où T={debut, tantque, c, fin, i, ;} ; N={S,L} ; et P :*

$$\begin{cases} Z \rightarrow S\# \\ S \rightarrow \textit{tantque c S} \mid \textit{debut L fin} \mid i \\ L \rightarrow L;S \mid S \end{cases}$$

- *G est-elle LL(1) ?*
  1. *G est récursive gauche directe en L. L'élimination de la récursivité gauche directe en L produit les règles suivantes :*
     L → SL'
     L' →;SL' | ϵ
     Ainsi, la grammaire G' non récursive gauche équivalente à G est définie par : <T,N',S,P'> où T={début, tantque, c, fin, i, ;} ; N'={S,L,L'} ; et P' :

$$\begin{cases} Z \rightarrow S\# \\ S \rightarrow \textit{tantque c S} \mid \textit{début L fin} \mid i \\ L \rightarrow SL' \\ L' \rightarrow ;SL' \mid \epsilon \end{cases}$$

     L'ensemble des débuts et l'ensemble des suivants associés aux non terminaux de la grammaire G sont donnés par la Table suivante :

	Débuts	Suivants
S	tantque début i	fin ; #
L	tantque début i	fin
L'	; ϵ	fin

TABLE 4.9 – Les débuts et les suivants de la grammaire G

  2. *G' est factorisée,*
  3. *la table d'analyse associée à la grammaire G' est mono-définie car :*
     — *pour les règles S → tantque c S | Debut L fin | i*

— *Débuts(tantque c S) ∩ Débuts(début L fin)=∅*

— *Débuts(tantque c S) ∩ Débuts(i)=∅*

— *Débuts(début L fin) ∩ Débuts(i)=∅*

— *S' contient un seul MDP,*

— *pour les règles L' →;SL' | ϵ , Débuts(;SL') ∩ Suivants(L')=∅*

- *La table d'analyse LL(1) associée à la grammaire G' est définie comme suit :*

	tanque	c	début	fin	i	;	#
S	$S \rightarrow$ tantque c S		$S \rightarrow$ debut L fin		$S \rightarrow$ i		
L	$L \rightarrow$ SL'		$L \rightarrow$ SL'		$L \rightarrow$ SL'		
L'				$L' \rightarrow \epsilon$		L' →; SL'	

TABLE 4.10 – Table d'analyse LL(1) de la grammaire G'

La mono-définition de la table d'analyse de la grammaire G' confirme bien que G' est LL(1).

- *Les étapes d'analyse de la chaîne* debut i ;i fin#*sont données par la table suivante.*

Pile	**Chaine**	**Action**
#S	début i ;i fin#	S → début L fin : Dépiler() ; Empiler(miroir(début L fin)) ;
#fin L début	début i ;i fin#	Avancer() ; Dépiler() ;
# fin L	i ;i fin#	L → SL' : Dépiler() ; Empiler(miroir(SL'))
#fin L'S	i ;i fin#	S → i : Dépiler() ; Empiler(miroir(i)
#fin L'i	i ;i fin#	Avancer() ; Dépiler() ;
#fin L'	;i fin#	L' → ;SL' : Dépiler() ; Empiler(miroir(;SL'))
#fin L'S ;	;i fin#	Avancer() ; Dépiler() ;
#fin L'S	i fin#	S → i : Dépiler() ; Empiler(miroir(i)
#fin L'i	i fin#	Avancer() ; Dépiler() ;
#fin L'	fin#	L' → ϵ : Dépiler() ;
#fin	fin#	Avancer() ; Dépiler() ;
#	#	Chaîne acceptée :

TABLE 4.11 – Analyse de la chaîne *début i ;i fin#*

4.2.1 La méthode de la descente récursive

C'est une méthode descendante déterministe qui s'applique aux grammaires dont les conditions LL(1) sont vérifiées. Elle consiste à associer à chaque non-terminal de la grammaire une procédure qui traite toutes ses membres droits. L'analyse se fait à l'aide d'appels de procédures.

Exemple 22 *Soit* G *la grammaire définie par <T,N,S,P> où T={a b ( ) ,} ; N={S,T} ; et P :*

$$\begin{cases} Z \rightarrow S\# \\ S \rightarrow a \mid b(T) \\ T \rightarrow T, S \mid S \end{cases}$$

- *Vérification des conditions LL(1) :*
  1. *La grammaire G est récursive directe en T. L'élimination de la récursivité gauche directe en T se réalise comme suit :*
     Les règles $T \rightarrow T\underbrace{,S}_{\alpha} \mid \underbrace{S}_{\beta}$ *vont être transformées en :*
     T → ST'
     T' → ,ST' | ϵ
     Ainsi, la grammaire G' non récursive gauche équivalente à G est définie par <T,N',S,P> où T={a b ( ) ,} ; N'={S,T,T'} ; et P :

$$\begin{cases} Z \rightarrow S\# \\ S \rightarrow a \mid b(T) \\ T \rightarrow ST' \\ T' \rightarrow , ST' \mid \epsilon \end{cases}$$

TABLE 4.12 – Les débuts et les suivants associés aux non terminaux de la grammaire G'

	début	suivant
S	a b	# ,)
T	a b	)
T'	, ϵ	)

  2. *G' est factorisée.*
  3. *La table d'analyse de la grammaire G' est mono-définie car :*
     — *pour S, débuts(a) ∩ débuts(b(T) =∅,*
     — *pour T, il existe un seul MDP,*

— *pour T', débuts(,ST') ∩ suivants(T')=∅*

La grammaire G' vérifient toutes les conditions LL(1).

- *définition des procédures :*

Programme principal

```
Z() : Z → S #
début
    Lire(chaine) ; tc := 1^ere entité de la chaîne ;
    si (tc ∈ debut(S)) alors
        S() ;
        si (tc = '#') alors
            Écrire('Chaine acceptée') ;
        sinon
            Écrire("Erreur :# expected")) ;
    sinon
        Écrire("Erreur : a ou b expected")) ;
fin
```

Procédure S() : S → a | b(T)

```
début
    si (tc = 'a') alors
        tc := ts ;
    sinon
        tc := ts ;
        si (tc = '(') alors
            tc := ts ;
            si (tc ∈ debut(T)) alors
                T() ;
                si (tc =')') alors
                    tc := ts ;
                sinon
                    Écrire("Erreur :) expected") ;
            sinon
                Écrire("Erreur : a ou b expected") ;
        sinon
            Écrire("Erreur :( expected") ;
fin
```

Procédure T() :T → ST'

```
début
    S() ;
    si (tc ∈ Débuts(T')-{ϵ }) alors
        T'() ;
    sinon si (tc ∉ suivant(T)) alors
        Écrire("Erreur :) ou , expected") ;
fin
```

Procédure T'() : T' → ,ST' | ϵ

```
début
    tc := ts ;
    si (tc ∈ debut(S)) alors
        S() ;
        si (tc ∈ Débuts(T')-{ϵ }) alors
            T'() ;
        sinon si (tc ∉ Suivants(T')) alors
            Écrire("Erreur :, ou ) expected") ;
    sinon
        Écrire("Erreur :a ou b expected") ;
fin
```

- *Analyse de la chaine b(a,a)#*

Pile	Chaine	Action
#Z	b(a,a)#	Appel(S)
#ZS	b(a,a)#	Avancer()
#ZS	(a,a)#	Avancer()
#ZS	a,a)#	Appel(T)
#ZST	a,a)#	Appel(S)
#ZSTS	a,a)#	Avancer()
#ZSTS	,a)#	Dépiler(S)
#ZST	,a)#	Appel (T')
#ZSTT'	,a)#	Avancer()
#ZSTT'	a)#	Appel (S)
#ZSTT'S	a)#	Avancer()
#ZSTT'S	)#	Dépiler (S)
#ZSTT'	)#	Dépiler (T')
#ZST	)#	Dépiler (T)
#ZS	)#	Avancer()
#ZS	#	Dépiler(S)
#Z	#	Chaine acceptée

- *Analyse de la chaine b(ba)#*

Pile	Chaine	Action
#Z	b(ba)#	Appel(S)
#ZS	b(ba)#	Avancer()
#ZS	(ba)#	Avancer()
#ZS	ba)#	Appel(T)
#ZST	ba)#	Appel(S)
#ZSTS	ba)#	Avancer()
#ZSTS	a)#	Erreur ('(expected')

Remarque 8 *L'appel d'une procédure induit l'empilement du non terminal correspondant. A la fin d'une procédure, le non terminal correspondant est alors dépilé.*

4.3 Conclusion

La méthode de la descente récursive est simple, efficace et elle est facile à implémenter. Elle présente cependant un inconvénient majeur : la méthode est liée à la grammaire. Si une règle est modifiée alors il va falloir apporter des modification à la procédure correspondante.

5 Analyse ascendante

Le principe de l'analyse ascendante est de construire un arbre de dérivation de la base (les feuilles) vers le haut (la racine représentée par l'axiome). Le modèle utilisé est celui de *Shift/Reduce* (Décalage/Réduction), cela signifie qu'il existe deux opérations :

Décalage (*Shift*) consiste à décaler d'un élément le pointeur sur la chaine en entrée.

Réduction (*Reduce*) consiste à réduire une suite de terminaux et de non-terminaux en un non-terminal en utilisant une règle de production.

Soit G la grammaire définie par <T,N,S,P> où T={a b c d} ; $\mathbb{N}$={S,A,B} et P :

$$\begin{cases} Z \rightarrow S\# \\ S \rightarrow AcA\textcircled{1} \mid aBA \mid ScBa\textcircled{2} \\ A \rightarrow SBa\textcircled{3} \mid b\textcircled{4} \\ B \rightarrow cB\textcircled{5} \mid dA\textcircled{6} \mid c\textcircled{7} \end{cases}$$

L'arbre syntaxique généré d'une manière ascendante associé à la chaîne 'adbbccacbcdba' est donné par la Figure 23.

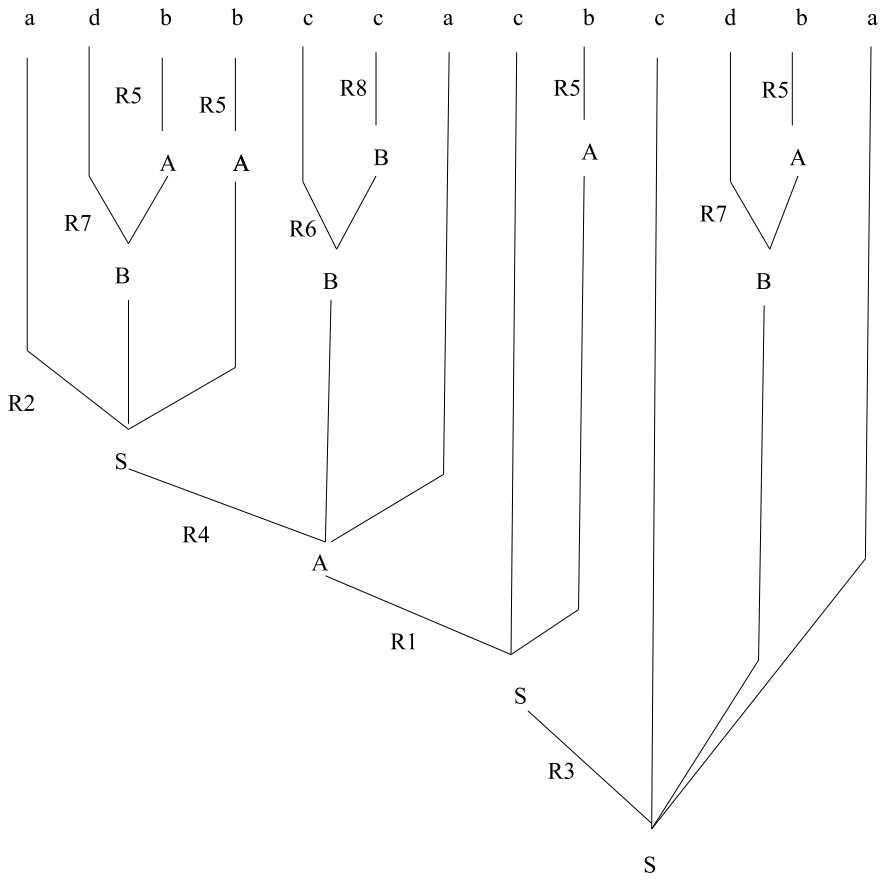


FIGURE 4.13 – Arbre syntaxique associé à la chaîne 'adbbccacbcdba'

Les méthodes d'analyse LR sont des méthodes d'analyses syntaxiques ascendantes déterministes efficaces. Elles consistent, à partir d'une chaîne à analyser, d'exécuter un ensemble d'actions basées sur des décalages et des réductions (shift/reduce) afin d'aboutir à l'axiome de la grammaire. Les analyseurs issus de la classe LR lisent la chaîne d'entrée de gauche à droite (Left to right) afin de construire une dérivation (Right).

Définition 4.4 *Une dérivation droite est une dérivation qui remplace à chaque étape le symbole terminal le plus à droite.*

Le fonctionnement d'un analyseur de la classe LR nécessite :

- la chaîne à analyser,

- une pile, initialisée à l'état initial. Elle contient, en alternance, les symboles de la grammaire X $\in (T \cup N)^*$ et les états,

- une table d'analyse M qui décrit un automate à états finis déterministe associé à des actions à effectuer sur la pile et sur la chaîne. Elle est composée de deux partie :

  1. la table des actions dont les indices des colonnes du tableau sont constitués par les terminaux de la grammaire ainsi que le symbole de fin d'entités # et et les indices des lignes représentent les différents états développés en exploitant des procédures spécifiques à la méthode (fermeture et GOTO). Le contenu de cette partie de la table spécifie quatre types d'action en fonction de l'état courant du sommet de pile et de l'entité courante :

     (a) Décaler (Shift), noté *D,i* qui indique qu'il faut déplacer l'entité courante dans la pile,

     (b) Réduire (Reduce), noté $R_{n\check{r}r\grave{e}gle}$ du type A$\rightarrow \alpha$ avec A $\in$ N et $\alpha \in (T \cup N)^*$ qui indique qu'il faut réduire α à A,

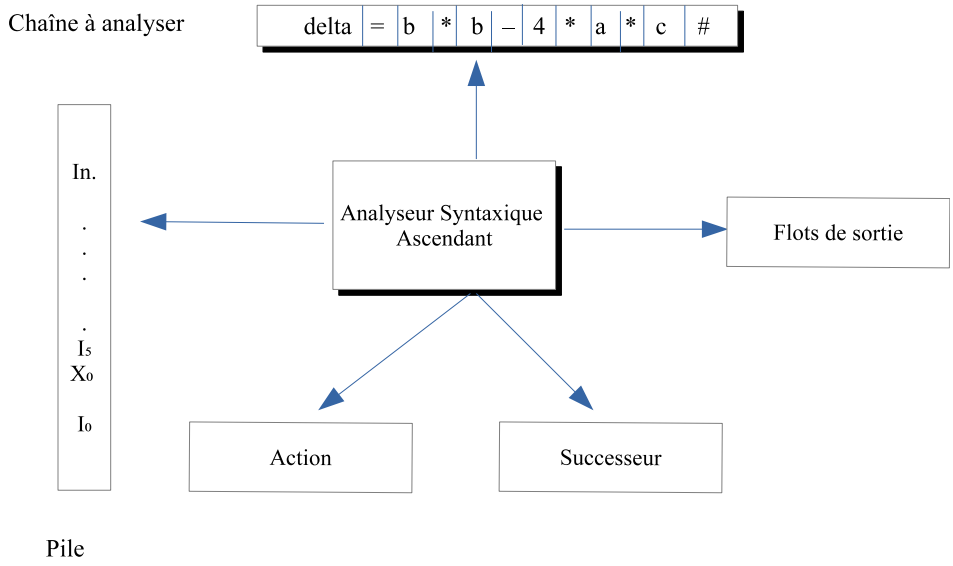
     (c) Erreur suivie du numéro de la ligne du programme concerné ainsi que le code de l'erreur,

     (d) Accepter.

  2. la table des branchements, dont les indices des colonnes du tableau sont les non terminaux de la grammaires, qui indique le prochain état ou l'état successeur.

Le fonctionnement d'un analyseur ascendant est illustré par la Figure suivante.

FIGURE 4.14 – Schéma général d'un analyseur syntaxique ascendant



Il existe plusieurs méthodes d'analyse ascendantes basées sur les opérations de décalage/réduction (Shift/Reduce). Les analyseurs LR basés sur les items figurent parmi les méthodes déterministes ascendantes les plus puissantes. Ils englobent trois principaux types :

1. L'analyseur LR(K)qui procède comme suit :
   - les terminaux sont lus de gauche à droite (L),
   - l'analyseur cherche une dérivation droite (R),
   - l'analyseur décide de l'action à exécuter en consultant K entités lexicales.
2. l'analyseur Simple LR (SLR(1)) est un cas particulier de l'analyseur LR(K) où K=0. Ce qui signifie que seul le contenu de la pile suffit pour décider de l'action à exécuter. Cette méthode est facile à implémenter mais peu de grammaires sont SLR.
3. L'analyseur LALR(1) (LookAhead LR) qui est une version optimisée de l'analyseur LR(1) qui s'applique à la plupart des grammaires des langages de programmation.

définition 5.1 *une grammaire est LR(0) si le contenu de la pile suffit pour décider de l'action à exécuter.*

définition 5.2 *Une grammaire est LR(k) si elle nécessite k entités de la chaine afin de décider de l'action à exécuter.*

5.1 Analyse LR(K) par les items

Une méthode pratique d'analyse ascendante est la méthode LR par les items qui se base sur le calcul des items LR(K). Un item est une production de la grammaire avec un point (.) repérant la position de son *MDP*. la partie gauche du point représente le sous-arbre de la grammaire ayant déjà été réduite au cours de l'analyse et la partie à droite du point représente ce qui reste à analyser.

définition 5.3 *un item LR(k) est de la forme :*

[$A \rightarrow \alpha.\beta$,w] $A \in \mathbb{N}$; $\alpha,\beta \in (T \cup \mathbb{N})^*$; $w \in T^* \cup \{\#\}$ tels que :
$A \rightarrow \alpha\beta$ est une règle de la grammaire.
α : constitue ce qui a été réduit,
β : constitue la partie qui reste à analyser,
k représente le nombre d'entités lexicales à consulter afin de décider de l'action à exécuter (shift/reduce).

Si k=0 l'item correspondant est un item LR(0) ayant la forme [$A \rightarrow \alpha.\beta$]

Le fonctionnement d'un analyseur LR(K) est guidé par la table d'analyse qui requiert au préalable la construction de la collection de toutes les fermeture des items LR(K).

5.1.1 La construction de l'ensemble des items LR(K)

La construction des ensembles des items LR(K) de la grammaire nécessite la mise en œuvre de deux fonctions :

- la fermeture d'un item,
- la fonction goto.

1. Le calcul de la fermeture d'un item se fait à l'aide de l'algorithme 16.

début
  pour *chaque item de la forme [A → α.Bβ,w] et B ∈ ℕ de l'ensemble des item I* **faire**
    pour *chaque règle B → γ , γ∈ (T ∪ ℕ)** **faire**
      répéter
        ajouter dans l'item I, l'item [B → .γ,debut$_k$(βw)
      jusqu'à *ce qu'il n'y ait plus d'items à rajouter*;
    fin
  fin
fin

Algorithme 16 : Calcul de la fermeture d'un item LR(K)

2. La fonction GOTO (I,X)tels que :

   — I représente l'ensemble des items,

   — X est un symbole de G, X ∈ $(T \cup N)^*$

   est la fermeture de l'ensemble formé de tous les items de la forme [A → αX.β,w] tel que [A → α.Xβ,w] ∈ I.

Après avoir défini les procédures de la construction des items et la fonction GOTO, l'algorithme de construction de la collection C des items LR(K) est comme suit :

début
  Augmenter la grammaire G avec la règle Z → S#
  I_0 ← Fermeture [Z → .S,#]
  Mettre I_0 dans la collection ;
  répéter
    pour *chaque ensemble d'items I ∈ collection* **faire**
      pour *Chaque symbole X de la grammaire* **faire**
        si *(GOTO(I,X) n'est pas vide) et (il n'est pas déjà dans la collection C))* **alors**
          rajouter GOTO(I,X) à la collection C ;
        fin
      fin
    fin
  jusqu'à *il n'y ait plus d'items à rajouter*;
fin

Algorithme 17 : Construction de la collection des items LR(K)

5.1.2 Construction de la table d'analyse LR(k)

La construction de la table d'analyse LR(K) est réalisée en appliquant l'algorithme 18.

début
  Construire la collection des items LR(K) ;
  si I_j = *GOTO(I_i,A) avec A* $\in N$ **alors**
    T[I_i,A] := I_j
  fin
  si I_j = *GOTO(I_i,a) avec a* $\in T$ **alors**
    pour *chaque item de* I_i *de la forme* $[A \rightarrow \alpha.a\overset{w'}{(\beta w)}]$ **faire**
      pour *chaque élément aβw' de* $debut_k(aw')$ **faire**
        T[I_i,aw'] := D,$_j$
      fin
    fin
  fin
  si *dans* I_i *il existe un item de la forme* $[A \rightarrow \alpha.,\omega]$ **alors**
    T[I_i,ω] := $R_{A\rightarrow\alpha}$
  fin
  si $[A \rightarrow S.,\#] \in I_j$ **alors**
    T[I_j,#] :="Accepter"
  fin
fin

Algorithme 18 : Construction de la table d'analyse LR(K)

définition 5.4 *Une grammaire G est LR(K) par la méthode des items si la table d'analyse LR(K) est mono-définie.*

Ainsi, afin de vérifier si la grammaire G est LR(1), il faut :

1. calculer la collection de la fermeture des items LR(1),
2. construire la table d'analyse LR(1),
3. **si** *(la table d'analyse LR(1) est mono-définie)* **alors**
   G est LR(1)⇒ G est LR(K) $\forall$ K $\geq$ 1
   fin

Remarque 9 — *Cette méthode permet d'analyser plus de grammaires que la méthode descendante car il y a plus de grammaires qui sont LR(1) que LL(1).*

— *La récursivité gauche ne gène pas lors de l'analyse ascendante.*

Exemple 23 *Soit G la grammaire définie par <T,N,S,P> où T={a b c} ; N={S} ; et P :*

$$\begin{cases} S \rightarrow Sbc\textcircled{1} \mid a\textcircled{2} \end{cases}$$

La grammaire augmentée associée à G est obtenue en rajoutant à G la règle Z→ S# :

$$\begin{cases} Z \rightarrow S\# \\ S \rightarrow Sbc\textcircled{1} \mid a\textcircled{2} \end{cases}$$

1. *Calcul des items LR(1) :* I_0 = { [Z → .S,#] [S → .Sbc,#] [S → .a,#] [S → .Sbc,b] [S → .a,b] }
   $I_1 = GOTO(I_0,S)$ = { [Z → S.,#] [S → S.bc,#] [S → S.bc,b] }
   $I_2 = GOTO(I_0,a)$ = { [S → a.,#] [S → a.,b] }
   $I_3 = GOTO(I_1,b)$ = { [S → Sb.c,#] [S → Sb.c,b] } = {[S → Sb.c,#/b]}
   $I_4 = GOTO(I_3,c)$ = { [S → Sbc.,#/b] }

   Ainsi, la collection des fermetures des items LR(1) obtenue est C = $\{I_0,I_1,I_2,I_3,I_4\}$.
2. *Construction de la table d'analyse LR(1) :*

TABLE 4.13 – Table d'analyse LR(1) associée à la grammaire G

	a	b	c	#	S
I_0	D,2				1
I_1		D,3		Accept	
I_2		S → *a*		S → *a*	
I_3			D,4		
I_4		S → *Sbc*		S → *Sbc*	

3. *La table mono-définie ⇒ G est LR(1) ⇒ G est LR(K)* $\forall\ k \geq 1$.

Soit à analyser la chaîne abc# :

Pile	**Chaine**	**Action**
⓪	abc#	D,2
⓪*a*②	bc#	R2
⓪S①	bc#	D,3
⓪S①b③	c#	D,4
⓪*S*①*b*③*c*④	#	R1
⓪S①	#	ACCEPT

L'arbre syntaxique généré est donné par la Figure suivante :

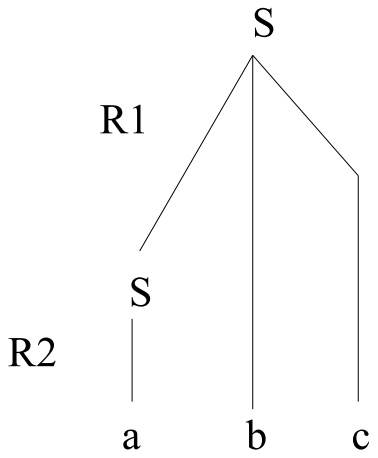


FIGURE 4.15 – Arbre syntaxique généré correspondant à la chaîne abc

5.1.3 Conclusion

La méthode d'analyse LR(1) est performante. Elle permet d'analyser plus de grammaires que les méthode descendantes car il y a plus de grammaires LR(1) que des grammaires LL(1). De plus, la récursivité gauche ne gène pas lors de l'analyse ascendante car le processus d'analyse est guidé par la chaîne contrairement à l'analyse LL(1), qui à partir de l'axiome, opère un certain nombre de dérivations jusqu'à obtenton de la chaîne à analyser. Néanmoins, étant donné que la table d'analyse associée peut être très volumineuse, son implémentation est couteuse en temps et en espace en raison de la présence de l'entité lookahead.
Afin de palier à ces contraintes par rapport aux coûts induits par la méthode LR(1), la méthode d'analyse SLR(1) a été mise au point. Cette dernière ne nécessite pas de lire des entités supplémentaires pour décider de l'action à exécuter.

5.2 Analyseur Simple LR(1)par les items

La méthode SLR(1) est basée sur le calcul des items LR(0). C'est une version simplifiée de la méthode LR(K) étant donné que K=0.

définition 5.5 *un item LR(0) est de la forme :*

$[A \rightarrow \alpha.\beta]$ $A \in \mathbb{N}$, $\alpha, \beta \in (T \cup \mathbb{N})^*$
α : constitue ce qui a été réduit,
β : constitue la partie qui reste à analyser, $A \rightarrow \alpha\beta$ est une règle de la grammaire.

La mise en œuvre d'un analyseur SLR(1) est guidé par la table d'analyse SLR(1) qui requiert au préalable la construction de la collection de toutes les fermeture des items LR(0).

5.2.1 La construction de l'ensemble des items LR(0)

La construction des ensembles des items LR(0) de la grammaire se base sur deux fonctions :

- la fermeture d'un item LR(0),
- la fonction GOTO.

1. Le calcul de la fermeture d'un item LR(0) se fait à l'aide de l'algorithme 23.

```
début
    pour chaque item de la forme [A → α.Bβ] et B ∈ ℕ de l'ensemble des
     item I faire
        pour chaque règle B → γ , γ∈ (T ∪ ℕ)* faire
            répéter
                ajouter dans l'item I, l'item [B → .γ
            jusqu'à ce qu'il n'y ait plus d'items à rajouter;
        fin
    fin
fin
```

Algorithme 19 : Calcul de la fermeture d'un item LR(0)

2. La fonction GOTO (I,X)tels que :
   (a) I représente l'ensemble des items,
   (b) X est un symbole de G, X $\in (T \cup N)^*$

   est la fermeture de l'ensemble formée de tous les items de la forme [A $\rightarrow$ αX.β] tel que [A $\rightarrow$ α.Xβ] $\in$ I.

En exploitant les procédures de calcul de la fermeture d'un item LR(0) et la fonction GOTO, la construction de la collection C des items LR(0) est réalisé comme

suit :

début
  Augmenter la grammaire G avec la règle
  Z → S#
  I_0 ← Fermeture [Z → .S]
  Mettre I_0 dans la collection ;
  répéter
    pour *chaque ensemble d'items I ∈ collection* **faire**
      pour *Chaque symbole X de la grammaire* **faire**
        si *(GOTO(I,X) n'est pas vide) et (il n'est pas déjà dans la collection C))* **alors**
          rajouter GOTO(I,X) à la collection C ;
        fin
      fin
    fin
  jusqu'à *ce qu'il n'y ait plus d'items à rajouter*;
fin

Algorithme 20 : Construction de la collection des items LR(0)

5.2.2 Construction de la table d'analyse SLR(1)

La construction de la table d'analyse SLR(1) est réalisée en appliquant l'algorithme 25.

début
  Construire la collection des items LR(0) ;
  si *(I_j = GOTO(I_i,A) avec A ∈ N)* **alors**
    T[I_i,A] := I_j
  fin
  si *(I_j = GOTO(I_i,a) avec a ∈ T)* **alors**
    T[I_i,a] := D_j
  fin
  si *(I_i contient un item de la forme [A → α.)]* **alors**
    T[I_i,a] := $R_{A\to\alpha}$ avec a ∈ suivants(A)
  fin
  si *([A → S.] ∈ I_j)* **alors**
    T[I_j,#] :="Accepter"
  fin
fin

Algorithme 21 : Construction de la table d'analyse SLR(1)

définition 5.6 *Une grammaire G est SLR(1) par la méthode des items si la table d'analyse SLR(1) est mono-définie.*

Ainsi, afin de vérifier si la grammaire G est SLR(1), il faut :

1. calculer la collection de la fermeture des items LR(0),
2. construire la table d'analyse SLR(1),
3. **si** *(la table d'analyse SLR(1) est mono-définie)* **alors**
   | G est SLR(1)⇒ G est SLR(K) ∀ K ≥ 1
   fin

Remarque 4.1 *Pour la méthode LR(1), les opérations de réductions qui correspondent aux items LR(1) du type $S \rightarrow \alpha.,w$, s'effectuent avec l'entité lookahead w. Par contre, pour la méthode SLR(1), les opérations de réductions qui correspondent aux items du type $A \rightarrow \alpha.$ s'effectuent avec les suivants du membre gauche de la production (suivants(A)).*

Exemple 24 *Soit G la grammaire définie par <T,N,E,P> où T={+ * ( ) nb} ; N={E,T,F} ; et P :*

$$\begin{cases} E \rightarrow E + T\textcircled{1} \mid T\textcircled{2} \\ T \rightarrow T * F\textcircled{3} \mid F\textcircled{4} \\ F \rightarrow (E)\textcircled{5} \mid nb\textcircled{6} \end{cases}$$

La grammaire augmentée associée à G est obtenue en rajoutant à G la règle Z→ E# :

$$\begin{cases} Z \rightarrow E\# \\ E \rightarrow E + T\textcircled{1} \mid T\textcircled{2} \\ T \rightarrow T * F\textcircled{3} \mid F\textcircled{4} \\ F \rightarrow (E)\textcircled{5} \mid nb\textcircled{6} \end{cases}$$

1. *Calcul des débuts et des suivants*

	debuts	suivants
E	nb (	# +)
T	nb (	# + *)
F	nb (	# + *)

2. *Calcul de la collection des items LR(0)*

   I_0 = { [S → .E] [E → .E+T] [E → .T] [T → .T*F] [T → .F] [F → .(E)] [F

$\rightarrow .nb]\ \}$

$I_1 = GOTO(I_0, E) = \{\ [E' \rightarrow E.]\ [E \rightarrow E.+T]\ \}$

$I_2 = GOTO(I_0, T) = \{\ \underline{[E \rightarrow T.]}\ [T \rightarrow T.*F]\ \}$

$I_3 = GOTO(I_0, F) = \{\ \underline{[T \rightarrow F.]}\ \}$

$I_4 = GOTO(I_0, () = \{\ [F \rightarrow (.E)]\ [E \rightarrow .E+T]\ [E \rightarrow .T]\ [T \rightarrow .T*F]\ [T \rightarrow .F]\ [F \rightarrow .(E)]\ [F \rightarrow .nb]\ \}$

$I_5 = GOTO(I_0, nb) = \{\ \underline{[F \rightarrow nb.]}\ \}$

$I_6 = GOTO(I_1, +) = \{\ [E \rightarrow E+.T]\ [T \rightarrow .T*F]\ [T \rightarrow .F]\ [F \rightarrow .(E)]\ [F \rightarrow .nb]\ \}$

$I_7 = GOTO(I_2, *) = \{\ [T \rightarrow T*.F]\ [F \rightarrow .(E)]\ [F \rightarrow .nb]\ \}$

$I_8 = GOTO(I_4, E) = \{\ [F \rightarrow (E.)]\ [E \rightarrow E.+T]\ \}$

$I_2 = GOTO(I_4, T)$

$I_3 = GOTO(I_4, F)$

$I_4 = GOTO(I_4, ()$

$I_5 = GOTO(I_4, nb))$

$I_9 = GOTO(I_6, T) = \{\ \underline{[E \rightarrow E+T.]}\ [T \rightarrow T.*F]\ \}$

$I_3 = GOTO(I_6, F)$

$I_4 = GOTO(I_6, ()$

$I_5 = GOTO(I_6, nb)$

$I_{10} = GOTO(I_7, *) = \{\ \underline{[T \rightarrow T*F.]}\ \}$

$I_4 = GOTO(I_7, ()$

$I_5 = GOTO(I_7, nb)$

$I_{11} = GOTO(I_8,)) = \{\ \underline{[F \rightarrow (E).]}\ \}$

$I_6 = GOTO(I_8, +)$

$I_7 = GOTO(I_9, *)$

Remarque 4.2 *Les items soulignés correspondent à des actions de réductions car le . est à la fin des MDP.*

3. *Construction de la table d'analyse SLR(1)*

	nb	+	*	(	)	#	E	T	F
0	D,5			D,4			1	2	3
1		D,6				ACCEPT			
2		R2	D,7		R2	R2			
3		R4	R4		R4	R4			
4	D,5			D,4			8	2	3
5		R6	R6		R6	R6			
6	D,5			D,4				9	3
7	D,5			D,4					10
8		D,6			D,11				
9		R1	D,7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

TABLE 4.14 – Table d'analyse SLR(1) de la grammaire de l'exemple 24

La table d'analyse SLR(1) est mono-définie ⇒ G est SLR(1)⇒ G est SLR(k) avec k ≥ 1.

*Soit à analyser la chaine **3+4*2#. Après l'analyse lexicale, cette suite d'entité sera codifiée comme suit :** $nb_3 + nb_4 * nb_2$.*

***Remarque 10** Le chiffre représenté en indice indique la valeur de l'entité nb.*

Pile	**Chaine**	**Action**
#⓪	$nb_3 + nb_4 * nb_2$#	D,5
#⓪$\underbrace{nb_3⑤}$	$+nb_4 * nb_2$#	R6 : F → nb
#⓪$\underbrace{F③}$	$+nb_4 * nb_2$#	R4 : T → F
#⓪$\underbrace{T②}$	$+nb_4 * nb_2$#	R2 : E → T
#⓪E①	$+nb_4 * nb_2$#	D,6
#⓪E①+⑥	$nb_4 * nb_2$#	D,5
#⓪E①+⑥$\underbrace{nb_4⑤}$	$*nb_2$#	R6 : F → nb
#⓪E①+⑥$\underbrace{F③}$	$*nb_2$#	R4 : T → F
#⓪E①+⑥T⑨	$*nb_2$#	D,7
#⓪E①+⑥T⑨*⑦	nb_2b#	D,5
#⓪E①+⑥T⑨*⑦$\underbrace{nb_2⑤}$	#	R6 : F → nb
#⓪E①+⑥$\underbrace{T⑨ * ⑦F⑩}$	#	R3 : T → T*F
#⓪$\underbrace{E① + ⑥T⑨}$	#	R1 : E → E+T
#⓪E①	#	ACCEPT

L'arbre syntaxique, représenté par la Figure 4.17 et généré lors de l'analyse syntaxique de la chaîne $nb_3 + nb_4 * nb_2$, décrit les différentes opérations (shift/reduce) effectuées.

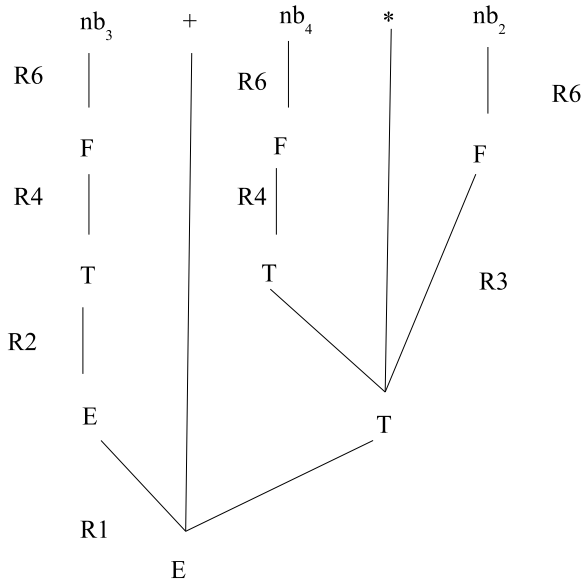


FIGURE 4.16 – arbre syntaxique de la chaîne $nb_3 + nb_4 * nb_2$

Remarque 11 *La grammaire permet de générer des expressions arithmétiques basées sur la multiplication et sur l'addition en respectant l'ordre des priorités croissant entre +,*,(). En effet, les terminaux apparaissent dans la grammaire dans le sens inverse de leur priorité.*

5.2.3 Conclusion

La méthode d'analyse SLR(1) est une méthode optimale car elle ne fait qu'intervenir les items LR(0) pour décider de l'action à exécuter. Elle est aussi facile à implémenter. Néanmoins, peu de grammaires sont SLR(1). La méthode d'analyse *LALR(1)* est une optimisation de la méthode LR(1).

5.3 Analyseur LALR(1)

La méthode d'analyse LALR(1) (Look Ahead LR(1)) est une méthode intermédiaire entre l'analyseur LR(1) et l'analyseur SLR(1). Elle représente un bon compromis entre la méthode d'analyse LR(1) et la méthode d'analyse SLR(1). En effet,

— elle est plus puissante que la méthode d'analyse SLR(1) car elle prend en charge une classe de grammaire plus importante,

— elle produit des tables d'analyse moins volumineuses que celles produites par la méthode d'analyse LR(1).

La méthode LALR(1) est basée sur le calcul des items LR(1). Afin de réduire le nombre des états générés, l'idée consiste à regrouper les ensembles des items qui ont le même cœur afin de former un ensemble constitué de l'union des deux ensembles de base avec des symboles de prévision différents. Cette opération induit la réduction de la table d'analyse LR(1).

Remarque 12 *le cœur d'un item LR(1) est représenté par la partie gauche qui se trouve avant la virgule :*

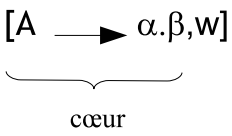


FIGURE 4.17 – cœur d'un item LR(1)

La construction de la table d'analyse LALR(1) associée à une grammaire G est réalisée en appliquant l'algorithme 26.

début
  Construire la collection C des items LR(1 : C={$I_0, I_1, ..., I_n$} ;
  pour *chaque ensemble d'items* $I_i \in C$ **faire**
    Chercher s'il existe dans la collection C un ensemble d'items I_j tels que tous les items ont le même cœur que ceux de l'ensemble I_i ;
    Remplacer les deux ensembles d'items par un ensemble d'items I_{ij} formé par l'union des ensemble d'items I_i et I_j ;
  fin
  Soit C' la nouvelle collection des ensembles des items ;
  Construire la table d'analyse LALR(1) à partir de la nouvelle collection des items C' ;
  si *(la table d'analyse LALR(1) est mono-définie)* **alors**
    la grammaire G est LALR(1) $\Longrightarrow$ G est LALR(k) avec K $\geq$ 1
  fin
fin

Algorithme 22 : Construction de la table d'analyse LALR(1)

Remarque 13 *Les générateurs des analyseurs syntaxiques, tel que Bison, se basent sur la méthode d'analyse LALR(1).*

Exemple 25 *Soit G la grammaire définie par <T,N,S,P> où T={c,d} ; N={S,C} ; et P :*

$$Soit \quad G = \begin{cases} Z \rightarrow S\# \\ S \rightarrow CC(1) \\ C \rightarrow cC(2) \mid d(3) \end{cases}$$

La grammaire G est-elle LALR(1) ?

1. *Construction des ensembles d'items LR(1) :*
   I_0 = { [Z → .S,#] [S → .CC,#] [C → .cC,c|d] [C → .d,c|d] }
   I_1 = GOTO(I_0,S) ={ [Z → S.,#] }
   I_2 = GOTO(I_0,C) ={ [S → C.C,#] [C → .cC,#] [C → .d,#] }
   I_3 = GOTO(I_0,c) ={ [C → c.C,c|d] [C → .cC,c|d] [C → .d,c|d] }
   I_4 = GOTO(I_0,d) ={ [C → d.,c|d] }
   I_5 = GOTO(I_2,C) ={ [S → CC.,#] }
   I_6 = GOTO(I_2, c) ={ [C → c.C,#] [C → .cC,#] [C → .d,#] }
   I_7 = GOTO(I_2,d) ={ [C → d.,#] }
   I_8 = GOTO(I_3,C) ={ [C → cC.,c|d] }
   I_4 = GOTO(I_3,d)
   I_3 = GOTO(I_3,c)
   I_9 = GOTO(I_6,C) ={ [C → cC.,#] }
   I_6 = GOTO(I_6,c)
   I_7 = GOTO(I_6,d)
   Ainsi la collection C des items LR(1) est :
   C={I_0,I_1,I_2,I_3,I_4,I_5,I_6,I_7,I_8,I_9}.
2. *Construction de la table d'analyse LR(1) :*
   La table d'analyse LR(1) est mono-définie ⇒ la grammaire G est LR(1).

	c	d	#	S	C
0	D,3	D,4		1	2
1			ACCEPT		
2	D,6	D,7			5
3	D,3	D,4			8
4	R3	R3			
5			R1		
6	D,6	D,7			9
7			R3		
8	R2	R2			
9			R2		

3. *Construction de la nouvelle collection des items C' :*
   $I_{3'} = I_3 \cup I_6$=*{[C → c.C,c| d|#] [C → .cC,c|d|#] [C → .d,c|d|#]}*
   $I_{4'} = I_4 \cup I_7$=*{[C → d.,c|d| #]}*
   $I_{6'} = I_8 \cup I_9$=*{[C → cC.,c|d|#]}*
   D'où la nouvelle collectionn des items C'=$\{I_0,I_1,I_2,I_{3'},I_{4'},I_5,I_{6'}\}$

Remarque 14 *les ensembles d'items qui ne sont pas concernés par la fusion restent inchangés.*

4. *Construction de la table d'analyse LALR(1) :*

	c	d	#	S	C
0	D,3'	D,4'		1	2
1			ACCEPT		
2	D,3'	D,4'			5
3'	D,3'	D,4'			6'
4'	R3	R3	R3		
5			R1		
6'	R2	R2	R2		

La table d'analyse LALR(1) est mono-définie ⇒ la grammaire G est LALR(1)⇒ la grammaire G est LALR(k), k ≥ 1.

Remarque 15 *La table d'analyse LALR(1) est obtenue en superposant les lignes de la table d'analyse LR(1) qui correspondent aux ensembles d'items ayant le même cœur.*

Soit à analyser la chaine **ccdd# :**

Pile	Chaine	Action
#⓪	ccdd#	D,3’
#⓪c③’	cdd#	D,3’
#⓪c③’c③’	dd#	D,4’
#⓪c③’c③’*d*④’	d#	R3 : C → d
#⓪c③’*c*③’*C*⑥’	d#	R2 : C → cC
#⓪*c*③’*C*⑥’	d#	R2 : C → cC
#⓪C②’	d#	D,4’
#⓪C②’*d*④’	#	R3 : C → d
#⓪*C*②’*C*⑤’	#	R1 : S → CC
#⓪S①	#	ACCEPT

Exemple 26 *Soit G la grammaire définie par <T,N,S,P> où T={a,b,c,d,e} ; N={S,A,B,C} ; et P :*

$$Soit \quad G = \begin{cases} S \rightarrow A \\ A \rightarrow bB \mid a \\ B \rightarrow cC \mid cCe \\ C \rightarrow dAf \end{cases}$$

La grammaire augmentée associée à G est obtenue en rajoutant à G la règle Z→ S# :

$$Soit \quad G = \begin{cases} Z \rightarrow S\# \\ S \rightarrow A① \\ A \rightarrow bB② \mid a③ \\ B \rightarrow cC④ \mid cCe⑤ \\ C \rightarrow dAf⑥ \end{cases}$$

Les ensembles des débuts et des suivants associés aux non terminaux de la grammaire G sont donnés par la Table suivante :

	debuts	suivants
S	a b	#
A	a b	f #
B	c	f #
C	d	f e #

La grammaire G est-elle LALR(1) ?

1. *Construction des ensembles d'items LR(1) :*
   I_0 = *{[Z → .S,#] [S → .A,#] [A → .bB,#] [A → .a,#]}*
   I_1 = *GOTO(I_0,S) ={[Z → S.,#]}*
   I_2 = *GOTO(I_0,A) ={[S → A.,#]}*
   I_3 = *GOTO(I_0,b) ={[A → b.B,#] [B → .cC,#] [B → .cCe,#]}*
   I_4 = *GOTO(I_0,a) ={[A → a.,#]}*
   I_5 = *GOTO(I_3,B) ={[B → bB.,#]}*
   I_6 = *GOTO(I_3, c) ={[B → c.C,#] [B → .cCe,#] [C → .dAf,e|#]}*
   I_7 = *GOTO(I_6,C) ={[B → cC.,#] [B → cC.e,#]}*
   I_8 = *GOTO(I_6,d) ={[C → d.Af,e|#] [A → .bB,f] [A → .a,f]}*
   I_9 = *GOTO(I_7,e)={[B → cCe.,#]}*
   I_{10} = *GOTO(I_8,A) ={[C → dA.f,e|#]}*
   I_{11} = *GOTO(I_8,b) ={[A → b.B,f] [B → .cC,f] [B → .cCe,f]}*
   I_{12} = *GOTO(I_8,a) ={[A → a.,f]}*
   I_{13} = *GOTO(I_{10},f)={[C → dAf.,#]}*
   I_{14} = *GOTO(I_{11},B) ={[B → bB.,f]}*
   I_{15} = *GOTO(I_{11}, c) ={[B → c.C,f] [B → c.Ce,f] [C → .dAf,e|f]}*
   I_{16} = *GOTO(I_{15},C) ={[B → cC.,f] [B → cC.e,f]}*
   I_{17} = *GOTO(I_{15},d) ={[C → d.Af,e|f] [A → .bB,f] [A → .a,f]}*
   I_{18} = *GOTO(I_{16},e)={[B → cCe.,f]}*
   I_{19} = *GOTO(I_{17},A)={[C → dA.f,e|f]}*
   I_{11} = *GOTO(I_{17},b)*
   I_{12} = *GOTO(I_{17},a)*
   I_{20} = *GOTO(I_{19},f)={[C → dAf.,e|f]}*
   D'où C={I_0,I_1,I_2,I_3,I_4,I_5,I_6,I_7,I_8,I_9,I_{10},I_{11},I_{12},I_{13},I_{14},I_{15},I_{16},I_{17},I_{18},I_{19},I_{20}}

2. *Construction de la table d'analyse LR(1) :*

États	a	b	c	d	e	f	#	S	A	B	C
0	D,4	D,3						1	2		
1							ACCEPT				
2							R1				
3			D,6							5	
4							R3				
5							R2				
6				D,8							7
7					D,9		R4				
8	D,12	D,13							10		
9							R5				
10						D,13					
11			D,15							14	
12						R3					
13					R6		R6				
14						R2					
15				D,17							16
16					D,18	R4					
17	D,12	D,11							19		
18						R5					
19						D,20					
20					R6	R6					

La table d'analyse LR(1) est mono-définie ⇒ la grammaire G est LR(1)⇒ la grammaire G est LR(k), k ≥ 1.

3. *Construction de la nouvelle collection des items C' en fusionnant les ensembles d'items ayant les mêmes cœurs :*
   $I_{0'} = I_0$; $I_{1'} = I_1$; $I_{2'} = I_2$;
   $I_{3'} = I_3 \cup I_{11}$=GOTO(I_0,b) ={[A → b.B,f |#] [B → .cC,f |#] [B → .cCe,f |#]}
   $I_{4'} = I_4 \cup I_{12}$=GOTO(I_0,a) ={[A → a.,f,|#]}
   $I_{5'} = I_5 \cup I_{14}$=GOTO($I_{3'}$,B) ={[B → bB.,f|#]}
   $I_{6'} = I_6 \cup I_{15}$=GOTO($I_{3'}$, c)={[B → c.C,f|#] [B → .cCe,f|#] [C → .dAf,e|f|#]}
   $I_{7'} = I_7 \cup I_{16}$= GOTO($I_{6'}$,C) ={[B → cC.,f|#] [B → cC.e,|#]}
   $I_{8'} = I_8 \cup I_{17}$=GOTO($I_{6'}$,d) ={[C → d.Af,e|f|#] [A → .bB,f|#] [A → .a,f|#]}
   $I_{9'} = I_9 \cup I_{18}$= GOTO($I_{7'}$,e)={[B → cCe.,f|#]}
   $I_{10'} = I_{10} \cup I_{19}$= GOTO($I_{8'}$,A)={[C → dA.f,e|f|#]}
   $I_{11'} = I_{13} \cup I_{20}$=GOTO($I_{10'}$,f)={[C → dAf.,e|f|#]}
   D'où C'={$I_{0'}$,$I_{1'}$,$I_{2'}$,$I_{3'}$,$I_{4'}$,$I_{5'}$,$I_{6'}$,$I_{7'}$,$I_{8'}$,$I_{9'}$,$I_{10'}$,$I_{11'}$}
4. *Construction de la table d'analyse LALR(1) :*

États	a	b	c	d	e	f	#	S	A	B	C
0’	D,4’	D,3’						1	2		
1’							ACCEPT				
2’							R1				
3’			D,6’							5’	
4’						R3	R3				
5’						R2	R2				
6’				D,8’							7’
7’					D,9’	R4	R4				
8’	D,4’	D,3’							10’		
9’						R5	R5				
10’						D,11’					
11’					R6	R6	R6				

La table d’analyse LALR(1) est mono-définie ⇒ la grammaire G est LALR(1)⇒ la grammaire G est LALR(k), k ≥ 1.

Remarque 16 *Une grammaire G peut être LR(1) mais non LALR(1).*

Exemple 27 *Soit G la grammaire définie par <T,N,S,P> où T={a,b,c,d,e} ; N={S,A,B} ; et P :*

$$Soit \quad G = \begin{cases} S \rightarrow aAd \mid bBd \mid aBe \mid bAe \\ A \rightarrow c \\ B \rightarrow c \end{cases}$$

La grammaire augmentée associée à G est obtenue en rajoutant à G la règle Z→ S# :

$$Soit \quad G = \begin{cases} Z \rightarrow S\# \\ S \rightarrow aAd\textcircled{1} \mid bBd\textcircled{2} \mid aBe\textcircled{3} \mid bAe\textcircled{4} \\ A \rightarrow c\textcircled{5} \\ B \rightarrow c\textcircled{6} \end{cases}$$

Les ensembles des débuts et des suivants associés aux non terminaux de la grammaire G sont donnés par la Table suivante :

	debuts	suivants
S	a b	#
A	c	d e
B	c	d e

La grammaire G est-elle LR(1) ?

1. *Construction des ensembles d'items LR(1) :*
   I_0 = *{[Z → .S,#] [S → .A,#] [S → .aAd,#] [S → .bBd,#][S → .aBe,#][S → .bAe,#]}*
   I_1 = *GOTO(I_0,S) ={[Z → S.,#]}*
   I_2 = *GOTO(I_0,a) ={[S → a.Ad,#][S → a.Be,#][A → .c,d][B → .c,e]}*
   I_3 = *GOTO(I_0,b) ={[S → b.Bd,#][S → b.Ae,#][B → .c,d][A → .c,e]}*
   I_4 = *GOTO(I_2,A) ={[S → aA.d,#]}*
   I_5 = *GOTO(I_2,B) ={[S → aB.e,#]}*
   I_6 = *GOTO(I_2, c) ={[A → c.,d] [B → c.,e]}*
   I_7 = *GOTO(I_3,B) ={[S → bB.d,#]}*
   I_8 = *GOTO(I_3,A) ={[S → bA.e,#]}*
   I_9 = *GOTO(I_3,c)={[B → c.,d][A → c.,e]}*
   I_{10} = *GOTO(I_4,d) ={[S → aAd.,#]}*
   I_{11} = *GOTO(I_5,e) ={[S → aBe.,#]}*
   I_{12} = *GOTO(I_7,d) ={[S → bBd.,#]}*
   I_{13} = *GOTO(I_8,e)={[S → bAe.,#]}*
   D'où C={I_0,I_1,I_2,I_3,I_4,I_5,I_6,I_7,I_8,I_9,I_{10},I_{11},I_{12},I_{13}}

2. *Construction de la table d'analyse LR(1) :*

États	a	b	c	d	e	#	S	A	B
0	D,2	D,3					1		
1						ACCEPT			
2			D,6					4	5
3			D,9					8	7
4				D,10					
5					D,11				
6				R5	R6				
7				D,12					
8					D,13				
9				R5	R6				
10						R1			
11						R3			
12						R2			
13						R4			

La table d'analyse LR(1) est mono-définie ⇒ la grammaire G est LR(1)⇒ la grammaire G est LR(k), k ≥ 1.

3. *Construction de la nouvelle collection des items C' en fusionnant les ensembles d'items ayant les mêmes cœurs :*
   $I_{69} = I_6 \cup I_9$=*{[A → c.,d|e] [B → c.,e|d]}*
   Ainsi, D'où C'={I_0,I_1,I_2,I_3,I_4,I_5,I_{66},I_7,I_8,I_{10},I_{11},I_{12},I_{13}*}*

4. *Construction de la table d'analyse LALR(1) :*

États	a	b	c	d	e	#	S	A	B
0	D,2	D,3					1		
1						ACCEPT			
2			D,69					4	5
3			D,69					8	7
4				D,10					
5					D,11				
69				R6 / R5	R6 / R5				
7				D,12					
8					D,13				
10						R1			
11						R3			
12						R2			
13						R4			

La table d'analyse LALR(1) est multi-définie ⇒ la grammaire G est non LALR(1).

Remarque 17 *Les deux cas de multi-définitions peuvent être détectés directement à partir de l'ensemble des items I_{69}. En effet,*

— *l'item [A → c.,d] ∈ I_{69} ⇒ M[I_{69},d] := R5 et l'item [B → c.,d] ∈ I_{69} ⇒ M[I_{69},d] := R6,*

— *l'item [A → c.,e] ∈ I_{69} ⇒ M[I_{69},e] := R5 et l'item [B → c.,e] ∈ I_{69} ⇒ M[I_{69},e] := R6.*

5.4 Conclusion

La méthode SLR est une méthode optimale car elle ne fait qu'intervenir les item LR(0) pour décider de l'action à exécuter. Elle est aussi facile à implémenter. Néanmoins, peu de grammaires sont SLR(1). Les méthodes LR(1)et LALR(1) sont plus générales. les relations qui existent entre les trois types d'analyseurs syntaxiques ascendants sont illustrées par la Figure suivante :

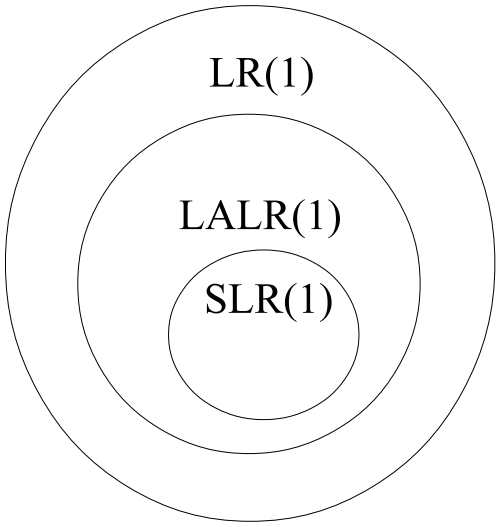


FIGURE 4.18 – Les relations entre les grammaires LR(1), LALR(1) et SLR(1)

Les relations suivantes sont déduites :

— Si une Grammaire G est SLR(1) ⇒ G est LALR(1) ⇒ G est LR(1),

— Si une Grammaire G est non LR(1) ⇒ G est non LALR(1) ⇒ G est non SLR(1).

5.5 Gestion des conflits dans les grammaires ambiguës

Les grammaires ambiguës provoquent des conflits qui induisent des tables multi-définie. Pour les méthodes ascendantes, les conflits sont de deux types :

1. conflits décalage/réduction :
   $T[I_i,a]=D_{IJ}$
   $T[I_i,a]=R_{A\rightarrow\alpha}$
   Dans ce cas, à la lecture du terminal a, deux actions sont possibles :
   — réduire α à A en utilisant la règle $\mathbf{A} \rightarrow \alpha$,
   — décaler le terminal a.
2. conflits réduction/réduction :
   $T[I_i,a]=R_{A\rightarrow\alpha}$
   $T[I_i,a]=R_{B\rightarrow\beta}$

   Dans ce cas, à la lecture du terminal a deux actions sont possibles :

— réduireα à A en utilisant la production $\mathbf{A} \rightarrow \alpha$,

— réduire β à B en utilisant la production $\mathbf{B} \rightarrow \beta$.

Il existe deux solutions afin de résoudre ces conflits :

— attribuer des priorités aux actions (décaler/réduire)ou exploiter les propriétés d'associativité pour les grammaires faisant intervenir des opérateurs dans le cas de conflits décalage/réduction,

— imposer des priorités aux règles de productions pour les cas de conflits réduction/réduction.

Remarque 18 *les cas de conflit décalage/décalage ne peut se produire.*

Exemple 28 (Conflit Décalage-Réduction) *Soit G la grammaire définie par <T,N,E,P> où T={+,*,/,-,nb} ; N={E} ; et P :*

$$E \rightarrow E+E \mid E*E \mid E/E \mid E-E \mid (E) \mid nb$$

La grammaire augmentée associée à G est comme suit : :

$$\begin{cases} Z \rightarrow E\# \\ E \rightarrow E+E\textcircled{1} \\ E \rightarrow E*E\textcircled{2} \\ E \rightarrow E/E\textcircled{3} \\ E \rightarrow E-E\textcircled{4} \\ E \rightarrow (E)\textcircled{5} \\ E \rightarrow nb\textcircled{6} \end{cases}$$

Les ensembles des débuts et des suivants de la grammaire G sont donnés par la table suivante :

TABLE 4.15 – Les débuts et les suivants associés aux non terminaux de la grammaire G

	débuts	suivants
E	(nb	+ * / -) #

TABLE 4.16 – Les débuts et les suivants associés aux non terminaux de la grammaire G

Afin de vérifier si la grammaire G est SLR(1), il faut :

1. *construire la collection des items LR(0) :*
   I_0 = { [Z → .E] [E → .E+E] [E → .E*E] [E → .E/E] [E → .E-E] [E → (E)][E → .nb] }
   I_1 = GOTO(I_0, E) ={[Z → E.] [E → E.+E] [E → E.*E] [E → E./E] [E → E.-E]}
   I_2 = GOTO(I_0, () ={[E → (.E)] [E → .E+E] [E → .E*E] [E → .E/E] [E → .E-E] [E → .(E)][E → .nb]}
   I_3 = GOTO(I_0, nb) ={[E → nb.]}
   I_4 = GOTO(I_1, +) ={[E → E+.E] [E → .E+E] [E → .E*E] [E → .E/E] [E → .E-E] [E → (E)][E → .nb]}
   I_5 = GOTO(I_1,*) ={[E → E*.E] [E → .E+E] [E → .E*E] [E → .E/E] [E → .E-E] [E → (E)][E → .nb]}
   I_6 = GOTO(I_1,/) ={[E → E/.E] [E → .E+E] [E → .E*E] [E → .E/E] [E → .E-E] [E → (E)][E → .nb]}
   I_7 = GOTO(I_1,-) ={[E → E-.E] [E → .E+E] [E → .E*E] [E → .E/E] [E → .E-E] [E → (E)][E → .nb]}
   I_8 = GOTO(I_2, E)={[E → (E.)] [E → E.+E] [E → E.*E] [E → E./E] [E → E.-E]}
   I_2 = GOTO(I_2,()
   I_3 = GOTO(I_2,nb)
   I_9 = GOTO(I_4, E) ={[E → E+E.] [E → E.+E] [E → E.*E] [E → E./E] [E → E.-E]}
   I_2 = GOTO(I_4, ()
   I_3 = GOTO(I_4, nb)
   I_{10} = GOTO(I_5, E) ={[E → E*E.] [E → E.+E] [E → E.*E] [E → E./E] [E → E.-E]}
   I_2 = GOTO(I_5,()
   I_3 = GOTO(I_5,nb)
   I_{11} = GOTO(I_6, E) ={[E → E/E.] [E → E.+E] [E → E.*E] [E → E./E] [E → E.-E]}
   I_2 = GOTO(I_6,()
   I_3 = GOTO(I_6,nb)
   I_{12} = GOTO(I_7, E) ={[E → E-E.] [E → E.+E] [E → E.*E] [E → E./E] [E → E.-E]}
   I_2 = GOTO(I_7,()
   I_3 = GOTO(I_7,nb)
   I_{13} = GOTO(I_8,))={[E → (E).]}
   I_4 = GOTO(I_8,+)
   I_5 = GOTO(I_8,*)
   I_6 = GOTO(I_8,/)

$I_7 = GOTO(I_8,\text{-})$
$I_4 = GOTO(I_9,+)$
$I_5 = GOTO(I_9,*)$
$I_6 = GOTO(I_9,/)$
$I_7 = GOTO(I_9,\text{-})$
$I_4 = GOTO(I_{10},+)$
$I_5 = GOTO(I_{10},*)$
$I_6 = GOTO(I_{10},/)$
$I_7 = GOTO(I_{10},\text{-})$
$I_4 = GOTO(I_{11},+)$
$I_5 = GOTO(I_{11},*)$
$I_6 = GOTO(I_{11},/)$
$I_7 = GOTO(I_{11},\text{-})$
$I_4 = GOTO(I_{12},+)$
$I_5 = GOTO(I_{12},*)$
$I_6 = GOTO(I_{12},/)$
$I_7 = GOTO(I_{12},\text{-})$

2. *construire la table d'analyse SLR(1) à partir de la collection des items LR(0) :*

Etat	+	*	/	-	(	)	nb	#	E
0					D,2		D,3		1
1	D,4	D,5	D,6	D,7				Accept	
2					D,2		D,3		8
3	R6	R6	R6	R6		R6		R6	
4					D,2		D,3		9
5					D,2		D,3		10
6					D,2		D,3		11
7					D,2		D,3		12
8	D,4	D,5	D,6	D,7		D,13			
9	R1 / D,4	R1 / D,5	R1 / D,6	R1 / D,7		R1		R1	
10	R2 / D,4	R2 / D,5	R2 / D,6	R2 / D,7		R2		R2	
11	R3 / D,4	R3 / D,5	R3 / D,6	R3 / D,7		R3		R3	
12	R4 / D,4	R4 / D,5	R4 / D,6	R4 / D,7		R4		R4	
13	R5	R5	R5	R5		R5		R5	

3. *La table d'analyse SLR(1) est multi-définie ⇒ G n'est pas SLR(1).*
4. *Comme la grammaire G est une grammaire d'opérateurs qui génère des expressions arithmétiques, + * / -, il est possible de résoudre les cas de conflits :*
   - *en faisant intervenir les priorités entre les opérateurs arithmétiques tels que / et * sont plus prioritaires que + et -,*
   - *en appliquant le critère d'associativité de gauche à droite.*

   Les différents cas de conflits présents dans la table SLR(1) de la grammaire G vont être résolus en sélectionnant :
   - *l'opération relative à l'opérateur le plus prioritaire dans le cas de conflits entre opérateurs ayant des priorités différentes,*
   - *la réduction dans le cas de l'application du critère d'associativité de gauche à droite pour les cas de conflits impliquant les mêmes opérateurs ou des opérateurs ayant la même priorité.*
   - *le décalage dans le cas de l'application du critère d'associativité de droite à gauche pour les cas de conflits impliquant les mêmes opérateurs ou des opérateurs ayant la même priorité.*

ces conventions permettent ainsi de résoudre les cas de conflits induits par la grammaire G comme suit :

- *cas de M[I_9,*]=D,5 et M[I_9,*]=R1*
  *Il correspond au conflit nb+nb*nb impliquant l'opération d'addition suivie de l'opération de multiplication. A la lecture du second opérateur *, deux actions sont envisageables :*
  - *soit avancer afin d'évaluer d'abord l'opération de multiplication (D,5),*

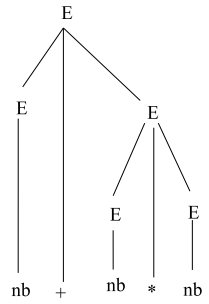


FIGURE 4.19 – cas de décalage

  - *ou encore évaluer en premier la première opération d'addition en effectuant la réduction (R1).*

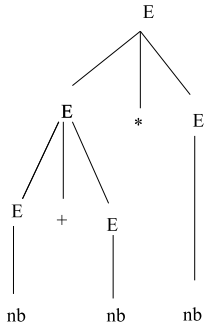


FIGURE 4.20 – cas de réduction

Comme l'opérateur de multiplication est plus prioritaire que l'opération d'addition, l'action de décalage D,5 sera sélectionnée.

- *cas de M[I_9,+]=D,4 et M[I_9,+]=R1*
  Il correspond au conflit nb+nb+nb impliquant deux opérations d'addition. A la lecture du second opérateur +, deux actions sont envisageables : soit avancer afin d'évaluer d'abord la seconde opération d'addition (D,4) ou encore évaluer en premier la première opération en effectuant la réduction (R1). Comme l'opérateur d'addition est associatif de gauche à droite, l'action de réduction R1 sera sélectionnée.
- *cas de M[I_9,/]=D,6 et M[I_9,/]=R1*
  Il correspond au conflit nb+nb/nb impliquant l'opération d'addition suivie de l'opération de division. A la lecture du second opérateur /, deux actions sont envisageables : soit avancer afin d'évaluer d'abord l'opération de division (D,6) ou encore évaluer en premier la première opération d'addition en effectuant la réduction (R1). Comme l'opérateur de division est plus prioritaire que l'opération d'addition, l'action de décalage D,6 sera sélectionnée.
- *cas de M[I_9,-]=D,7 et M[I_9,-]=R1*
  Il correspond au conflit nb+nb-nb impliquant l'opération d'addition suivie de l'opération de soustraction. A la lecture de l'opérateur -, deux actions sont envisageables : soit avancer afin d'évaluer d'abord l'opération de soustraction (D,7) ou encore évaluer en premier lieu l'opération d'addition en effectuant la réduction (R1). Comme les opérations d'addition et de soustraction ont la même priorité, le critère d'associativité de gauche à droite sera appliqué. L'action de réduction R1 sera alors sélectionnée.
- *cas de M[I_{10},+]=D,4 et M[I_{10},+]=R2*
  *Il correspond au conflit nb*nb+nb impliquant l'opération de multiplication suivie de l'opération d'addition. A la lecture de l'opérateur +, deux actions sont envisageables : soit avancer afin d'évaluer d'abord l'opération d'addition (D,4) ou encore évaluer en premier lieu l'opération de multiplication en effectuant la réduction (R2). Comme l'opérateur de multiplication est plus prioritaire que l'opérateur d'addition, l'action de réduction R2 sera alors sélectionnée.*
- *cas de M[I_{10},*]=D,5 et M[I_{10},*]=R2*
  *Il correspond au conflit nb*nb*nb impliquant deux opérations de multiplication. A la lecture du second opérateur *, deux actions sont envisageables : soit avancer afin d'évaluer d'abord la seconde opération de multiplication (D,4) ou encore évaluer en premier la première opération en effectuant la réduction (R1). Comme l'opérateur de multiplication est associatif de gauche à droite, l'action de réduction R2 sera*

sélectionnée.

- *cas de M[I_{10},/]=D,6 et M[I_{10},/]=R2*
  *Il correspond au conflit nb*nb/nb impliquant l'opération de multiplication suivie de l'opération de division. A la lecture de l'opérateur /, deux actions sont envisageables : soit avancer afin d'évaluer d'abord l'opération de division(D,6) ou encore évaluer en premier lieu l'opération de multiplication en effectuant la réduction (R2). Comme les opérations de multiplication et de division ont la même priorité, le critère d'associativité de gauche à droite sera appliqué. L'action de réduction R2 sera alors sélectionnée.*
- *cas de M[I_{10},-]=D,7 et M[I_{10},-]=R2*
  *Il correspond au conflit nb*nb-nb impliquant l'opération de multiplication suivie de l'opération de soustraction. A la lecture de l'opérateur -, deux actions sont envisageables : soit avancer afin d'évaluer d'abord l'opération de soustraction (D,7) ou encore évaluer en premier lieu l'opération de multiplication en effectuant la réduction (R2). Comme l'opérateur de multiplication est plus prioritaire que l'opérateur de soustraction, l'action de réduction R2 sera alors sélectionnée.*

Les autres cas de conflits seront résolus de la même manière L'opérateur de soustraction se comporte comme l'opérateur d'addition et l'opérateur de division se comporte comme l'opérateur de multiplication.
La table d'analyse SLR(1) mono-définie qui en résulte après la phase de résolution de conflits est la suivante :

État	+	*	/	-	(	)	nb	#	E
0					D,2		D,3		1
1	D,4	D,5	D,6	D,7				ACCEPT	
2					D,2		D,3		8
3	R6	R6	R6	R6		R6		R6	
4					D,2		D,3		9
5					D,2		D,3		10
6					D,2		D,3		11
7					D,2		D,3		12
8	D,4	D,5	D,6	D,7		D,13			
9	R1	D,5	D,6	R1		R1		R1	
10	R2	R2	R2	R2		R2		R2	
11	R3	R3	R3	R3		R3		R3	
12	R4	D,5	D,6	R4		R4		R4	
13	R5	R5	R5	R5		R5		R5	

Remarque 19 *La philosophie des analyseurs LR(k) repose sur le concept que pour décider de l'action à exécuter, il est nécessaire de consulter k entités lexicales supplémentaires. En pratique, seuls les analyseurs LR(1) sont automatisables. En effet, lorsque k≥2, la table d'analyse devient volumineuse et le temps d'analyse va augmenter considérablement.*

Remarque 20 *Dans le cas de grammaire d'opérateurs (arithmétiques, booléens,...), il est possible de lever les cas de multi-définitions :*

— *en faisant intervenir les priorités entre les opérateurs dans l'écriture de la grammaire. Ainsi, les opérateurs vont apparaître dans la grammaire dans le sens inverse de leurs priorités.*
A titre d'exemple, la grammaire G' définie par <T,N',E,P'> où T={+,,(,),nb} N'={E,T,F} et P' :*

$$\begin{cases} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid nb \end{cases}$$

est une grammaire équivalente à la grammaire G définie par <T,N,E,P> où T={+,,(,),nb} N={E} et P :*

$$E \rightarrow E + E \mid E * E \mid E/E \mid (E) \mid nb$$

En effet, les opérateurs arithmétiques apparaissent dans la grammaires dans le sens inverse de l'ordre de priorité défini par dans l'ordre décroissant comme suit : (),,+.*

— *en appliquant les critères d'associativité pour lorsque le conflit est induit par les mêmes opérateurs ou par des opérateurs ayant des priorités identiques (tels que l'addition et la soustraction).*
A titre d'exemple, la grammaire G' définie par <T,N',E,P'> où T={+,,(,),nb} N'={E,T,F} et P' :*

$$\begin{cases} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid nb \end{cases}$$

est une grammaire pour laquelle le critère d'associativité est de gauche à droite.
Pour la grammaire G" définie par <T,N',E,P"> où T={+,,(,),nb} N'={E,T,F} et P" :*

$$\begin{cases} E \rightarrow T + E \mid T \\ T \rightarrow F * T \mid F \\ F \rightarrow (E) \mid nb \end{cases}$$

est une grammaire pour laquelle le critère d'associativité est de droite à gauche.

Exemple 29 (Conflits réduction/réduction) *Soit G la grammaire définie par <T,N,E,P> où T={a,b,v} ; N={E,A,V} ; et P :*

$$\begin{cases} E \rightarrow AV\textcircled{1} \mid A\textcircled{2} \\ A \rightarrow a\textcircled{3} \mid b\textcircled{4} \\ V \rightarrow v\textcircled{5} \mid \epsilon\textcircled{6} \end{cases}$$

La grammaire augmentée associée à G est obtenue en rajoutant la règle Z→ E# :

$$\begin{cases} Z \rightarrow E\# \\ E \rightarrow AV\textcircled{1} \mid A\textcircled{2} \\ A \rightarrow a\textcircled{3} \mid b\textcircled{4} \\ V \rightarrow v\textcircled{5} \mid \epsilon\textcircled{6} \end{cases}$$

Les ensembles des débuts et des suivants de la grammaire G sont donnés par la table :

	debuts	suivants
E	a,b	#
A	a,b	v,#
V	v,ϵ	#

La grammaire G est-elle SLR(1) ?

1. *Calcul de la collection des items LR(0)*
   I_0 = *{[Z → .E] [E → .AV] [E → .A] [A → .a] [A → .b]}*
   I_1 = *GOTO(I_0, E) ={[Z → E.]}*
   I_2 = *GOTO(I_0, T) ={[E → A.V]*$\underline{[E \rightarrow A.]}$ *[V → .v]* $\underline{[V \rightarrow .]}$*}*
   I_3 = *GOTO(I_0, a) ={[A → a.]}*
   I_4 = *GOTO(I_0,b) ={[A → b.]}*
   I_5 = *GOTO(I_2, V) ={[E → AV.]}*
   I_6 = *GOTO(I_2,v) ={[V → v.]}*

2. *Construction de la table d'analyse SLR(1)*

Etats	a	b	v	#	E	A	V
0	D,3	D,4			1	2	
1				ACCEPT			
2			D,6	R2 / R6			5
3		R3	R3				
4		R4	R4				
5				R1			
6				R5			

La table d'analyse SLR(1) est multi-définie ⇒ G est non SLR(1)⇒.
En effet,

— *l'item* $[E \rightarrow A.] \in I_2 \Rightarrow M[I_2,\#] := R2$,

— *l'item* $[V \rightarrow .v] \in I_2 \Rightarrow M[I_2,\#] := R6$.

Afin de résoudre ce type de conflit, il faudrait associer des priorités aux règles de production. Si l'ordre d'apparition des règles dans la grammaire est significatif alors la règle R2 sera retenue.

5.6 Gestion des erreurs syntaxiques

La plupart des erreurs de compilation sont générées lors de l'analyse syntaxique. Le gestionnaire des erreurs doit :

— indiquer les détails de l'erreur afin de permettre au programmeur de la corriger,

— faire un recouvrement des erreurs en les traitant afin de pouvoir poursuivre le processus de l'analyse et de ne pas créer de nouvelles erreurs.

Lorsque le nombre d'erreurs devient important, le processus de compilation est alors interrompu car le recouvrement devient impossible.
Il existe plusieurs stratégies de traitement des erreurs syntaxiques :

1. Récupération d'erreurs en mode panique
   C'est la méthode la plus simple à implémenter. Lorsqu'il y a une erreur, l'analyseur élimine les symboles jusqu'à en rencontrer une entité lexicale qui appartient à un ensemble précis d'entités lexicales dites de synchronisation représentées par certains séparateurs ou mots clés telles que ; } end.
2. Récupération d'erreurs au niveau syntaxique
   Dans ce cas, à la rencontre d'une erreur, l'analyseur peut effectuer des corrections locales, telles que le remplacement de **,** par **;** ou encore **wile** par **while**, afin de permettre à l'analyse de se poursuivre.

La gestion des erreurs syntaxique est matérialisée par la mise en œuvre de routines qui seront exécutées à la rencontre des erreurs. Ces routines vont opérer un certain nombre d'actions selon la politique utilisée. Afin de permettre l'exécution des routines, des pointeurs vers les routines seront insérées au niveau des cases vides des tables d'analyse.

Exemple 30 *Considérons la grammaire G de l'exemple 24 définie par <T,N,E,P> où T={+ * ( ) nb} ; N={E,T,F} ; et P :*

$$\begin{cases} E \rightarrow E + T\textcircled{1} \mid T\textcircled{2} \\ T \rightarrow T * F\textcircled{3} \mid F\textcircled{4} \\ F \rightarrow (E)\textcircled{5} \mid nb\textcircled{6} \end{cases}$$

La table d'analyse SLR(1) associée à la grammaire G peut être complétée par les routines sémantiques comme suit :

États	nb	+	*	(	)	#	E	T	F
0	D,5	E1	E1	D,4	E1	E1	1	2	3
1	E2	D,6	E2	E2	E2	ACCEPT			
2	E3	R2	D,7	E3	R2	R2			
3	E3	R4	R4	E3	R4	R4			
4	D,5	E1	E1	D,4	E1	E1	8	2	3
5	E3	R6	R6	E3	R6	R6			
6	D,5	E1	E1	D,4	E1	E1		9	3
7	D,5	E1	E1	D,4	E1	E1			10
8	E4	D,6	E4	E4	D,11	E4			
9	E3	R1	D,7	E3	R1	R1			
10	E3	R3	R3	E3	R3	R3			
11	E3	R5	R5	E3	R5	R5			

TABLE 4.17 – Table d'analyse SLR(1) de la grammaire de l'exemple 24 complétée par les routines d'erreurs

Les routines d'erreurs sont définies par :

- *Routine E1 : L'analyseur syntaxique a détecté une des entités parmi '+,*,),#' alors que l'analyseur attend une opérande ou une parenthèse ouvrante : 'operande ou (expected'.*
- *Routine E2 : L'analyseur syntaxique a détecté une entité parmi 'opérande, *, (,)' alors que l'analyseur attend l'opérateur d'addition : '+' expected.*

— *Routine E3 : L'analyseur syntaxique a détecté une entité parmi 'opérande, (' alors que l'analyseur attend l'opérateur de multiplication ou l'opérateur d'addition ou) ou #.*

— *Routine E4 : L'analyseur syntaxique a détecté une entité parmi opérande, *, (, # alors que l'analyseur attend l'opération d'addition + ou une parenthèse fermante : '+ ou) expected'.*

6 Écriture des grammaires

Il n'est pas toujours facile de concevoir une grammaire pour décrire un langage. Il est alors conseillé de suivre une certaine méthodologie. Il faudrait :

— commencer par spécifier les constructions principales du langage. A titre d'exemple, pour les langages de programmations procéduraux, un programme est composé par :

1. la partie déclaration qui regroupe une liste de variables suivie par un type. Les types peuvent être simples (Entier, Réel,...) ou composés (tableau, liste, enregistrement,...),
2. la partie instructions qui regroupe une liste d'instruction telle que chaque instruction peut être une instruction d'affectation, une instruction conditionnelle, une instruction répétitive,...

— définir des règles générales et simples. Une grammaire décrit l'agencement des entités lexicales. Il ne faut alors pas l'encombrer avec des aspects textuels qui doivent être traités lors de la phase sémantique.

Exemple 4.2 *Dans un tableau, les indices doivent être de type entier. Des contrôles sémantiques doivent être réalisés lors de la déclaration d'un tableau (var tableau : array[1..n;1..m]) et lors de la référence à un élément du tableau (c :=tableau[i,j]).*

— définir des règles récursives dans le cas où une partie du langage doit se reproduire à plusieurs reprises comme c'est le cas de la déclaration des variables : liste variables : type, ou encore dans la partie du bloc d'instruction. Pour ce faire, les règles récursives doivent être utilisées.

Exemple 4.3 *Les règles syntaxiques suivantes décrivent le bloc de déclara-*

tion du langage Fortran :

$$\begin{cases} < declaration > \rightarrow VAR < liste - decl > \\ < liste - decl > \rightarrow < liste - decl >< declaration > | \epsilon \\ < declaration > \rightarrow < liste - var > :< type > ; | \epsilon \\ < liste - idf > \rightarrow IDENTIFICATEUR, < liste - idf > | IDENTIFICATEUR \\ < type > \rightarrow INTEGER | REAL | DOUBLEPRECISION | LOGICAL \end{cases}$$

Ces règle permettent ainsi de générer plusieurs instructions de déclarations telle que chaque déclaration peut englober une liste de variables.

Le choix entre l'utilisation des règles récursives droites ou gauches est guidé par les traitements sémantiques à réaliser :

1. Une règle récursive droite est de la forme : $A \rightarrow \alpha A \mid \epsilon$ avec $A \in Net \alpha \in (T \cup N)^+$. Le second membre droit de la production correspond à la dernière occurrence de α générée.

   Exemple 4.4 *La figure suivante illustre l'arbre syntaxique dans le cas de l'application des règles :*
   $< liste{-}idf > \rightarrow IDENTIFICATEUR, < liste{-}idf > | IDENTIFICATEUR$

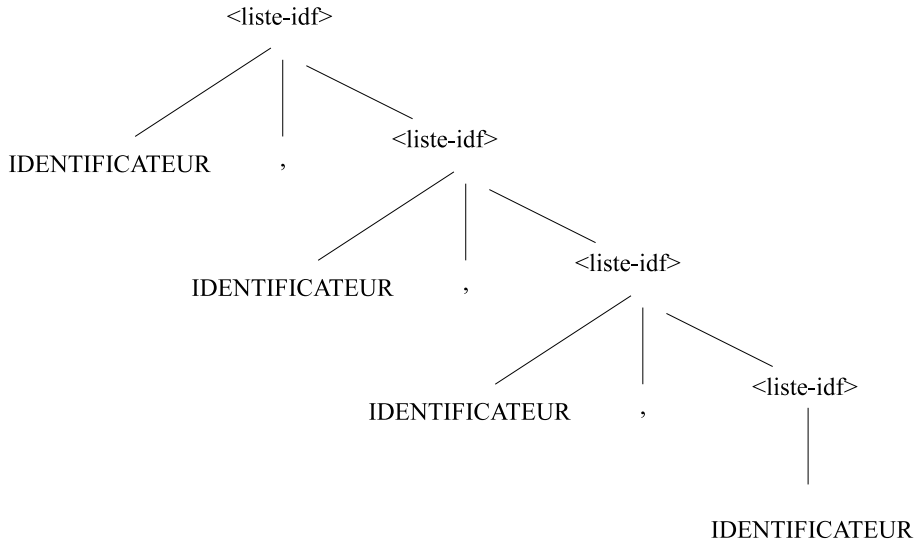


FIGURE 4.21 – Exemple d'application des règles récursives gauches

2. Une règle récursive gauche est de la forme : $A \rightarrow A\alpha \mid \epsilon$ avec $A \in N$ et $\alpha \in (T \cup N)^+$. Le second membre droit de la production correspond à la première occurrence de α générée.

   Exemple 4.5 *La figure suivante donne l'arbre syntaxique dans le cas de l'application des règles :*
   $< liste-idf > \rightarrow < liste-var >, IDENTIFICATEUR \mid IDENTIFICATEUR$

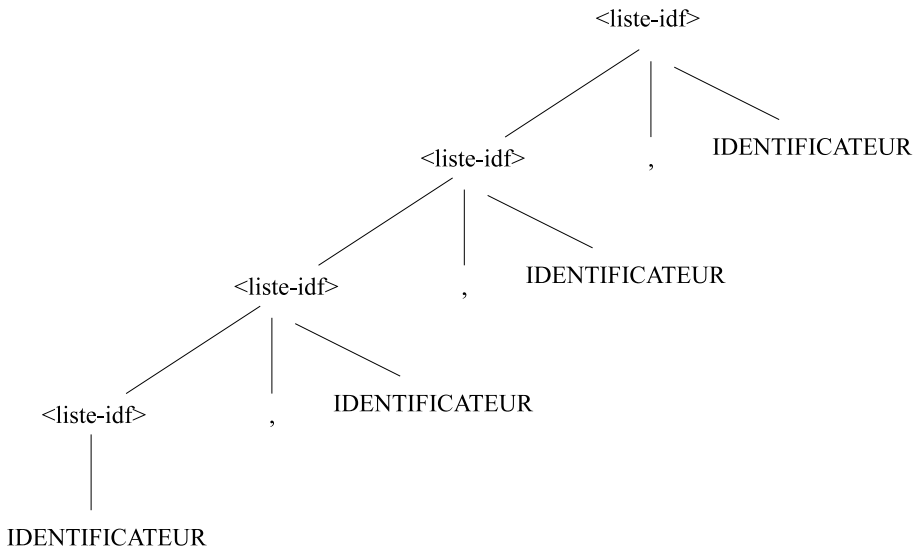


FIGURE 4.22 – Exemple d'application des règles récursives droites

Ainsi, si un traitement spécifique doit être effectué lors de la première occurrence de la séquence, il est préconisé d'utiliser une règle récursive gauche. Dans le cas contraire, il faut utiliser une règle récursive droite.

Remarque 21

7 conclusion

Les différentes méthodes d'analyse syntaxique abordées font intervenir des mécanismes différents. Néanmoins, il existe des relations entre les analyseurs syntaxiques ascendants et descendants comme l'illustre la Figure 7.

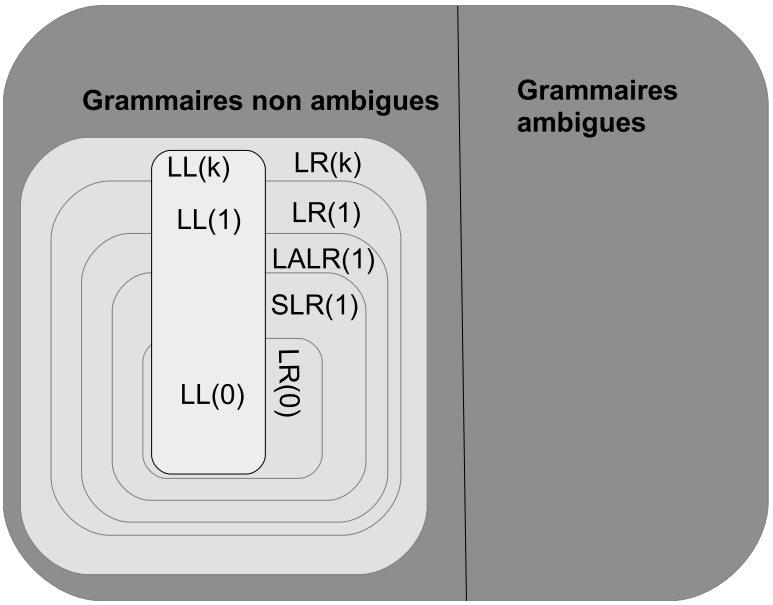


FIGURE 4.23 – Relations entre les analyseurs syntaxiques ascendants et descendants

Ainsi,

— LR(0)⊂ SLR(1) ⊂ LALR(1) ⊂ LR(1),

— LL(1) ⊂ LR(1),

En adéquation avec ces relations d'inclusion quelques exemples de grammaires sont cités :

— Grammaire LL(1) mais non SLR(1) :
Soit G la grammaire définie par <T,N,S,P> où T={a,b} ; N={S,A,B} ; et P :

$$\begin{cases} S \rightarrow AaAb \mid BbBa \\ A \rightarrow \epsilon \\ B \rightarrow \epsilon \end{cases}$$

1. G est LL(1) :
Considérons la table des débuts suivants associée par la grammaire :

	debuts	suivants
S	a b	#
A	ϵ	a b
B	ϵ	a b

Ainsi, G est LL(1) car débuts(AaBb) ∩ débuts(BbBa)=∅.

2. G n'est pas SLR(1) : Calcul des items LR(0) :
   I_0 = { [Z → .S] [S → .AaAb] [S → .BbBa] [A → .] [B → .]}
   I_1 = GOTO(I_0,S) ={[Z → S.]}
   I_2 = GOTO(I_0, A) ={[S → A.aAb]}
   I_3 = GOTO(I_0, B) ={[S → B.bBa]}
   I_4 = GOTO(I_2, a) ={[S → Aa.Ab] [A → .]}
   I_5 = GOTO(I_3, b) ={[S → Bb.Ba] [B → .]}
   I_6 = GOTO(I_5, A) ={[S → AaA.b]}
   I_7 = GOTO(I_6, B) ={[S → BbB.a]}
   I_8 = GOTO(I_6, b) ={[S → AaAb.]}
   I_9 = GOTO(I_7, a) ={[S → BbBa.]}
   C={I_0,I_1,I_2,I_3,I_4,I_5},I_6,I_7,I_8,I_9}
   G n'est pas SLR(1) car :
   - l'item [A → .] ∈ I_0 ⟹ M[I_0,a] ← $R_{A\to\epsilon}$ et M[I_0,b] ← $R_{A\to\epsilon}$ car a,b ∈ suivants(A),
   - l'item [B → .] ∈ I_0 ⟹ M[I_0,a] ← $R_{B\to\epsilon}$ et M[I_0,b] ← $R_{B\to\epsilon}$ car a,b ∈ suivants(B).

   ainsi, la table d'analyse correspondante va présenter deux cas de multi-définitions dûs aux cas deux cas de conflits $R_{A\to\epsilon}$ et $R_{B\to\epsilon}$.

— Grammaire SLR(1) mains non LL(1) :
Soit G la grammaire définie par <T,N,S,P> où T={id} ; N={S,X,Y} et P :

$$\begin{cases} S \to X \\ X \to Y \mid id \\ Y \to id \end{cases}$$

— Grammaire LR(1) et non LALR(1)
Soit G la grammaire définie par <T,N,S,P> où T={a,b,c,d} ; N={S,A,B} ; et P :

$$\begin{cases} S \to Aa \mid bAc \mid Bc \mid bBa \\ A \to d \\ B \to d \end{cases}$$

La phase de l'analyse syntaxique est importante lors de la conception d'un compilateur. En effet, la grammaire joue un rôle central dans les différentes phases d'analyse et de génération de codes.
En pratique, l'analyse syntaxique s'inscrit dans l'étape de traduction dirigée par la syntaxe qui englobe :

— l'analyse syntaxique,

— l'analyse sémantique,

— la génération du code intermédiaire.

Le fonctionnement de la traduction dirigée par la syntaxe s'appuie sur la grammaire syntaxique qui inclut :

— les règles syntaxiques,

— les routines sémantiques qui se chargent d'opérer des contrôles sémantiques et de générer le code intermédiaire.

Comme le processus de la traduction dirigée par la syntaxe (TDS) est complexe, il est utile d'aborder les différentes étapes séparément avant d'entamer le fonctionnement de la TDS dans le cas des analyseurs ascendants et descendants.

8 Exercices

Exercice 1 :

Soit la grammaire G définie par <T,ℕ,S,P> tels que T={a b (,} ; ℕ={S,T} et P :

$$\begin{cases} S \rightarrow a \mid b(T) \\ T \rightarrow T,S \mid S \end{cases}$$

a. Développez l'arbre de dérivation associé à la chaine b(a,b(a,a)).

b. Éliminez la récursivité gauche dans G.

c. Factorisez éventuellement la grammaire obtenue en a.

d. La grammaire obtenue est-elle LL(1) ?

e. Analysez les chaines : *b(a,a)#* et *b(a,(a,b))#* .

Exercice 2 :

Soit la grammaire G définie par <T,ℕ,L,P> tels que T={= ( ) + - num id int float print} ; ℕ={L,I,E,T} et P :

$$\begin{cases} L \rightarrow IL \mid \epsilon \\ I \rightarrow T\ id \mid id = E \mid print(E) \\ E \rightarrow E+E \mid E-E \mid id \mid num \\ T \rightarrow int \mid float \end{cases}$$

1. Donnez une arbre de dérivation pour la chaine *print(id+id+id)*.
2. Éliminez la récursivité gauche dans G.
3. Factorisez éventuellement la grammaire obtenue en 2.
4. Calculez les ensembles Début et Suivant de la grammaire obtenue en 3.
5. Construisez la table d'analyse LL(1) de la grammaire obtenue en 3.
6. La grammaire obtenue en 3 est-elle LL(1) ? Justifiez.

Exercice 3 :

Soit la grammaire G définie par <T,ℕ,L,P> tels que T={a b c d} ; ℕ={S, T, U} et P :

$$\begin{cases} S \rightarrow aTS \\ T \rightarrow bUT \mid \epsilon \\ U \rightarrow cU \mid dSc \mid \epsilon \end{cases}$$

1. Calculez les ensembles des débuts et des suivants de la grammaire G.
2. Construisez la table d'analyse LL(1) de la grammaire G.
3. La grammaire G est-elle LL(1) ? Justifiez.
4. Développez un analyseur syntaxique basé sur la descente récursive pour la grammaire G.

Exercice 4 :

Soit la grammaire G définie par <T,ℕ,S,P> tels que T={a b} ; ℕ={S,A,B} et P :

$$\begin{cases} S \rightarrow BA \\ A \rightarrow aB \mid \epsilon \\ A \rightarrow Bb \mid \epsilon \end{cases}$$

1. G est-elle LR(1) ?
2. G est-elle LALR(1) ?
3. Es-elle SLR(1) ?
4. Comparez les trois tables d'analyse.

Exercice 5 :

Soit la grammaire G définie par <T,ℕ,X,P> tels que T={a b c d} ;ℕ={X,M} et P :

$$\begin{cases} X \rightarrow Ma \mid bMc \mid dc \mid bda \\ M \rightarrow \mid d \end{cases}$$

1. G est-elle SLR(1) ? Justifiez.
2. G est-elle LR(1) ? Justifiez.
3. G est-elle LALR(1) ? Justifiez.
4. Comparez les trois tables d'analyse.

Exercice 6 :

Soit la grammaire G définie par <T,ℕ,E,P> tels que T={ ? @ op nb} ;ℕ={E} et P :

$$\begin{cases} E \rightarrow ?E \mid E@E \mid EopE \mid nb \end{cases}$$

a. G est-elle ambiguë ?
b. Construisez la table d'analyse SLR(1) .
c. Est-il possible d'éliminer les multi-définitions en adoptant les conventions suivantes ?
   — ? plus prioritaire que @
   — @ plus prioritaire que op
   — l'associativité est de gauche à droite
d. Analysez la chaine *nb @ nb op nb#*.

Exercice 7 :

Soit la grammaire G définie par <T,ℕ,E,P> tels que T={id sep par entier ser ( )} ; ℕ={S D C} et P :

$$\begin{cases} S \rightarrow D\ sep\ C \\ D \rightarrow id\ entier \mid D\ id\ entier C \rightarrow id \mid C\ par\ C \mid C\ ser\ C \mid (C) \end{cases}$$

a. G est-elle ambiguë ?

b. Construisez la table d'analyse SLR(1) .

c. Est-il possible d'éliminer les multi-définitions en adoptant certaines conventions ?

Chapitre 5

Les formes intermédiaires

1 Introduction

Le but d'un compilateur est de traduire un programme source en un langage exécutable par une machine. Néanmoins, étant donné que les langages sont souvent complexes, il est nécessaire de générer en premier lieu un code intermédiaire avant de générer le code objet.
Le processus de génération du code intermédiaire, comme l'illustre la Figure 5.1, est intégré dans la phase de traduction dirigée par la syntaxe dont les actions sont guidées par une grammaire sémantique.
Une grammaire sémantique est une grammaire syntaxique à laquelle sont intégrées des routines sémantiques qui se chargent d'opérer des contrôles sémantiques et de générer le code intermédiaire.
Concrètement, la génération de code intermédiaire traduit l'arbre syntaxique en un code destiné pour une machine abstraite disposant d'une mémoire sans limite. Il est à préciser que la structure du code intermédiaire est simple.

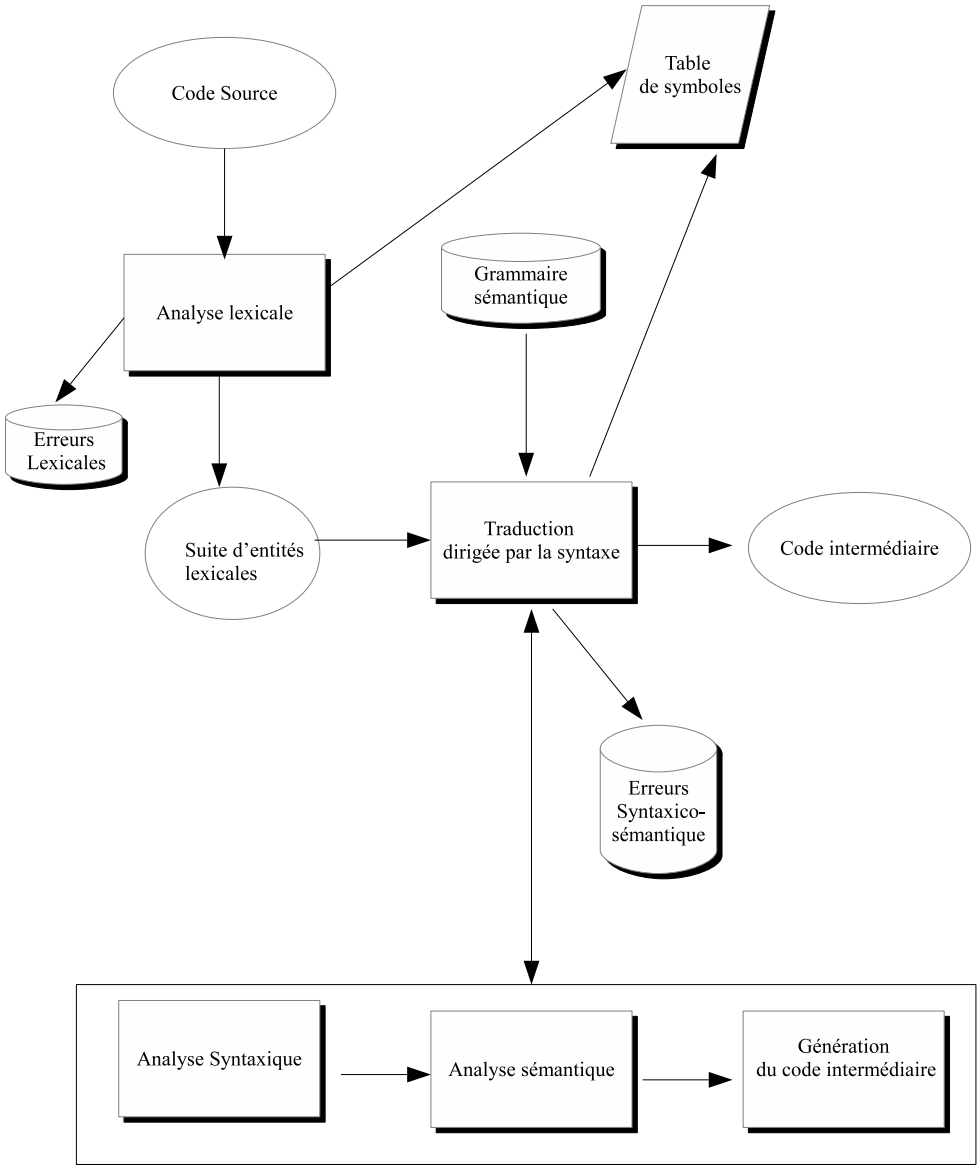


FIGURE 5.1 – Les étapes de la traduction dirigée par la syntaxe

Définition 5.1 *Une forme intermédiaire est un code dont l'ensemble des instructions sont élémentaires.*
Une instruction élémentaire est composée :

— *d'un opérateur (arithmétique, logique,...),*

— *de deux opérandes (constantes, nom de procédure,...).*

Il existe plusieurs formes intermédiaires :

— la forme post-fixée,

— la forme préfixée,

— les arbres abstraits qui sont des des représentations structurées offrant une vision globale du fichier source,

— les codes à 3 adresses qui sont des représentations linéaires simples à manipuler et qui incluent les quadruplets, les triplets directs et les triplets indirects.

2 La forme post-fixée ou la notation polonaise

Une forme post-fixée (FP) se présente comme suit :

Opérande1 *Opérande2 Opérateur*

L'opérateur se positionne après les opérandes.
Les formes post-fixées élémentaires sont :

FP(T_1 Opérateur T_2)	=	FP(T_1) FP(T_2) Opérateur	cas d'un opérateur binaire
FP(opérateur T)	=	FP(T) Opérateur	cas d'un opérateur unaire
FP(constante)	=	constante	
FP(variable_simple)	=	variable_simple	

2.1 Évaluation d'une expression en notation post-fixée :

L'évaluation des expressions arithmétiques et booléennes requirent l'utilisation d'une pile pour sauvegarder les opérandes. Le processus d'évaluation se déroule comme suit :

— si le terme courant est un opérande alors il est empilé et le pointeur vers l'entité courante est avancé,

— si le terme courant est un opérateur binaire alors :

1. l'opération de dépilement est effectuée deux fois afin de récupérer l'opérande 2 puis l'opérande 1,
2. l'opération est effectuée et le résultat est récupéré dans un temporaire,
3. le pointeur vers l'entité courante est avancé.

— si le terme courant est un opérateur unaire alors :

1. l'opération de dépilement est effectuée une fois afin de récupérer l'opérande,
2. l'opération est effectuée et le résultat est récupéré dans un temporaire,
3. le pointeur vers l'entité courante est avancé.

L'algorithme 27 décrit les étapes d'évaluation des expressions faisant intervenir des opérateurs binaires et des opérateurs unaires.

```
Begin
if (tc est une opérande) then
   emplier(tc);
   tc ← tc+1;
else
   if (tc est un opérateur binaire) then
      dépiler(opérande2);
      dépiler(opérande1);
      évaluer(tc, opérande1, opérande2, temporaire);
      empiler(temporaire);
      tc ← tc+1;
   else
      if (tc est un opérateur unaire) then
         depiler(opérande);
         évaluer(tc, opérande, temporaire);
         empiler(temporaire);
         tc ← tc+1;
      end if
   end if
end if
End
```

Algorithme 23 : Évaluation des expressions

Exemple 31 *Soit l'expression* ***a-c+d*e****. La forme post-fixée de l'expression est :* ***ac-de*+****. L'évaluation de l'expression est illustrée par la table suivante :*

Pile	Chaine	Action
Vide	ac-de*+	empiler(a) ; avancer() ;
a	c-de*+	empiler(c) ;avancer() ;
a c	-de*+	dépiler(c), dépiler(a), évaluer(-,a,c,t1) ; empiler(t1) ; avancer() ;
t1	de*+	empiler(d) ; avancer() ;
t1 d	e*+	empiler(e) ; avancer() ;
t1 d e	*+	dépiler(e) ; dépiler(d), évaluer(*,d,e,t2) ; empiler() ;avancer() ;
t1 t2	+	dépiler(t2) ; dépiler(t1) ; évaluer(+,t1,t2,t3) ; empiler(t3) ;avancer() ;
t3	Vide	

Remarque 22 *Pour évaluer les expressions arithmétiques et booléennes, les critère d'associativité gauche et les priorités entre les opérateurs sont utilisés afin de spécifier l'ordre d'évaluation des opérations.*

- *l'expression associée à l'opérateur le plus prioritaire est évaluée en premier lieu.*
- *si l'opérateur est associatif à gauche, l'expression la plus à gauche est évaluée en premier*
- *Dans le cas où l'opérateur est associatif à droite, c'est l'expression la plus à droite qui est évaluée en premier.*

Remarque 23 *Durant la phase de génération du code intermédiaire, le nombre des temporaires est supposé illimité. Chaque temporaire correspond à un registre virtuel. Les temporaires sont alors crées à la demande. D'ailleurs, la génération du code intermédiaire crée beaucoup de temporaires de courte durée. La correspondance entre registre virtuel et registre réel se fera ultérieurement.*

Remarque 24 *En général, les opérandes représentent des identificateurs, des variables du programme, des variables temporaires, des étiquettes de branchement et des constantes.*

2.2 La forme postfixée d'une affectation

La forme postfixée de l'instruction d'affectation est définie par :
Variable := <Expression> est :

FP(Variable) FP(<Expression>) :=

Exemple 32 *Soit l'instruction d'affectation delta :=b*b-4*a*c. Sa forme postfixée se présente comme suit : delta b b *4 a* c* - :=*
L'ordre d'évaluation des différents opérateurs de cette expression obéit :

- *au critère d'associativité gauche de la multiplication,*
- *à la priorité supérieur de (*) par rapport à la soustraction (-).*

2.3 La forme postfixée d'un branchement inconditionnel

La forme postfixée de l'instruction du branchement inconditionnel définie par :
GOTO étiquette
est :

BRL, où l'étiquette est définie dans le programme source.
opérande **BR**, où opérande est une position dans la chaine post-fixée.

2.4 La forme postfixée d'un branchement conditionnel

<$Opérande_1$><$Opérande_2$> Opérateur de comparaison par rapport à 0

où :

<$Opérande_1$> représente la forme postfixée de la condition,
<$opérande_2$> représente la position dans la chaine postfixée.

Les opérateurs de comparaison par rapport à 0 sont :	BZ(=0) , BNZ($\neq$0)
	BP(>0) , BPZ($\geq$0)
	BM(<0) , BMZ($\leq$0)

Remarque 25 *Si la condition se rapporte à une valeur différente de 0, il va falloir passer par la soustraction car les opérateurs disponibles pour la forme postfixée se rapportent à 0.*

Exemple 5.1 *Une expression du type "IF (x+2>3)" sera transformée en une expression équivalente "IF (x+2-3>0)".*

2.5 La forme post-fixée d'une déclaration de tableau

L'instruction de déclaration d'un tableau est définie par :
ARRAY nom-tableau [L_1 :U_1, L_2 :U_2, ..., L_n :U_n] tels que L_i et U_i définissent respectivement la borne inférieure et la borne supérieure de la dimension i.
Les indices sont décrits par des expressions arithmétiques de type entier.
La forme postfixée de l'instruction est comme suit :
FP(L_1)FP(U_1)FP(L_2)FP(U_2) ... FP(L_n)FP(U_n) nom-tableau ADEC
ADEC (Array Declaration) est un opérateur de déclaration d'un tableau.

Exemple 33 *La forme postfixée de l'instruction :*
*tab[1 :a*b,a :c + d * b] est : 1 a b* a c d b*+ tab ADEC*

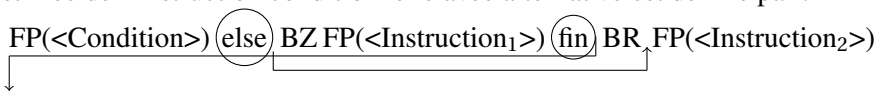
2.6 La forme post-fixée d'une référence à un élément de tableau

La forme post-fixée qui correspond à la référence à un élément du tableau du type :
nom-tableau[<expression$_1$>,<expression$_2$>,...,<expression$_n$>] est :
FP(<Expression$_1$>)FP(<Expression$_2$>)...FP(<Expression$_n$>) nom-tableau SUBS
SUBS est un opérateur de référence à un élément d'un tableau.

Exemple 34 *Soit l'instruction d'affectation définie par : x := M[i*j-l, b, l+m] tel que M est un tableau à trois dimensions.*
*La forme post-fixée de l'instruction est : x i j *l - b l m + M SUBS :=*

2.7 La forme post-fixée de l'instruction conditionnelle avec alternative

L'instruction avec alternative se présente comme suit :
IF (<Condition>) THEN <$Instruction_1$> ELSE <$Instruction_2$>.
La forme postfixée de l'instruction conditionnelle avec alternative est définie par :

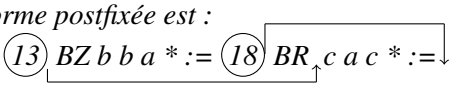


où :

- else : représente la position de la première entité de la chaine postfixée de (<inst2>),
- fin : représente la position dans la chaine postfixée qui suit immédiatement la dernière entité syntaxique de la forme postfixée de (<inst2>).

Remarque 26 *Par convention, l'expression booléenne est égale à '0' si elle est fausse et elle est égale à '1' sinon.*

Exemple 35
*Soit l'instruction suivante : IF (a<>b) THEN b :=b*a ELSE c :=a*c*
Sa forme postfixée est :

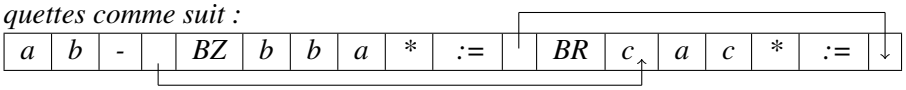


Remarque 27

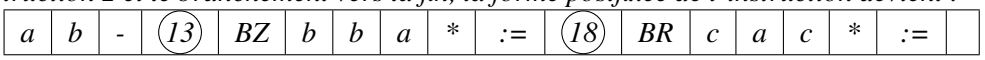
La forme postfixée peut être représentée par un vecteur ou un tableau à une dimension, noté FP, dans lequel chaque élément du vecteur FP représente soit un opérateur soit un opérande (variable, étiquette, constante,...).

- *Si lors de la génération des codes intermédiaires des branchements conditionnels ou inconditionnels, la position de l'étiquette de branchement n'est pas connue, alors il est nécessaire de sauvegarder cette position afin de pouvoir la mettre à jour à l'aide des routines sémantiques.*

Ainsi, l'instruction précédente peut être représentée avant la mise à jour des étiquettes comme suit :



Après la résolution des étiquettes concernant le branchement vers début de l'instruction 2 et le branchement vers la fin, la forme postfixée de l'instruction devient :



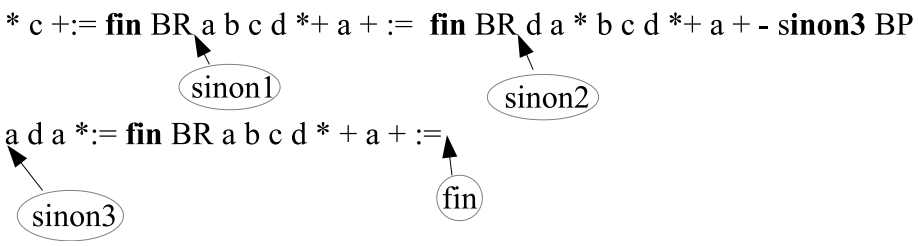
Remarque 28
Les étiquettes 13 et 18 représentent les indices du vecteur FP dans le cas où le premier indice est égale à 1. dans le cas générale, si l'indice du début de l'instruction est "i" alors les indices du else et de la fin de l'instruction seront respectivement "i+13" et "i+18".

Exemple 36 *Soit l'instruction a :=MIN(a*b+c,d*a,b+c*d+a) tel que MIN est un opérateur qui désigne le minimum.*
Afin de représenter cette instruction en code postfixée, il est nécessaire de spécifier son fonctionnement sous forme d'un code élémentaire comme suit :

```
début
    si (a*b+c<=d*a) alors
        si (a*b+c<=b+c*d+a) alors
            a := a*b+c ;
        sinon
            a := b+c*d+a ;
    sinon
        si (d*a<=b+c*d+a) alors
            a := d*a ;
        sinon
            a := b+c*d+a ;
fin
```

Ainsi, le code postfixé associé à l'instruction d'affectation est comme suit :

a b * c + d a * **sinon2** BP a b * c + b c d * + a + **-** **sinon1** BP a a b



3 la forme préfixée

La forme préfixée est une forme intermédiaire telle que l'opérateur précède les opérandes. Elle se présente comme suit :

- pour un opérateur binaire : Opérateur Opérande$_1$ Opérande$_2$,
- pour un opérateur unaire : Opérateur Opérande.

4 Les quadruplets

Un quadruplet a la forme générale suivante :

(Opérateur, Opérande$_1$, Opérande$_2$, Temporaire)

Cette forme intermédiaire est intéressante car elle permet de préciser les mémoires de manœuvre ou temporaires) nécessaires pour sauvegarder les résultats intermédiaires lors de l'évaluation des expressions arithmétiques, booléennes. La représentation sous forme de quadruplets peut être étendue aux différentes instructions d'un langage.

4.1 L'instruction d'affectation

<var> := <expression>
(:=, <temporaire.expression>, , <var>)

Exemple 37 *Soit l'instruction d'affectation suivante : a :=a*b/c+d. La forme intermédiaire correspondante est :*

1. *(*, a, b, T1)*
2. *(/, T1, c, T2)*

3. *(+, T2, d, T3)*
4. *(:=,T3, ,a)*

T1, T2, T3 désignent des mémoires de manœuvre. Une certaine optimisation des mémoires de manœuvre est nécessaire avec cette méthode, sinon il y a un risque d'aboutir à une occupation mémoire importante.

Remarque 29 *Dans un quadruplet, les trois opérandes ne sont pas toujours nécessaires.*

4.2 Le branchement inconditionnel (To label)

GOTO étiquette ⇒ (BRL, étiquette, ,)
(BR, N° quadruplet, ,)

Remarque 30 *Dans un quadruplet, les trois opérandes ne sont pas toujours nécessaires.*

4.3 Le branchement conditionnel

La forme de quadruplets offre :

— des opérateurs de comparaison par rapport à zéro :

(BZ[BNZ,BM,BMZ,BP,BPZ], N° quadruplet, T,)

— des opérateurs binaires de comparaison : BE(=), BNE (≠), BL (<), BLE(<=), BP(>), BPZ(>=0) :

(BE[BNE,BL,BLE,BG,BGE], N° quadruplet, T1, T2)

4.4 La déclaration de tableau

Array A[L_1 :U_1, ..., L_n :U_n]

(quadruplets de $L_1 \rightarrow T_{11}$)
(quadruplets de $U_1 \rightarrow T_{12}$)
(BOUNDS, T_{11}, T_{12},)
.
.
.
(quadruplets de $L_n \rightarrow T_{n1}$)

(quadruplets de $U_n \rightarrow T_{n2}$)
(BOUNDS, T_{n1}, T_{n2},)
(ADEC, A, ,)

Exemple 38 *Soit l'instruction de déclaration d'une matrice suivante :*
Array A[1 :1+j-k, 2 :c-d]

1. *(+, 1, j, T1)*
2. *(-, T1, k, T2)*
3. *(BOUNDS, 1, T2,)*
4. *(-, c, d, T3)*
5. *(BOUNDS, 2, T3,)*
6. *(ADEC, A, ,)*

4.5 La référence à un élément de tableau

A[<exp_1>, <exp_2>, <exp_3>, ...,<exp_n>]

(quadruplets de <exp_1> $\rightarrow$ T_1)
(quadruplets de <exp_2> $\rightarrow$ T_2)
.
.
.
(quadruplets de <exp_n> $\rightarrow$ T_n)
A[T_1, T_2, ..., T_n]

Exemple 39 *Soit l'instruction suivante x := x+tab[m+4,j-l*i]. Le code intermédiaire sous forme de quadruplets associé à cette instruction est la suivante : (+,m,4,T1)*
(,l,i,T2)*
(-,j,T2,T3)
(+,x,tab[T1,T3],T4)
(:=,T4, ,x)

4.6 Instruction conditionnelle

La forme quadruplet de l'instruction conditionnelle définie par :
If (<Condition>) THEN <$Instruction_1$> ELSE <$Instruction_2$> est :
Quadruplets(condition) dont l'évaluation finale sera sauvegardée dans un temporaire (T.cond) ;
(BZ, else,T.cond, ,)

Quadruplets(<$Instruction_1$>
.
.
(BR, fin, ,)
Quadruplets (<$Instruction-2$>
.
.

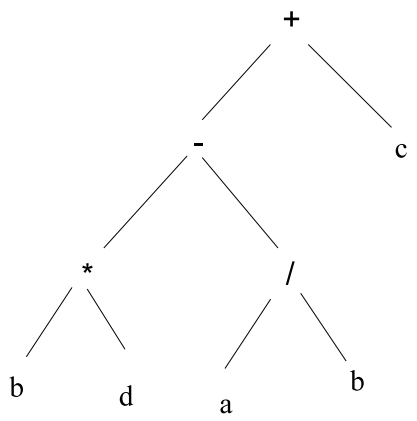
Exemple 5.2 *soit l'instruction : IF(b > c+1) THEN b :=b*a/c ELSE c :=a*c ;*
Le code intermédiaire, sous forme de quadruplets, associé à l'instruction est :
1- (+,c,1,T1)
2- (BLE, ,b,T1)
3- (,b,a,T2)*
4- (/,t2,c,T3)
5- (:=,T3, ,b)
6- (BR, , ,)
7- (,a,c,T4)*
8- (:=,T4, ,a)
9-
Mise à jour des adresses :
1- (+,c,1,T1)
2- (BLE,7,b,T1)
3- (,b,a,T2)*
4- (/,t2,c,T3)
5- (:=,T3, ,b)
6- (BR,9 , ,)
7- (,a,c,T4)*
8- (:=,T4, ,a)
9-

L'utilisation des quadruplets est couteuse en terme de consommation de l'espace mémoire. Les triplets ont été définis pour palier à cet inconvénient.

5 Les arbres abstraits

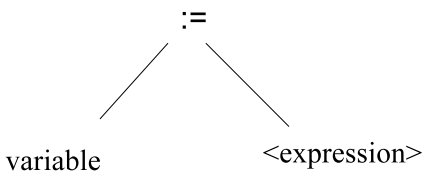
Les arbres abstraits représentent les formes intermédiaires les plus utilisées en compilation. C'est une représentation arborescente qui présente l'avantage de représenter l'arbre syntaxique déduit de l'analyse syntaxique de manière abstraite : les non-terminaux de la grammaire sont supprimés et seuls les opérandes et les opérateurs sont considérés.

Exemple 40 *Soit l'expression arithmétique suivante : b*d-a/b+c. L'arbre abstrait correspondant à cette expression est représenté par :*



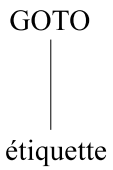
5.1 Affectation

L'arbre abstrait relatif à l'instruction d'affectation "variable :=<expression>" est représenté comme suit :



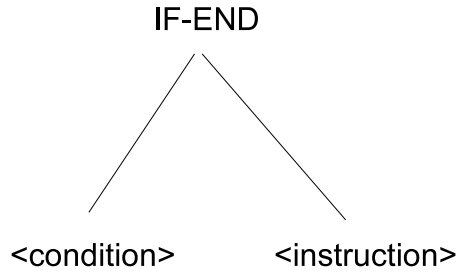
5.2 Branchement inconditionnel

L'arbre abstrait relatif à l'instruction du branchement inconditionnel "GOTO étiquette" est illustré comme suit :



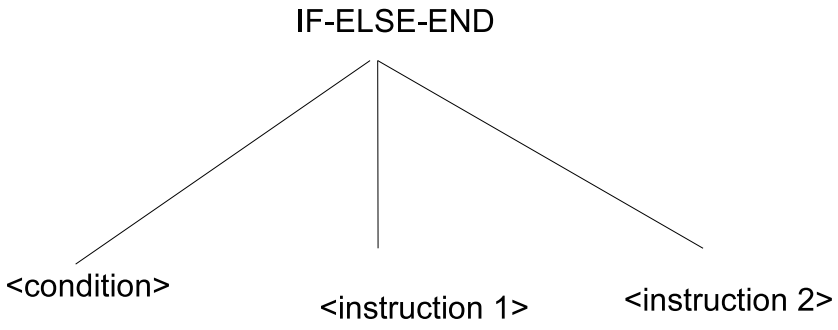
5.3 Instruction conditionnelle sans alternative

L'arbre abstrait relatif à l'instruction conditionnelle sans alternative " IF <condition> THEN <instruction> END est donné par la Figure 5.3.



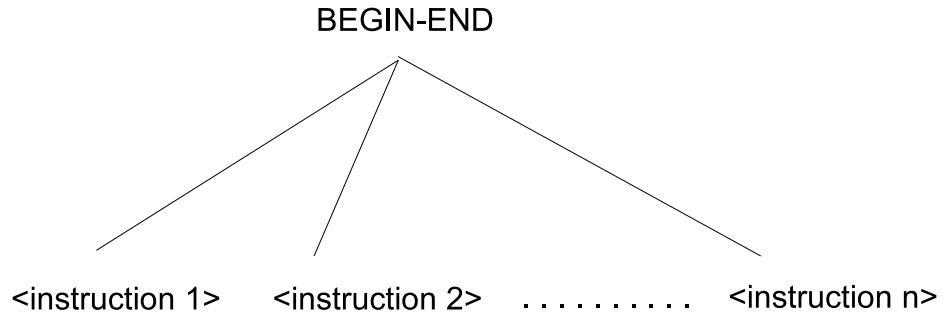
5.4 Instruction conditionnelle avec alternative

L'arbre abstrait relatif à l'instruction conditionnelle avec alternative " IF <condition> THEN <$instruction_1$> ELSE <$instruction_2$> END" est donné est illustré comme suit :



5.5 Bloc d'instructions

L'arbre abstrait relatif à l'instruction qui définit un bloc de n instructions "BEGIN <$instruction_1$>, <$instruction_2$>;...;<$instruction_n$>" est est illustré comme suit :

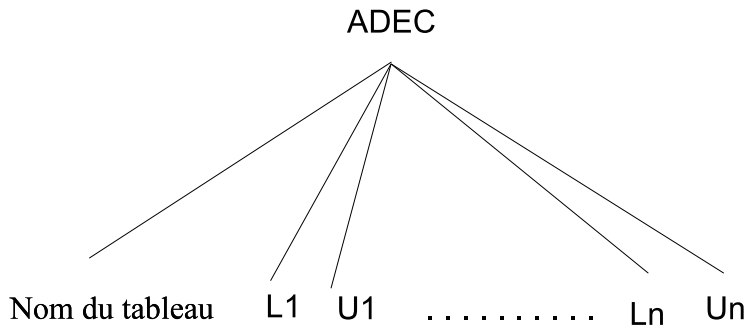


5.6 Déclaration d'un tableau

L'arbre abstrait relatif à l'instruction de déclaration d'un tableau à n dimensions définie par :

ARRAY nom-tableau [L_1 :U_1,..., L_n :U_n]

tels que L_i :U_i représentent respectivement la borne inférieure et la borne supérieure de la dimension i, est illustré comme suit :

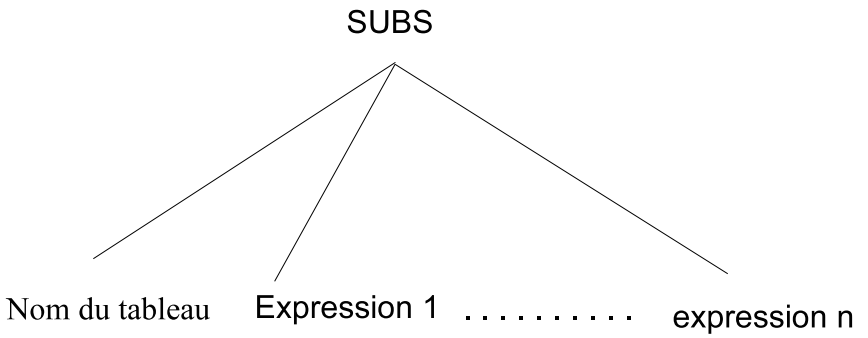


5.7 Référence à un élément de tableau

La référence à un élément d'un tableau à n dimensions du type

nom-tableau[<exp_1>, ..., <exp_n>]

est représentée par un arbre abstrait comme indiqué par la Figure suivante :

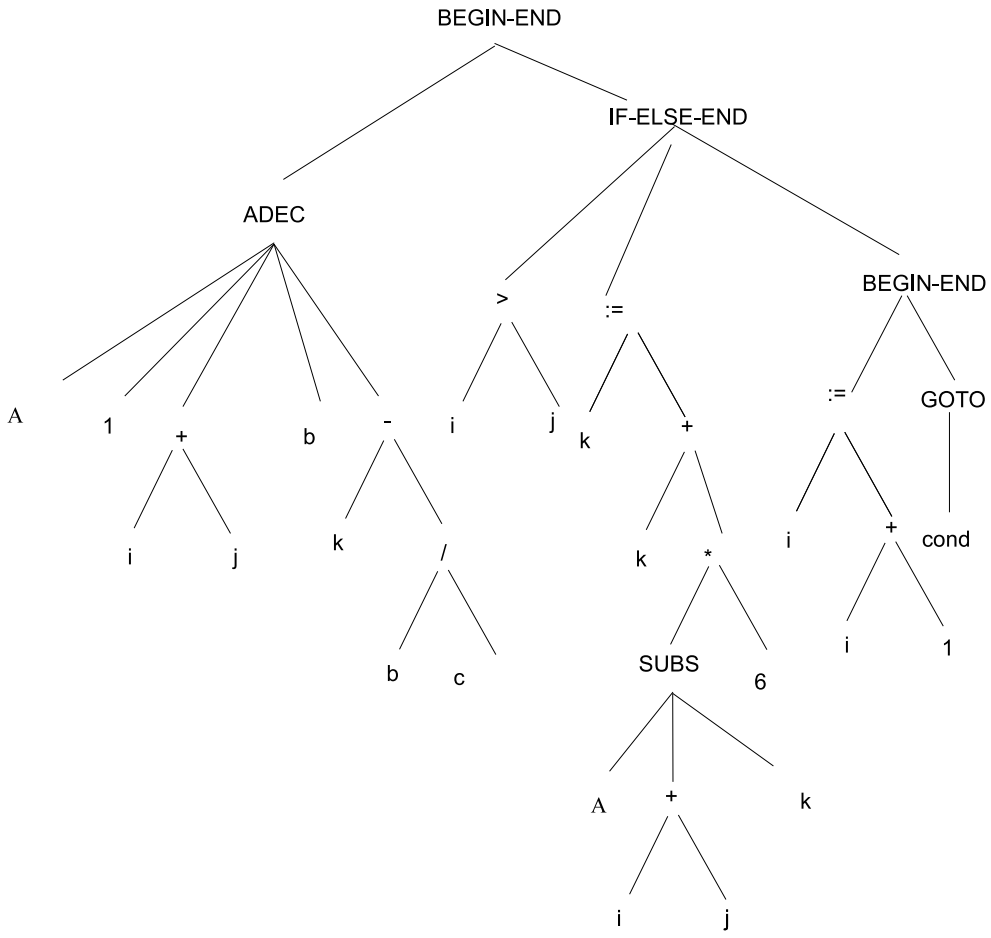


Chaque indice est exprimé sous forme d'une expression arithmétique de type ENTIER.

Exemple 41
Soient les instructions suivantes :

```
début
    Array A[1 :i+j, b :k-b/c]
    cond : si (i>j) alors
        k := k + A[i+j, k]*6 ;
    sinon
        début
            i := i+1 ;
            GOTO cond ;
        fin
fin
```

L'arbre abstrait qui correspond à cet algorithme est donné par la Figure suivante :



6 Exercices

Exercice 1 :

Traduire les expressions suivantes en code postfixé, en quadruplets et en arbres abstraits :

1. Begin Integer Array V[a :a+b] ;
   For I := a to a+b Step 1
   Do V[I] := V[I] + V[a+b] ;
   End ;
2. Begin If(a<b*c)
   Then While a<=b*c

```
        Do Begin a := a+1 ;
                 a :=b*c ;
        End ;
    Else If a<0 Then a :=0 ;
                Else a :=b*c ;
End ;
```

Exercice 2 :

Traduire les expressions suivantes en code postfixé, en quadruplets et en arbres abstraits :

1. A := A+B*((D+a)/(D+B))*C
2. P := (P*(S/2))+(X-X/Y)

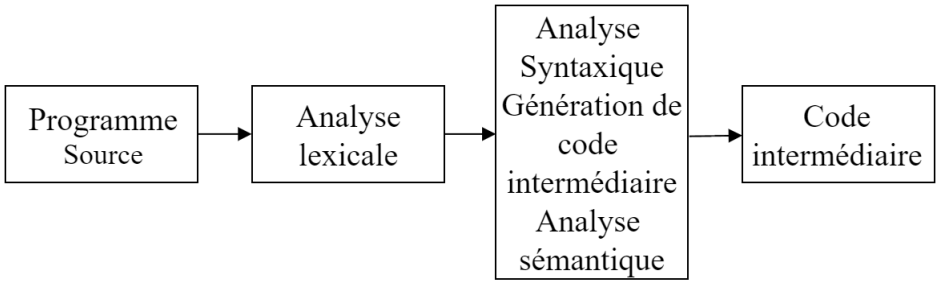
Exercice 3 :

Traduire les expressions suivantes en quadruplets en utilisant les opérateurs BZ, BNZ et BR :

```
v :=x AND y OR z AND y OR NOT x
```

Chapitre 6

Traduction dirigée par la syntaxe



L'analyse sémantique, en liaison étroite avec l'analyse syntaxique syntaxique, permet de donner un sens à ce qui à été reconnu dans cette phase. Les actions sémantiques sont multiples telles que :

- vérifier si un identificateur a été bien déclaré,
- vérifier la concordance des types dans une expression,
- vérifier l'utilisation correcte d'une étiquette(déclaration, utilisation, référence),
- ...

La traduction dirigée par la syntaxe consiste à associer à l'analyseur syntaxique, la phase d'analyse sémantique ainsi que la phase de génération du code intermédiaire. Afin de mettre en œuvre cette étape composée, la grammaire syntaxique sera augmentée par des routines sémantiques qui vont se charger :

- d'opérer des contrôles sémantiques,
- de générer le code intermédiaire.

Une routine sémantique sera donc exécutée à chaque fois qu'un contrôle sémantique est nécessaire :

- après une réduction dans le cas d'une analyse ascendante,
- lors de la dérivation dans le cas d'une analyse descendante.

Un schéma de traduction est constitué de l'ensemble :

- un code intermédiaire,
- une grammaire associée (transformée),
- des routines sémantiques.

1 Traduction dirigée par la syntaxe dans le cas de l'analyse descendante :

Les routines sémantiques seront exécutées lors des dérivations. La grammaire doit être transformée comme suit :
Si A $\rightarrow \alpha\beta$ est une règle de la grammaire G tels que $\alpha, \beta \in (T \cup N)^*$ et si une routine doit être placée entre α et β, la règle sera modifiée de la manière suivante :

$$\begin{cases} \text{A} \rightarrow \alpha \text{B} \beta \\ \text{B} \rightarrow \epsilon \end{cases}$$

Ainsi, il est nécessaire de rajouter des non-terminaux qui vont se dériver en ϵ afin de permettre l'exécution des routines sémantiques qui vont opérer des traitements qui consistent à faire des contrôles sémantiques et à générer le code intermédiaire sans pour autant modifier la syntaxe.

1.1 Instruction conditionnelle

La syntaxe de l'instruction conditionnelle est définie par la règle syntaxique suivante :

<inst-if> → IF <Condition> THEN <Instruction1> ELSE <Instruction2>

Le schéma de traduction de l'instruction en utilisant les quadruplets est défini par :

- Le code intermédiaire généré sous forme de quadruplets :
  Afin de représenter les quadruplets, une matrice QUAD est utilisée tel que le premier quadruplet libre est pointé par l'indice qc.

  {quadruplets de <Condition>
  (BZ, else , <Condition>.temp,)
  {quadruplets de <Instruction1>
  (BR, Fin, ,)

{quadruplets de <Instruction2>

- La grammaire associée :
  Elle est obtenue en insérant dans les règles syntaxiques de l'instruction des routines sémantiques qui vont se charger de réaliser des contrôles sémantiques et de générer le code intermédiaire.

  <inst-if> → IF <Condition> **<A>** THEN <Instruction1> **<B>** ELSE <Instruction2> **<C>**

  <A> → ϵ (Cette routine permet de générer un branchement vers le début des instructions qui correspondent au cas où la condition est fausse. Le numéro du quadruplet de branchement est sauvegardé afin de mettre à jour l'étiquette.)

  <B> → ϵ (Cette routine permet de générer l'instruction du branchement inconditionnel BR vers la fin de l'instruction conditionnelle. L'étiquette de l'instruction BZ est aussi mise à jour.)

  <C> → ϵ (Cette routine permet la mise à jour de l'étiquette de l'instruction BR.)

  A chaque routine sera associé un non terminal qui se dérivera en epsilon. Ainsi, lors de la dérivation, ce non terminal va permettre d'exécuter la routine sémantique correspondante. Afin de ne pas modifier la syntaxe de l'instruction, les noms terminaux rajoutés se dérivent en epsilon.

- Les routines sémantiques :
  Routine <A>

```
début
    QUAD(qc) := (BZ, , <Condition>.temp, ) ;
    sauv_bz := qc ;
    qc :=qc+1 ;
fin
```

  Routine <B>

```
début
    QUAD(qc) := (BR, , , ) ;
    sauv_br := qc ;
    qc := qc+1 ;
    quad(sauv_bz, 2) := qc ;
fin
```

  Routine <C>

début
| quad(sauv_br, 2) := qc ;
fin

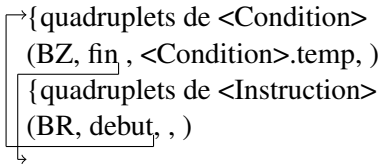
1.2 Instruction while

La syntaxe de l'instruction WHILE est définie comme suit :

<inst-while> → WHILE <Ccondition> DO <Instruction>

Le schéma de traduction de l'instruction en utilisant les quadruplets est défini par :

- Le code intermédiaire généré sous forme de quadruplets :



- La grammaire transformée :
  <inst-while> → WHILE **<A>** <Condition> **<B>** DO <Instruction> **<C>**
  <A> → ϵ (elle permet la sauvegarde du début de la boucle pour y revenir)
  <B> → ϵ (elle permet de tester la condition)
  <C> → ϵ (elle permet de générer l'instruction BR vers le début de la boucle et de mettre à jour l'instruction BZ)

- Les routines sémantiques :
  Routine <A>

début
| sauv_debut := qc ;
fin

Routine <B>

début
| QUAD(qc) := (BZ, , <condition>.temp,) ;
| sauv_bz := qc ;
| qc :=qc+1 ;
fin

Routine <C>

début
| QUAD(qc) := (BR, sauv_debut, ,) ;
| qc :=qc+1 ;
| quad(sauv_bz, 2) := qc ;
fin

1.3 Instruction Repeat

La syntaxe de l'instruction repeat se présente comme suit :

<inst-repeat> → REPEAT <Insruction> UNTIL <Condition>

Le schéma de traduction de l'instruction sous forme de quadruplets dans le cas d'une analyse descendante est défini par :

1. le code intermédiaire généré sous forme de quadruplets :
   Début →
   { Quadruplets de <Instruction>
   { Quadruplets de <Condition>
   (BZ, Début ,<Condition>.temp ,)
2. la grammaire transformée :
   <inst-repeat> → REPEAT **<A>** <Instruction> UNTIL <Condition> **<B>**
   <A> → ϵ (Elle permet de sauvegarder le début de l'instruction)
   <B> → ϵ (Elle permet le branchement vers le début de l'instruction)
3. les routines sémantiques :
   Routine <A>

début
| deb-inst :=qc ;
fin

Routine <B>

début
  QUAD(qc) :=(BZ,deb-inst,<Condition>.temp,) ;
  qc :=qc+1 ;
fin

1.4 Instruction For

<inst-for> → FOR id :=<Expression1> TO <Expression2> STEP <Expression3> DO <Instruction>

Le schéma de traduction de l'instruction sous forme de quadruplets est défini par :

1. le code intermédiaire généré sous forme de quadruplets :

.
. Quadruplets de <Expression1>
.
(:=,<Expression1>.temp, ,id)
.
. Quadruplets de <Expression2>
.
(BG, finfor , id,<Expression2>.temp)
.
. Quadruplets de <Expression3>
.
.
. Quadruplets de <Instruction>
.
(+,id,<Expression3>.temp,id))
(BR, debut, ,)

2. la grammaire transformée :
   <inst-for> → FOR id **A** :=<Expression1> **B** TO <Expression2> **C** STEP <Expression3> **D** DO <Instruction> **E**
   <A> → ϵ (elle permet de rechercher l'identificateur dans la TS afin de vérifier si elle est déclarée et de vérifier son type)
   <B> → ϵ (elle permet de vérifier le type de l'expression 1 et d'initialiser le compteur.)
   <C> → ϵ (elle permet de vérifier le type de l'expression 2 et de tester si la borne supérieure du compteur a été atteinte.)

<D> $\rightarrow \epsilon$ (elle permet de vérifier le type de l'expression 3 et de récupérer sa valeur.)
<E> $\rightarrow \epsilon$ (elle permet de générer l'instruction de l'incrémentation du compteur et de générer le branchement vers le début de la boucle.)

3. les routines sémantiques :
   Routine <A>

```
début
    lookup(id.nom,P) ;
    si (P=0) alors
        écrire ("erreur : l'identificateur n'a pas été déclaré")
    sinon
        si (P.type ≠ Entier) alors
            écrire ("erreur : l'identificateur n'est pas de type entier")
        sinon
            variable := id.nom
fin
```

   Routine <B>

```
début
    si (<Expression1>.type ≠ Entier) alors
        écrire ("erreur : l'expression n'est pas de type entier")
    sinon
        QUAD(qc) :=( :=,<Expression1>.temp, , variable) ;
        qc :=qc+1 ;
fin
```

   Routine <C>

```
début
    si (<Expression2>.type ≠ Entier) alors
        écrire ("erreur : l'expression n'est pas de type entier")
    sinon
        QUAD(qc) :=(BG, , variable, <Expression2>.temp) ;
        sauve_BG := qc ;
        qc :=qc+1 ;
fin
```

   Routine <D>

```
début
    si (<Expression3>.type ≠ Entier) alors
        écrire ("erreur : l'expression n'est pas de type entier")
    sinon
        QUAD(qc) :=(+,variable,<Expression3>.temp,variable);
        qc :=qc+1;
fin
```

Routine <E>

```
début
    QUAD(qc) := (BR, sauv_BG, , );
    qc :=qc+1;
    quad(sauv_BG, 2) := qc;
fin
```

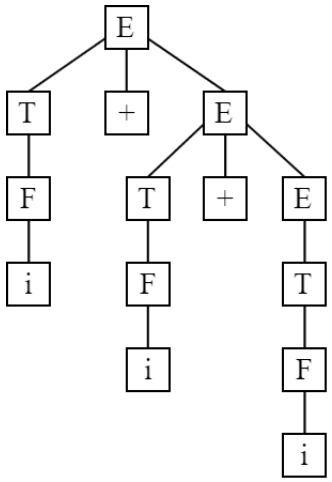
1.5 Les expressions arithmétiques

Soit la grammaire G définie par <T,N,E,P> tels que T={id + * ( )} ; N={E,T,F} et P

$$\begin{cases} E \rightarrow T + E \mid T \\ T \rightarrow F * T \mid T \\ F \rightarrow id \mid (E) \end{cases}$$

L'associativité des opérateurs d'addition et de multiplication se fait de droite à gauche.

Exemple 42 *Soit l'expression arithmétique i+i+i. L'arbre syntaxique généré pour l'expression est donné par la figure suivante :*



L'arbre illustre l'associativité droite de l'opérateur +. Comme les opérateurs d'addition et de multiplication sont associatifs à gauche, il est nécessaire de transformer les règles de production de la grammaire G comme suit :

$$\begin{cases} E \rightarrow T\{+T\}^* \\ T \rightarrow F\{*F\}^* \\ F \rightarrow id \mid (E) \end{cases}$$

Insertion des procédures

$$\begin{cases} E \rightarrow T\{+T < A >\}^* \\ T \rightarrow F\{*F < B >\}^* \\ F \rightarrow id \mid (E) \\ < A > \rightarrow \epsilon \\ < B > \rightarrow \epsilon \end{cases}$$

1.5.1 Analyse syntaxico-sémantique par la descente récursive

Procédure E (X : entité ; Y : type ; Erreur :booléen)
E → T{+T<A>}*

```
début
    Var :
    Opérande1, Opérande2 : Entité ;
    Type1, Type2 : Type ;
    T(Opérande1, Type1,Erreur) ;
    si (Erreur=vrai) alors
        Aller à fin
    fin
    tant que (tc='+') faire
        tc := ts ;
        T(Opérande2, Type2,Erreur) ;
        si (Erreur=vrai) alors
            Aller à Fin
        fin
        A(Opérande1, Type1, Opérande2, Type2) ;
    fin
    X := Opérande1 ;
    Y := Type1 ;
fin
```

Procédure T (X : Entité ; Y : Type ; Erreur :booléen)

T → F{*F<B>}*

```
début
    Var :
    Opérande1, Opérande2 : Entité ;
    Type1, Type2 : Type ; F(opérande1, Type1, Erreur) ;
    si (Erreur=vrai) alors
        Aller à Fin
    fin
    tant que (tc='*') faire
        tc := ts ;
        F(Opérande2, Type2,Erreur) ;
        si (Erreur=vrai) alors
            Aller à Fin
        fin
        B(Opérande1, Type1, Opérande2, Type2) ;
    fin
    X := Opérande1 ;
    Y := Type1 ;
fin
```

Procédure F (X : Entité ; Y : Type ; Erreur :booléen)

F → id | (E)

```
début
    si (tc = '(') alors
        tc := ts;
        E(X, Y,Erreur);
        si (Erreur=vrai) alors
            Aller à Fin
        fin
        si (tc =')') alors
            tc := ts;
        fin
        sinon
            Erreur := vrai;
            Écrire ("Erreur : ) expected"
        fin
    fin
    sinon
        lookup(tc, P);
        si (p ≠ 0) alors
            X := P.nom;
            Y := P.Type;
            tc := ts;
        fin
        sinon
            Erreur := vrai;
            Écrire ("Erreur : Identificateur non déclaré")
        fin
    fin
fin
```

Procédure <A> (opd1, opd2 : Entité ; Type1, Type2 : Type)

- Vérifier compatibilité des types des deux opérandes ;
- CréerTemp(R) ;
- Générer(+, opd1, opd2, R) ;
- Calcul des types des deux opérandes : CalculType(Type1, Type2, R-Type).

```
début
    si (Type1 et Type2 sont compatibles) alors
        CréerTemp(R);
        QUAD(qc) := (+, opd1, opd2, R);
        qc := qc+1;
        CalculType(Type1, Type2, R.Type);
        opd1 := R;
        Type1 := R.Type;
    fin
    sinon
        Erreur :=Vrai;
        Écrire ("Erreur : Incompatibilité de type");
    fin
fin
```

Procédure <B> (opd1, opd2 : entité ; Type1, Type2 : Type)

- Vérifier compatibilité des types des deux opérandes ;
- CréerTemp(R) ;
- Générer(*, opd1, opd2, R) ;
- Calcul des types des deux opérandes : CalculType(Type1, Type2, R.Type).

```
début
    si (Type1 et Type2 sont compatibles) alors
        CréerTemp(R);
        QUAD(qc) := (*, opd1, opd2, R);
        qc := qc+1;
        CalculType(Type1, Type2, R.Type);
        opd1 := R;
        Type1 := R.Type;
    fin
    sinon
        Erreur := Vrai;
        Écrire ("Erreur : Incompatibilité de type");
    fin
fin
```

2 Traduction dirigée par la syntaxe dans le cas de l'analyse ascendante :

Dans le cas ascendant, une routine ne peut être exécutée qu'au moment de la réduction. Il faudra ainsi transformer la grammaire en opérant des opérations de découpage. Soit la règle :

$$\text{<A>} \rightarrow \alpha \underset{\underset{routine}{\uparrow}}{} \beta \text{ avec A} \in N, \alpha, \beta \in (T \cap N)^*$$

Si une routine doit être insérée entre α et β alors la règle sera transformée comme suit :

$$\text{<A>} \rightarrow \text{<B>}\ \beta$$
$$\text{<B>} \rightarrow \alpha$$

2.1 Instruction conditionnelle

La syntaxe de l'instruction conditionnelle est définie par la règle syntaxique suivante :

<inst-if> → IF <Condition> THEN <Instruction1> ELSE <Instruction2>

Le schéma de traduction de l'instruction en utilisant le code post-fixé est alors définie par :

1. La Grammaire associée :
   <inst-if> → <debut-inst-if> ELSE <Instruction2> (R3)
   <debut-inst-if> → <debut-if> THEN <Instruction1> (R2)
   <debut-if> → IF <Condition> (R1)
2. Le code intermédiaire en utilisant la forme post-fixée :
   FP(<Condition>) (else) BZ FP(<Instruction1>) (fin) BR↓FP(<Instruction2>)
   ↑
3. Les routines sémantiques :
   Routine (R1)

```
début
    sauv_bz := i ;
    i :=i+1 ;
    PF[i] := 'BZ' ;
    i :=i+1 ;
fin
```

Routine (R2)

```
début
    sauv_br := i ;
    i :=i+1 ;
    PF[i] := 'BR' ;
    i :=i+1 ;
    PF[sauv_bz] := i ;
fin
```

Routine (R3)

```
début
    PF[sauv_br] := i ;
fin
```

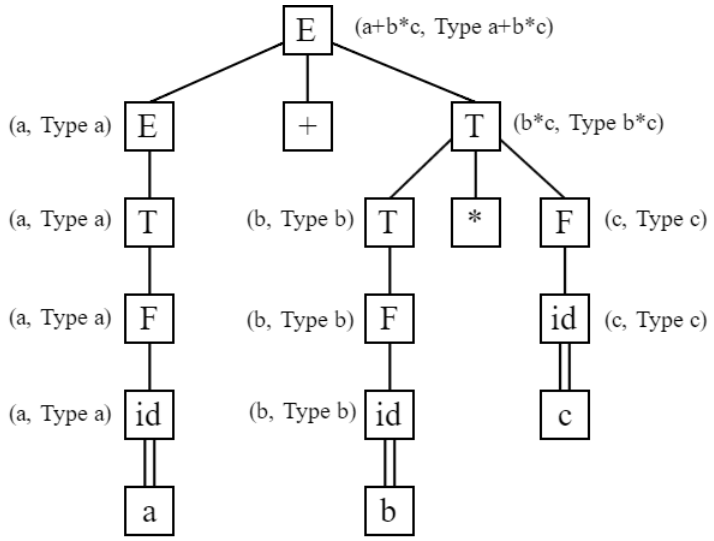
Remarque 31 *La représentation du code post-fixé est matérialisée en utilisant un vecteur PF tel que i représente la 1ère position libre dans le vecteur PF.*

2.2 Traduction des expressions arithmétiques sous forme de quadruplets dans le cas ascendant

La grammaire syntaxique qui génère les expressions arithmétiques est définie par <T,N,E,P> tels que T={id,+,*,(,)} ; N={E,T,F} et P :

$$\begin{cases} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow id \mid (E) \end{cases}$$

Exemple 43 *Soit l'expression arithmétique a+b*c : L'évaluation de cette expression est représentée par l'arbre syntaxique suivant :*



Les attributs concernant les opérandes sont stockés dans la table des symboles. Dans le cas d'une analyse ascendante, des réduction sont effectuées. Il est ainsi nécessaire de sauvegarder les informations sur les opérandes dans une pile d'analyse à trois champs appelée *pile d'opérandes* :

1. le 1er champ contient le MGP (Membre Gauche de Production),
2. le 2ème champ contient le nom,
3. le 3ème champ contient le type.

À chaque symbole de la grammaire sont associés les attributs nom et type des opérandes, pour la traduction des expressions arithmétiques.

- La grammaire augmentée associée est donnée par :

$$\begin{cases} E \rightarrow E + T\,(\text{R6}) \mid T\,(\text{R5}) \\ T \rightarrow T * F\,(\text{R4}) \mid F\,(\text{R3}) \\ F \rightarrow id\,(\text{R1}) \mid (E)\,(\text{R2}) \end{cases}$$

  Comme toutes les routines sont placées à la fin des membres droits des règles de production, il n'est pas nécessaire de découper les règles.

- Les routines sémantiques sont définies comme suit :
  Routine (R1)

```
/* permet de reconnaitre l'identificateur, de le réduire à F et d'empiler ses
  attributs */
début
    lookup(id, P) ;
    si (P = 0) alors
        erreur("idf non déclaré")
    fin
    sinon
        empiler(pile-opérandes, F, P.nom, P.type)
    fin
fin
```

Routine (R2)

```
/* permet de réduire (E) en F et d'empiler ses attributs */
début
    Opérande := dépiler(Pile-Opérandes) ;
    empiler(Pile-Opérandes, F, Opérande.nom, Opérande.type) ;
fin
```

Routine (R3)

```
/* permet de réduire F en T et d'empiler ses attributs */
début
    Opérande := dépiler(Pile-Opérandes) ;
    empiler(Pile-Opérandes, T, Opérande.nom, Opérande.type) ;
fin
```

Routine (R4)

```
/* permet d'effectuer la multiplication et de ranger le résultat */
début
    Opérande1 := dépiler(Pile-Opérandes);
    Opérande2 := dépiler(Pile-Opérandes);
    si (Opérande1.type et Opérande2.type sont incompatibles) alors
        erreur("incompatibilité des types");
    sinon
        CréerTemp(R);
        QUAD(qc) := (*, Opérande1.nom, Opérande2.nom, R);
        qc := qc+1;
        CalculType(Opérande1.type, Opérande2.type, R.type);
        empiler(Pile-Opérandes, T, R, R.type);
    fin
fin
```

Routine (R5)

```
/* permet de réduire T en E, et d'empiler ses attributs */
début
    opérande := dépiler(Pile-Opérandes);
    empiler(pile-opérandes, E, opérande.nom, opérande.type);
fin
```

Routine (R6)

```
/* permet d'effectuer l'addition et de ranger le résultat */
début
    Opérande1 := dépiler(Pile-Opérandes);
    Opérande2 := dépiler(Pile-Opérandes);
    si (Opérande1.type et Opérande2.type sont incompatibles) alors
        erreur("incompatibilité des types");
    sinon
        CréerTemp(R);
        QUAD(qc) := (+, Opérande1.nom, Opérande2.nom, R);
        qc := qc+1;
        CalculType(Opérande1.type, Opérande2.type, R.type);
        empiler(Pile-Opérandes, E, R, R.type);
    fin
fin
```

2.3 Traduction des expressions booléennes sous forme de quadruplets dans le cas ascendant

La grammaire syntaxique G qui génère les expressions booléennes est définie par <T,N,EL,P> tels que T={id,true, false, and, or, not } ; N={EL} et P :

$$\{EL \rightarrow EL\,or\,EL \mid EL\,and\,EL \mid not\,EL \mid id \mid true \mid false$$

La grammaire G est ambiguë. Afin de tenir compte des priorités entre les opérateurs, la grammaire G est transformée en une grammaire G' équivalente non ambiguë définie par <T,N,EL,P'> tels que T={id,true, false, and, or, not} ; N={EL, TL, FL} et P' :

$$\begin{cases} EL \rightarrow EL\,or\,TL \mid TL \\ TL \rightarrow TL\,and\,FL \mid FL \\ FL \rightarrow id \mid true \mid false \mid not\,EL \end{cases}$$

- La grammaire augmentée associée est donnée par :

$$\begin{cases} EL \rightarrow EL\,or\,TL(\text{R7}) \mid TL(\text{R8}) \\ TL \rightarrow TL\,and\,FL(\text{R5}) \mid FL(\text{R6}) \\ FL \rightarrow id(\text{R1}) \mid true(\text{R2}) \mid false(\text{R3}) \mid not\,EL(\text{R4}) \end{cases}$$

  Comme toutes les routines sont placées à la fin des membres droits des règles de production, il n'est pas nécessaire de découper les règles.

- Les routines sémantiques sont définies comme suit :
  Routine (R1)

```
début
    lookup(id, P) ;
    si (P = 0) alors
        erreur("idf non déclaré")
    sinon
        si (P.type ≠ 'boolean') alors
            erreur("Le type de l'identificateur doit être booléen")
        sinon
            FL.nom :=P.nom ;
            FL.type :=P.type ;
        fin
    fin
fin
```

Routine (R2)

```
début
    FL.nom :=true ;
    FL.type :='Boolean' ;
fin
```

Routine (R3)

```
début
    FL.nom :=false ;
    FL.type :='Boolean' ;
fin
```

Routine (R4)

```
début
    si (EL.type ≠ 'boolean') alors
        erreur("Le type de l'expression doit être booléen")
    sinon
        CréerTemp(R) ;
        Quad(qc) :=(BZ,qc+3, ,EL.nom) ;
        qc :=qc+1 ;
        Quad(qc) :=( :=,0, ,R) ;
        qc :=qc+1 ;
        Quad(qc) :=(BR, qc+2, , ) ;
        qc :=qc+1 ;
        Quad(qc) :=( :=, 1, , R) ;
        qc :=qc+1 ;
        FL.nom :R ;
        FL.type :='Boolean' ;
    fin
fin
```

Routine (R5)

```
début
    si (TL.type='Boolean') et (FL.type='boolean') alors
        CréerTemp(R);
        Quad(qc) :=('BZ',qc+4,TL.nom, );
        qc :=qc+1;
        Quad(qc) :=('BZ', qc+3, FL.nom, );
        qc :=qc+1;
        Quad(qc) :=( :=,1, , R) :
        qc :=qc+1;
        quad(qc) :=('BR', qc+2, , );
        qc :=qc+1;
        Quad(qc) :=( :=,0, ,R);
        qc :=qc+1;
        TL.nom :=R;
        TL.type := 'Boolean';
    sinon
        erreur("incompatibilité des types");
    fin
fin
```

Routine (R6)

```
début
    TL.nom :=FL.nom;
    TL.type :=FL.type;
fin
```

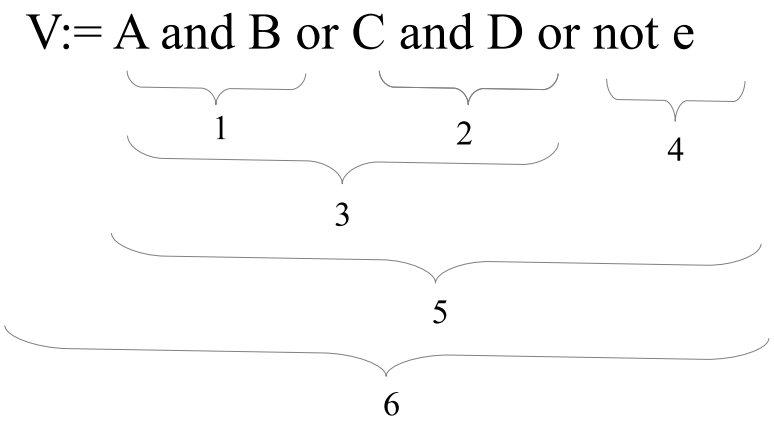
Routine (R7)

```
début
    si (EL.type='Boolean') et (TL.type='boolean') alors
        CréerTemp(R);
        Quad(qc) :=('BNZ',qc+4,EL.nom, );
        qc :=qc+1;
        Quad(qc) :=('BNZ', qc+3, TL.nom, );
        qc :=qc+1;
        Quad(qc) :=( :=,0, , R) :
        qc :=qc+1;
        quad(qc) :=('BR', qc+2, , );
        qc :=qc+1;
        Quad(qc) :=( :=,1, ,R);
        qc :=qc+1;
        EL.nom :=R;
        EL.type := 'Boolean';
    sinon
        erreur("incompatibilité des types");
    fin
fin
```

Routine (R8)

```
début
    EL.nom :=TL.nom;
    EL.type :=TL.type;
fin
```

Exemple 44 *Soit l'instruction d'affectation suivante :*
V := A and B or C and D or not E ;
L'ordre d'évaluation des sous expressions booléennes est comme suit :



Le code généré par les routines sémantiques est le suivant :
1- (BZ, (5), A,) ;
2- (BZ, (5), B,) ;
3- (:=,1, , T1) ;
4- (BR, (6), ,) ;
5- (:=, 0 , ,T1) ;
6- (BZ, (10),C,) ;
7- (BZ, (10), D,) ;
8- (:=,1, , T2) ;
9- (BR, (11), ,) ;
10- (:=,0, , T2) ;
11- (BNZ,(15), T1,) ;
12- (BNZ,(15), T2,) ;
13-(- :=, 0, ,T3) ;
14- (BR, (16), ,) ;
15- (:=, 1, , T3) ;
16-(BZ, (19), E,) ;
17- (:=,0, ,T4) ;
18- (BR, (20), ,) ;
19-(:=, 1, , T4) ;
20-(BNZ,(24),T3,) ;
21- (BNZ,(24),T4,) ;
22-(:=,0, , T5) ;
23-(BR, (25), ,) ;
24-(:=, 1, , T5) ;

25-(:=,T5, , V);

2.4 Instruction de branchement inconditionnel (Traitement des étiquettes)

L'instruction du branchement inconditionnel à une étiquette se présente comme suit :

GOTO étiquette

Une étiquette doit être déclarée, référencée et utilisée. Deux cas peuvent se présenter lors de la référence à une étiquette :

Cas 1 :

étiquette : <instruction> } définition
.
.
.
GOTO étiquette } référence
.
.
.
GOTO étiquette

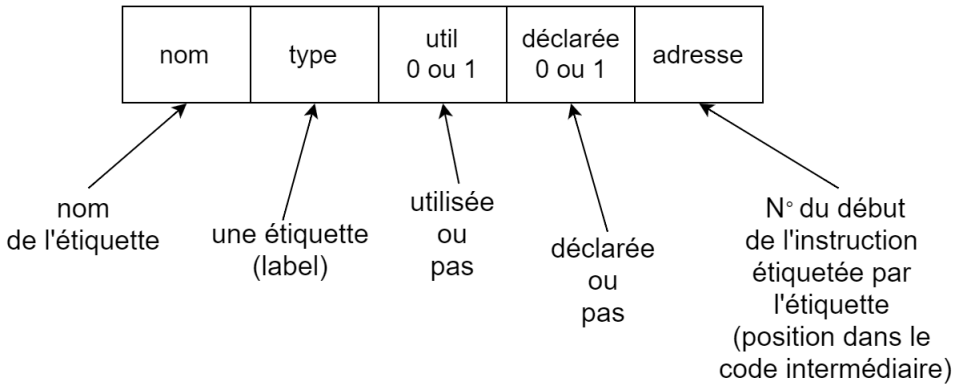
Cas 2 :

GOTO étiquette } référence
.
.
.
GOTO étiquette
.
.
.
étiquette : <instruction> } définition

— Dans le 1er cas, l'étiquette est utilisée avant d'être référencée, donc l'opérateur BRL peut être directement remplacé par BR en faisant référence à l'étiquette.

— Dans le 2ème cas, la définition de l'étiquette n'est pas connue lors des instructions de référence. Il n'est donc pas possible de traduire directement l'instruction GOTO en l'instruction BR.

Organisation de la T.S dans le cas des étiquettes



Il est nécessaire d'utiliser la fonction *lookup(nom,P)* qui permet de chercher une entité dans la T.S et de la fonction *insert(nom,P)* qui permet d'insérer une entité dans la T.S. Ces fonctions ont comme paramètres d'entrée le nom de l'identificateur et retournent l'indice P dans la table des symboles.
Il existe deux méthodes de traduction de l'instruction du branchement inconditionnel :

- **Utilisation du code BRL P**
  Cette méthode consiste à traduire le *GOTO étiquette* en *BRL P* où *P* est l'entrée de l'étiquette dans la T.S. Deux cas se présentent :
  — l'étiquette est déjà définie au moment de la référence. Dans ce cas, l'instruction *GOTO étiquette* est traduite en *BR P.adresse*,
  — l'étiquette est non connue au moment de la référence. Dans ce cas, l'instruction *GOTO étiquette* sera traduite en une première passe en *BRL P*. Ce code sera alors traduit lors d'une 2ème passe en *BR P.adresse*.
- **La méthode de chainage des références**
  Dans cette méthode , le code intermédiaire associé au branchement inconditionnel est BR au lieu de BRL. L'ensemble des codes générés BR faisant référence à la même étiquettes seront reliés entre eux en utilisant la méthode de chainage dan on relie les BR faisant référence à la même étiquette par la méthode de chainage en utilisant une liste. Cette liste est créée en utilisant le 2ème champ du quadruplet de BR et le champ adresse de l'étiquette de la TS. Le champ adresse va contenir le nº du dernier quadruplet ayant fait référence à l'étiquette.

Exemple 45 *Soient les instruction de référence à l'étiquette fin suivantes :*

```
.
.
GOTO fin        20- (BR, 0, , ) P.adresse = 20
.                          ↓
.
GOTO fin        35- (BR, 20 , , ) P.adresse = 35
.                          ↓
.
GOTO fin        65- (BR, 35, , ) P.adresse = 65
.
.
fin : <inst>    83- <inst> ⟹ mise à jour des adresses des BR
                              par l'adresse de l'étiquette fin (83)
```

À la rencontre d'une instruction GOTO vers une étiquette, un quadruplet est généré ayant comme opérateur 'BR' et comme adresse le numéro du quadruplet associé au dernier GOTO rencontré.

À la rencontre de l'instruction : *étiquette : instruction*, les adresses des quadruplets correspondants au code BR générés au préalable sont mises à jour. Le schéma du traduction pour la déclaration, la référence et l'utilisation de l'étiquette sous forme de quadruplets est défini par :

(a) La grammaire associée :

<dec-etiq>→ Label <list-etiq>
<list-etiq> → <list-etiq>, etiq (R1) | etiq (R1) } **déclaration**
<ref-etiq> → GOTO etiq (R2) } **référence**
<inst-etiq>→ etiq : (R3) <inst> } **utilisation**

Comme certaines routines sémantiques ne sont pas positionnées à la fin des membres droits des règles de production, il est nécessaire de procéder au découpage des règles comme suit :

<dec-etiq> → Label <list-etiq>
<list-etiq> → <list-etiq>, etiq (R1) | etiq (R1)
<ref-etiq> → GOTO etiq (R2)
<inst-etiq> → <debut-inst> <inst>
<debut-inst>→ etiq : (R3)

(b) Les routines sémantiques :
Routine (R1) : /* Elle permet d'insérer l'étiquette dans la TS*/

```
début
    lookup(tc, P);
    si (P=0) alors
        insert(tc, P);
        P.declare := 1;
        P.util := 0;
        P.type := 'label';
        P.adresse := 0;
    sinon
        Écrire("Erreur :double déclaration");
fin
```

Routine (R2) /*Elle fait référence à une étiquette*/

```
début
    lookup(tc, P);
    si (P ≠ 0) alors
        si (P.util=1) alors
            //étiquette déjà utilisée
            QUAD(qc) := (BR, P.adresse, , );
        sinon
            //étiquette non utilisée
            QUAD(qc) := (BR, P.adresse, , );
            P.adresse := qc;
        qc := qc+1;
    sinon
        Écrire("Erreur :étiquette non déclarée");
fin
```

Routine (R3) /* Elle correspond à l'utilisation d'une étiquette*/

```
début
    lookup(tc, P);
    si (P ≠ 0) alors
        si (P.util ≠ 1) alors
            a := P.adresse;
            P.util := 1;
            tant que (a ≠ 0) faire
                b := QUAD(a, 2);
                QUAD(a, 2) := qc
                a := b;
            fin
            P.adresse := qc;
        sinon
            Écrire("Erreur : double utilisation");
    sinon
        Écrire("Erreur : étiquette non déclarée");
fin
```

3 exercices

Exercice 1 :

Soit l'instruction suivante :
Si <exparithm> = neg <suite1> nul <suite2> pos <suite3> finisi

La sémantique de l'insruction est comme suit :

> Si *exparithm* est négative, exécuter suite1 et aller à finisi,
> Sinon si *exparithm* est nul, exécuter suite2 et aller à finisi,
> Si elle est positive, exécuter suite3.

1. Donnez la grammaire syntaxique.
2. Donnez le schéma de traduction sous forme de quadruplets dans le cas d'une analyse ascendante.

Exercice 2 :

Soit l'instruction :
Id := moyenne (<exp1>,<exp2>,... <expn>)
Elle consiste à affecter à l'identificateur la moyenne des n expressions arithmétiques.

1. Donnez la grammaire syntaxique.
2. Donnez le schéma de traduction sous forme de quadruplets, dans le cas d'un analyse ascendante.

Exercice 3 :

Soit l'instruction suivante :
Select (instruction1, instruction2) expression
La sémantique associée à cette instruction est comme suit :

> Si l'expression est vraie, exécuter instruction1 et sortir.
> Si l'expression est fausse, exécuter instruction2 et sortir.

Donnez le schéma de traduction sous forme post-fixée, dans le cas d'une analyse descendante.

Exercice 4 :

Soit l'instruction suivante :
<Tnstcombineloop> → COMBINELOOP(<Instruction1>,<Condition1>;<Instruction2>,<Condition2>;<Instruction3>) La sémantique associée à cette instruction est comme suit :

> Si la condition1 est vraie alors seule l'instruction1 est exécutée. Sinon, si la condition2 est vraie alors seule l'instruction2 est exécutée. Sinon, l'instruction3 est exécutée puis il y a un branchement vers le début de l'instruction COMBINELOOP.

Donnez le schéma de traduction sous forme se quadruplets, dans le cas d'une analyse ascendante.

Chapitre 7

bison

1 introduction

Comme pour l'analyse lexicale, divers outils ont été mis au point afin de construire des analyseurs syntaxiques à partir de grammaires. Leurs principales tâches consistent à construire automatiquement une table d'analyse à partir d'une grammaire donnée. YACC(Yet Another Compiler Compiler) est un des utilitaires syntaxiques. La version libre de YACC est intégrée dans le système LINUX en tant que package de base du système. Ce freeware est téléchargeable à partir de http ://www.gnu.org/software/bison. Bison accepte en entrée la description syntaxique du langage en termes de règles de productions et génère un analyseur syntaxique écrit en langage C. Vu les performances de l'analyse ascendante du type LALR, BISON manipule uniquement des grammaires LALR(1) en utilisant le modèle de decallages/réductions. Le processus de fonctionnement de l'utilitaire BISON est donné par la figure suivante :

Remarque 7.1 *Pour les systèmes Windows, FLEX et BISON sont disponibles dans l'émulateur LINUX CYGWIN à partir de http ://www.cygwin.com.*

Bison crée donc un analyseur LALR(1), auquel des actions sémantiques peuvent y être intégrées.

2 structure d'un fichier Bison

Exemple 7.1 *Le fichier Flex décrit ci-après correspond à un mini interprète des expressions arithmétiques formées par des nombres réels et des opérateurs arithmétique classiques .*

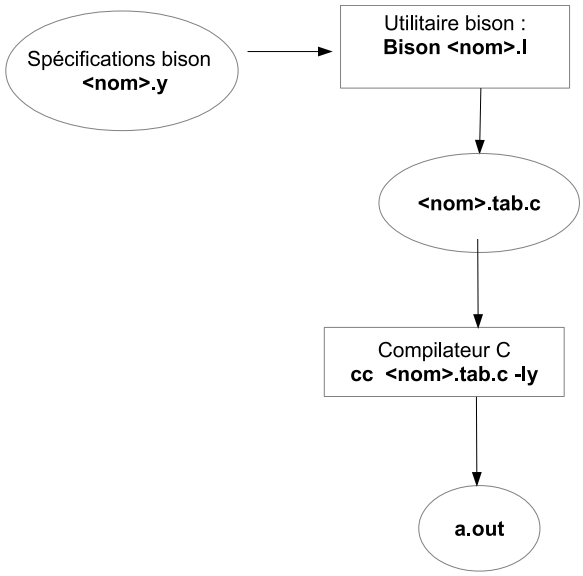


FIGURE 7.1 – Processus de fonctionnement de Bison

%{ #include "global.h" #include "calc.tab.h" #include <stdlib.h> %} blancs [ ]+ chiffre [0-9] entier chiffre+ exposant [eE][+-]?entier reel entier("."entier)?exposant? %% blancs / ignoré */ reel yylval=atof(yytext); return(NOMBRE); %% int yywrap() return 1;*

Le fichier de description bison correspondant sera défini comme suit :
%{

#include "global.h" #include <stdio.h> #include <stdlib.h> #include <math.h> %} token NOMBRE

Input : / Vide */ | Ligne Input ; Ligne : FIN | Expression FIN printf("Résultat : %f*
*n",$1); ; Expression : int yyerror(char *s) printf("%s*
n",s); int main(void) yyparse();
Il faut ajouter un fichier global.h qui contiendra :
#define YYSTYPE double extern YYSTYPE yylval;
La compilation se produira en trois étapes :

- *analyse lexicale avec flex : flex calc.lex*
- *traduction dirigée par la syntaxe avec Bison : bison -d calc.y*
- *Compilation du code C :cc calc.tab.c lex.yy.c –o calc*

3 Gestion des conflits

Pour les analyseurs LALR(1), deux types de conflits peuvent être détectés : le conflit réduction/réduction et le conflit décalage/réduction.

1. Le premier cas de conflit se produit lorsque le compilateur a le choix entre au moins deux productions différentes pour réduire une chaîne.
2. Le second cas de conflit apparait lorsque le compilateur doit sélectionner une action entre réduire une chaine en un non terminal en utilisant une règle de production et décaler le pointeur sur la chaîne d'entrée d'une entité lexicale.

Pour l'analyseur LALR(1) Bison, les éventuels conflits sont signalés comme suit :

> bison test.c
conflicts : 4 reduce/reduce, 3 shift/reduce.

— Pour le cas reduce/reduce, la règle sélectionnée est celle qui apparait en premier dans la partie spécifications.

— pour le cas de conflit shift/reduce qui correspond à choisir entre une réduction $A \rightarrow \alpha$ avec $\alpha \in \mathcal{N}$ et un décalage d'un symbole a, le conflit est traité comme suit :

  — si la priorité de la production est plus grande que celle accordé au symbole a, alors la réduction est sélectionnée,

  — si les deux actions ont la même priorité et si la production est associative à gauche, alors la réduction est sélectionnée.

  — pour les autres cas, c'est le décalage qui est sélectionné.

Remarque 7.2 *La priorité (respectivement l'associativité) d'une production correspondent à la priorité (respectivement à l'associativité) du symbole terminal le plus à droite.*

L'ensemble des cas de conflits rencontrés par BISON lors de l'analyse d'une grammaire est recensé dans la table d'analyse générée par BISON dans le fichier *y.outout*. Ce fichier est obtenu en utilisant l'option -v lors de la compilation. Les conflits peuvent être résolus :

1. en spécifiant explicitement l'ordre de priorités entre symboles terminaux et en affectant des associativités droites ou gauches,

Exemple 7.2 *Soient les déclarations suivantes dans la section définitions :*
%left operateur1 operateur2
%right operateur3
Ces déclarations spécifient que les opérateurs 1 et 2 sont associatifs à gauche et ont la même priorité par contre l'opérateur 3 est associatif à droite.

2. en déclarant les symboles terminaux dans l'ordre des priorités dans l'ordre croissant.

4 Communication Flex-Bison

Étant donnée le processus d'enchaînement des différentes étapes de la compilation, l'analyseur syntaxique BISON et l'analyseur lexical FLEX doivent communiquer. Cette communication est réalisée par l'intermédiaire de la variable *int yyval*. Cette variable sert à récupérer les entités lexicales reconnues par FLEX. Le type par défaut associé à la variable *yyval* est par défaut un entier. Néanmoins, il est possible de personnaliser ce dernier.

5 Écriture d'une grammaire

6 Gestion des erreurs

7 Conclusion

Corrigés des exercices

I- Analyse lexicale

Exercice 1 :

Soit l'automate suivant représenté par la Figure 7.2.

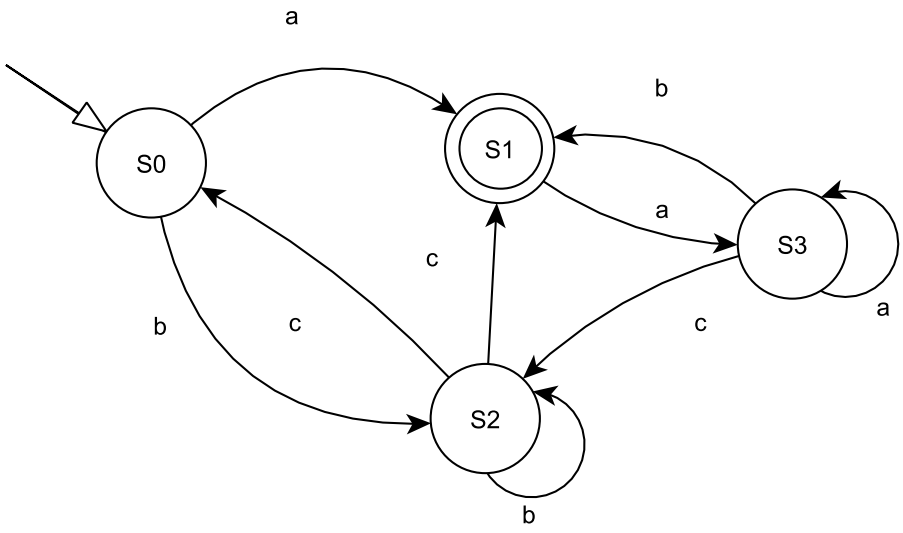


FIGURE 7.2 – Automate de l'exercice 1

Donnez l'automate simple déterministe correspondant.

- La matrice de transition associée à l'automate est définie comme suit :

États \ Terminaux	a	b	c
S0	S1	S2	-
S1	S3	-	-
S2	-	S2	S0S1
S3	S3	S1	S2
S0S1	S1S3	S2	-
S1S3	S3	S1	S2

États finaux

- Afin de rendre l'automate déterministe, il est nécessaire de rajouter deux nouveaux états issus du regroupements de certains états de l'automate non déterministe à savoir l'état S0S1 et l'état S1S3.
  L'ensemble des états finaux F de l'automate déterministe est constitué de tous les états contenant l'état final S1 : F={S1,S0S1 et S1S3}. L'état initial reste inchangé. Ainsi, l'automate déterministe correspondant est représenté par la Figure 7.3.

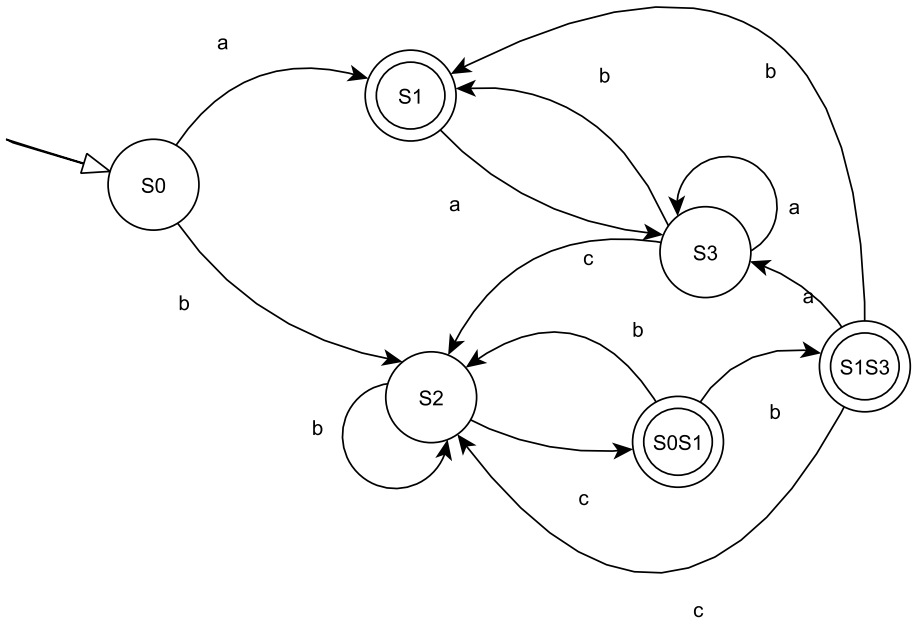


FIGURE 7.3 – Automate déterministe correspondant à l'automate de la Figure 7.2

Exercice 2 :

Il s'agit d'analyser le langage L défini sur $\{0,1\}^*$ et présentant les caractéristiques suivantes :

- Le langage est formé des entités de type X,Y et Z (chaines binaires) ;
- Une entité X commence par '0' et ne contient pas deux '0' consécutifs ;
- Une entité Y commence par un '1' qui est suivi d'un nombre pair de '0' (il y a au moins deux) ;
- Une entité Z commence par deux '0' ou bien deux '1' suivis par une séquence de '0' et de '1' ne contenant la sous chaine '101'.

1. Construire un automate déterministe reconnaissant les entités X,Y et Z.
   L'automate déterministe reconnaissant les entités X,Y et Z est représenté par la Figure 7.4.

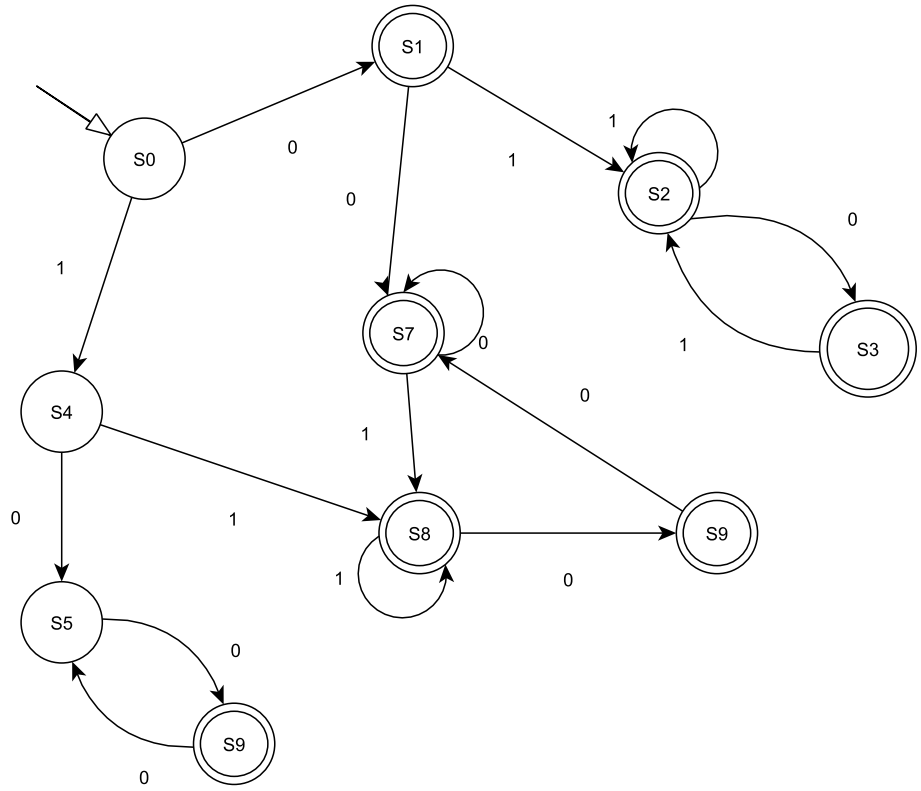


FIGURE 7.4 – Automate déterministe reconnaissant les entités X,Y et Z

L'ensemble F des états finaux est défini par l'ensemble des états finaux reconnaissant les entités X,Y et Z :
$F = F_X \cup F_Y \cup F_Z$
$F_X = \{S_1, S_2, S_3\}$
$F_Y = \{S_6\}$
$F_Z = \{S_7, S_8, S_9\}$

2. Écrire un algorithme qui réalise l'analyse lexicale du langage. L'automate est représenté par la matrice de transition I définie par :

Terminaux / États	0	1
S0	S1	S4
S1	S7	S2
S2	S3	S2
S3	S2	-
S4	S5	S8
S5	S6	-
S6	S5	-
S7	S7	S8
S8	S9	S8
S9	S7	-

La matrice de transition I est mono-définie $\Longrightarrow$ l'automate est déterministe.

```
début
    lire(entité);
    tc ← 1^er caractère de l'entité;
    Ec ← S0;
    tant que ((Ec ≠ ∅) et (tc ≠ #)) faire
        Ec := I[Ec, tc];
        tc := tc + 1;
    fin
    si (Ec = ∅) alors
        Erreur('Entité ∉ langage');
    sinon
        si (Ec ∉ F) alors
            Erreur('Entité ∉ langage');
        sinon
            si (Ec ∈ Ex) alors
                écrire("L'entité reconnue est X");
                codifier(entité,X);
                Insérer(entité);
            sinon
                si (Ec ∈ Ey) alors
                    écrire("L'entité reconnue est Y");
                    codifier(entité,Y);
                    Insérer(entité);
                sinon
                    écrire("L'entité reconnue est Z");
                    codifier(entité,Z);
                    Insérer(entité);
fin
```

8 II-Analyse syntaxique

II-1 Méthodes descendantes

Exercice 1 :

Soit la grammaire G définie par <T,ℕ,S,P> tels que :
T={ a b ( ) ,}
N={S,T} et

$$P : \begin{cases} S \rightarrow a \mid b(T) \\ T \rightarrow T,S \mid S \end{cases}$$

1. Donnez l'arbre de dérivation associé à la chaine b(a,b(a,a)).
   L'arbre de dérivation associé à la chaine *b(a,b(a,a))* est représenté par la Figure 7.5.

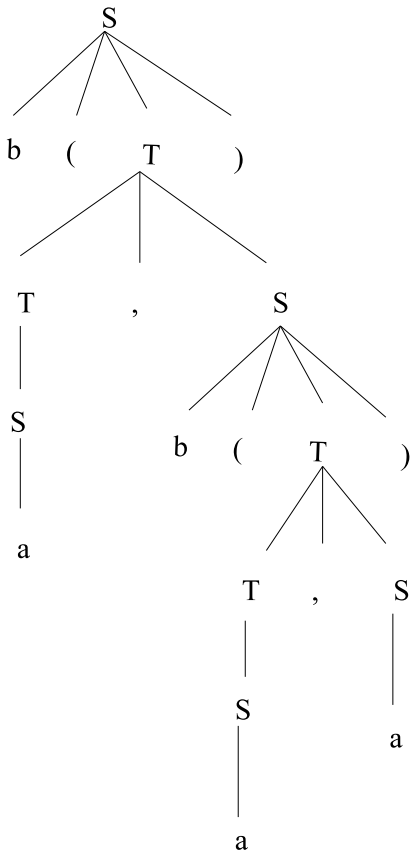


FIGURE 7.5 – Arbre de dérivation de la chaine *b(a,b(a,a))*

2. Éliminez la récursivité gauche dans G.
   G n'est pas LL(1) car il y a une récursivité à gauche directe en T.

Rappel

L'élimination de la récursivité gauche directe revient à transformer les règles de production du non terminal A du type :

$$\text{A} \rightarrow \text{A}\alpha_1 \mid ... \mid A\alpha_n \mid \beta_1 \mid ... \mid \beta_m \text{ avec A} \in \mathbb{N} \text{ et } \beta_i, \alpha_i \in (T \cup N)^* \Leftrightarrow$$

$$P' : \begin{cases} A \rightarrow \beta_1 A' \mid ... \mid \beta_m A' \mid \beta_1 \mid ... \mid \beta_m \\ A' \rightarrow \alpha A' \mid ... \mid \alpha_n A' \mid \alpha_1 \mid ... \mid \alpha_n \end{cases}$$

ou encore

$$P' : \begin{cases} A \rightarrow \beta_1 A' \mid ... \mid \beta_m A' \\ A' \rightarrow \alpha_1 A' \mid ... \mid \alpha_n A' \mid \epsilon \end{cases}$$

Ainsi, l'application des règles de transformation aux règles associées au non terminal T de la grammaire G, produit les règles suivantes :

$$\begin{cases} T \rightarrow S \mid ST' \\ T' \rightarrow, ST' \mid , S \end{cases}$$

ou encore :

$$\begin{cases} T \rightarrow ST' \\ T' \rightarrow, ST' \mid \epsilon \end{cases}$$

Remarque 32 *La seconde transformation produit des règles factorisées contrairement à la première où il serait nécessaire d'opérer l'opération de factorisation comme suit :*

$$\begin{cases} T \rightarrow SA \\ A \rightarrow T' \mid \epsilon \\ T' \rightarrow, SA \end{cases}$$

En utilisant la seconde transformation, la grammaire G est transformée en une grammaire équivalente G' définie par <T,$\mathbb{N}'$,S,P'> tels que :
T={ a b ( ) ,}
N={S,T,T'} et

$$P' : \begin{cases} S \rightarrow a\textcircled{1} \mid b(T)\textcircled{2} \\ T \rightarrow ST'\textcircled{3} \\ T' \rightarrow, ST'\textcircled{4} \mid \epsilon\textcircled{5} \end{cases}$$

Les ensembles des Débuts et des Suivants associés aux non terminaux de la grammaire G' sont donnés par la table suivante :

	Débuts	Suivants
S	a b	# ,)
T	a b	)
T'	, ϵ	)

3. La grammaire G' obtenue est-elle LL(1) ?
   La grammaire G' est LL(1) car :

   a) G' est non récursive à gauche,

   b) G' est factorisée,

   c)
   - pour les règles S → a |b(T) :
     Débuts(a) ∩ Débuts(b(T)) = {a} ∩ {b} = ∅
     ⟹ Les règles associées au non terminal S vérifient les conditions LL(1).

   - pour les règles T' → ,ST' |ϵ
     Débuts(,ST') ∩ Suivants(T') = ∅
     ⟹ Les règles associées au non terminal T' vérifient les conditions LL(1).

   La table d'analyse LL(1) M associée à la grammaire G' est définie par :

	a	b	(	)	,	#
S	R1	R2				
T	R3	R3				
T'				R5	R4	

   La table d'analyse LL(1) est mono-définie ⟹ G' est LL(1).

4. Analyse de la chaîne b(a,a)#

Pile	chaine	action
#S	b(a,a)#	R2 :S → b(T) Dépiler() ; Empiler(miroir(b(T)))
#)T(b	b(a,a)#	Égalité(Sp,Ec) : Dépiler() ; Avancer() ;
#)T(	(a,a)#	Égalité(Sp,Ec) : Dépiler() ; Avancer()
#)T	a,a)#	R3 :T → ST' Dépiler() ; Empiler(miroir(ST'))
#)T'S	a,a)#	R1 :S → a Dépiler() ; Empiler (miroir(a))
#)T'a	a,a)#	Égalité(Sp,Ec) : Dépiler() ; Avancer()
#)T'	,a)#	R4 :T' → ,ST' Dépiler() ; Empiler(miroir(,ST'))
#)T'S,	,a)#	Égalité(Sp,Ec) : Dépiler() ; Avancer()
#)T'S	a)#	R1 :S → a Dépiler() ; Empiler(miroir(a))
#)T'a	a)#	Égalité(Sp,Ec) : Dépiler() ; Avancer()
#)T'	)#	R5 :T' → ϵ Dépiler() ;
#)	)#	Égalité(Sp,Ec) : Dépiler() ; Avancer()
#	#	La chaîne b(a,a)# est acceptée

Remarque 33 *Sp désigne le symbole de la grammaire en sommet de la pile utilisée par l'analyseur syntaxique LL(1).*
Ec désigne l'entité courante de la chaîne.

L'arbre de dérivation associé à la chaine *b(a,a)* est représenté par la Figure 7.8.

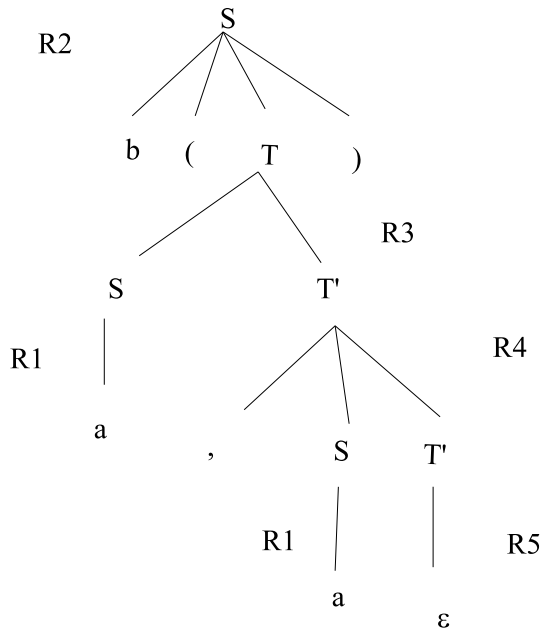


FIGURE 7.6 – Arbre de dérivation de la chaine *b(a,b(a,a))#*

5. Analyse de la chaîne b(a,(a,b))#.

Pile	chaine	action
#S	b(a,(a,b))#	R2 :S → b(T) Dépiler() ; Empiler(miroir(b(T))
#)T(b	b(a,(a,b))#	Égalité(Sp,Ec) : Dépiler() ; Avancer() ;
#)T(	(a,(a,b))#	Égalité(Sp,Ec) : Dépiler() ; Avancer() ;
#)T	a,(a,b))#	R R3 :T → ST' Dépiler() ; Empiler(miroir(ST'))
#)T'S	a,(a,b))#	R1 :S → a Dépiler() ; Empiler(miroir(a))
#)T'a	a,(a,b))#	Égalité(Sp,Ec) : Dépiler() ; avancer() ;
#)T'	,(a,b))#	R4 :T' → ,ST' Dépiler() ; Empiler(miroir(,ST'))
#)T'S,	,(a,b))#	Égalité(Sp,Ec) : Dépiler() ; Avancer() ;
#)T'S	(a,b))#	M[S,(] = ∅ ⟹ Erreur ('a ou b expected')

Ainsi, la chaine b(a,(a,b))# n'appartient pas au langage.

Exercice 2 :

Soit la grammaire G définie par <T,ℕ,L,P> tels que :
T={id = print ( ) + - num int float}
ℕ={L,I,E,T} et

$$P : \begin{cases} L \rightarrow IL \mid \epsilon \\ I \rightarrow T\ id \mid id = E \mid print(E) \\ E \rightarrow E + E \mid E - E \mid id \mid num \\ T \rightarrow int \mid float \end{cases}$$

1. Donnez une arbre de dérivation pour la chaine *print(id+id+id)*.
   La chaîne *print(id+id+id)* admet deux arbres syntaxiques représentés par la Figure 7.7.

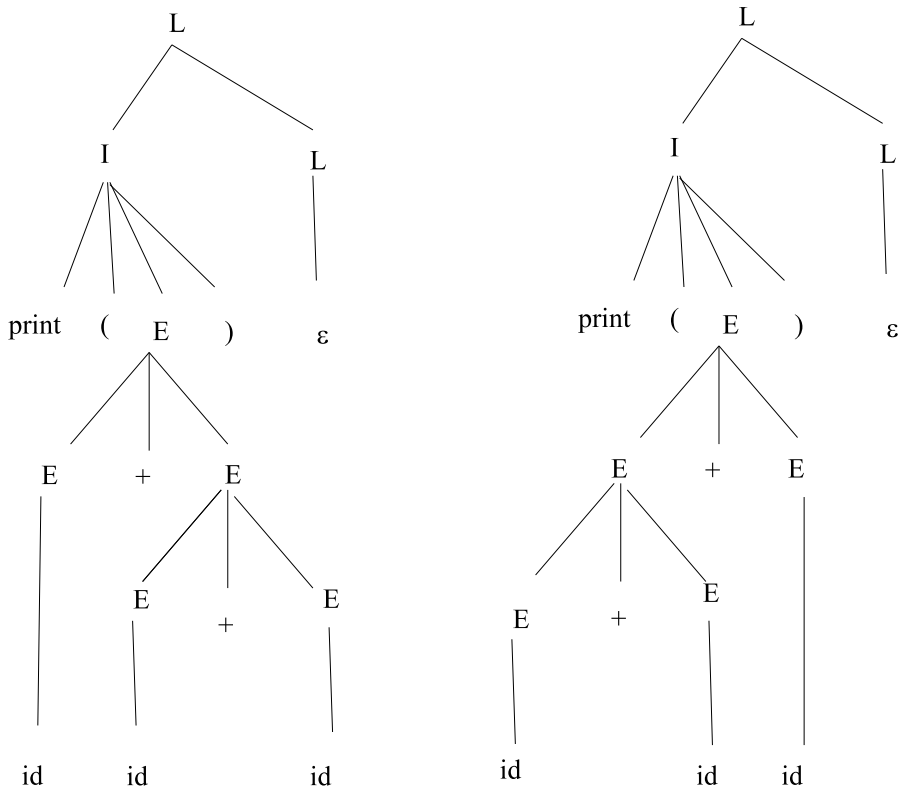


FIGURE 7.7 – arbres de dérivation associés à la chaine *print(id+id+id)*

La grammaire G est donc ambigüe.

2. Éliminez la récursivité gauche dans G.
   L'élimination de la récursivité gauche directe en E se fait en appliquant les règles de transformation suivantes :

$$A \rightarrow A\alpha_1 \mid ... \mid A\alpha_n \mid \beta_1 \mid ... \mid \beta_m \text{ avec } A \in \mathbb{N} \text{ et } \beta_i, \alpha_i \in (T \cup N)^* \Leftrightarrow$$

$$P' : \begin{cases} A \rightarrow \beta_1 A' \mid ... \mid \beta_m A' \mid \beta_1 \mid ... \mid \beta_m \\ A' \rightarrow \alpha_1 A' \mid ... \mid \alpha_n A' \mid \alpha_1 \mid ... \mid \alpha_n \end{cases}$$

ou encore

$$P' : \begin{cases} A \rightarrow \beta_1 A' \mid ... \mid \beta_m A' \\ A' \rightarrow \alpha_1 A' \mid ... \mid \alpha_n A' \mid \epsilon \end{cases}$$

Les règles de productions relatives au non terminal E deviennent :

$$\begin{cases} E \rightarrow idE' \mid numE' \\ E' \rightarrow +EE' \mid -EE' \mid \epsilon \end{cases}$$

Ainsi, la grammaire non récursive gauche G' équivalente à G est définie par <T,$\mathbb{N}'$,L,P'> tels que :
T={id = print ( ) + - num int float}
$\mathbb{N}'$={L,I,E,E',T} et

$$P' : \begin{cases} L \rightarrow IL \mid \epsilon \\ I \rightarrow T\ id \mid id = E \mid print(E) \\ E \rightarrow idE' \mid numE' \\ E' \rightarrow +EE' \mid -EE' \mid \epsilon \\ T \rightarrow int \mid float \end{cases}$$

3. Factorisez éventuellement la grammaire obtenue en 2.
La grammaire G' est factorisée. La Grammaire G' augmentée est alors définie par <T,$\mathbb{N}'$,L,P'> tels que :
T={id = print ( ) + - num int float}
$\mathbb{N}'$={L,I,E,E',T} et

$$P' : \begin{cases} Z \rightarrow L\# \\ L \rightarrow IL\ (1) \mid \epsilon\ (2) \\ I \rightarrow T\ id\ (3) \mid id = E\ (4) \mid print(E)\ (5) \\ E \rightarrow idE'\ (6) \mid numE'\ (7) \\ E' \rightarrow +EE'\ (8) \mid -EE'\ (9) \mid \epsilon\ (10) \\ T \rightarrow int\ (11) \mid float\ (12) \end{cases}$$

4. Calculez les ensembles Début et Suivant de la grammaire obtenue en 3.
   Les ensembles des débuts et des suivants associés à la grammaire G' sont donnés par le tableau suivant :

	Débuts	Suivants
L	id print int float ϵ	#
I	id print int float	id print int float #
E	id num	) + - id print int float #
E'	+ - ϵ	) + - id print int float #
T	int float	id

5. Construisez la table d'analyse LL(1) de la grammaire obtenue en 3.

	id	=	print	(	)	num	+	-	int	float	#
L	R1		R1						R1	R1	R2
I	R4		R5						R3	R3	
E	R6					R7					
E'	R10		R10		R10		R8 / R10	R9 / R10	R10	R10	R10
T									R11	R12	

6. La grammaire obtenue en 3 est-elle LL(1) ? Justifiez.
   G' n'est pas LL(1) car la table d'analyse LL(1) est multi-définie.
   Les cas de multi-définitions peuvent être détectés sans construire la table d'analyse comme suit :

   - Pour les règles $L \rightarrow IL \mid \epsilon$
     débuts(IL) $\cap$ suivants(L)=$\emptyset$
   - Pour les règles $I \rightarrow T\ id \mid id = E \mid print(E)$
     débuts(T id) $\cap$ débuts(id=E)=$\emptyset$
     débuts(T id) $\cap$ débuts(print(E))=$\emptyset$
     débuts(id=E) $\cap$ débuts(print(E))=$\emptyset$
   - Pour les règles $E \rightarrow idE' \mid numE'$
     débuts(id E') $\cap$ débuts(num E')=$\emptyset$
   - Pour les règles $E' \rightarrow +EE' \mid -EE' \mid \epsilon$
     débuts(+EE') $\cap$ débuts(-EE')=$\emptyset$
     débuts(+EE') $\cap$ suivants(E')=$\{+\} \neq \emptyset$
     débuts(-EE') $\cap$ suivants(E')= $\{-\} \neq \emptyset$

- Pour les règles $T \rightarrow int \mid float$
  débuts(int) $\cap$ débuts(float)=$\emptyset$

Remarque 34 *Les cas de multi-définitions peuvent être résolus en attribuant des priorité aux règles de production de la grammaire G. Par exemple, si l'ordre d'apparition des règles de production dans la grammaire induit une priorité entre les règles alors la règle R8 serait plus prioritaire que la règle R10 et la règle R9 serait aussi plus prioritaire que la règle R10. Ainsi, M[E',+] := R8 et M[E',-] :=R9. La table d'analyse LL(1) serait alors mono-définie.*

Exercice 3 :

Soit *G* la grammaire définie par <T,N,S,P> où T={a b c d} ; N={S,T,U} ; et P :

$$\begin{cases} Z \rightarrow S\# \\ S \rightarrow aTS \ (1) \\ T \rightarrow bUT \ (2) \mid \epsilon \ (3) \\ U \rightarrow cU \ (4) \mid dSc \ (5) \mid \epsilon \ (6) \end{cases}$$

- Vérification des conditions LL(1) :
  1. La grammaire G est non récursive gauche directe.
  2. La grammaire G est factorisée.
  3. La table d'analyse LL(1) est-elle mono-définie.
     Les débuts et les suivants relatifs aux non terminaux de la grammaire G sont donnés par la table suivante :

TABLE 7.1 – Les débuts et les suivants associés aux non terminaux de la grammaire G

	Débuts	Suivants
S	a	c, #
T	b, ϵ	a
U	c, d, ϵ	a, b

  4. La table d'analyse de la grammaire G est mono-définie. En effet,

     — pour S, il existe un seul MDP,

— pour T, Débuts(bUT) ∩ Suivants(T) =∅,
— pour U, Débuts(cU) ∩ Débuts(dSc)= ∅
Débuts(cU)∩ Suivants(U)=∅
Débuts(dSc)∩ Suivants(U)=∅.

La table d'analyse LL(1) de la grammaire G est comme suit :

	a	b	c	d	#
S	R1				
T	R3	R2			
U	R6	R6	R4	R5	

Ainsi, la grammaire G vérifie toutes les conditions LL(1).

- définition des procédures :

Programme principal Z() : Z → S #

```
début
    Lire(chaine);
    tc := 1^ere entité de la chaîne;
    si (tc ∈ Débuts(S)) alors
        S();
        si (tc = '#') alors
            Écrire('Chaine acceptée');
        sinon
            Écrire("Erreur :# expected"));
    sinon
        Écrire("Erreur : a expected"));
fin
```

Procédure S() S → aTS

```
début
    tc := ts;
    si (tc ∈ Débuts(T)) alors
        T();
        si (tc ∈ Débuts(S)) alors
            S();
        sinon
            Écrire("Erreur :a expected");
    sinon si (tc ∉ Suivants(T)) alors
        Écrire("Erreur :a expected");
fin
```

Procédure T() : T → bUT | ϵ

```
début
    tc := ts ;
    si (tc ∈ Débuts(U)-{ϵ }) alors
        U() ;
        si (tc ∈ Débuts(T)-{ϵ }) alors
            T() ;
        sinon si (tc ∉ Suivants(T)) alors
            Écrire("Erreur :a expected") ;
    sinon si (tc ∉ Suivants(U)) alors
        Écrire("Erreur : a ou b expected") ;
fin
```

Procédure U() : U → cU | dSc | ϵ

```
début
    si (tc ='c') alors
        tc := ts ;
        si (tc ∈ Débuts(U)- {ϵ }) alors
            U() ;
        sinon
            si (tc ∉ Suivants(U)) alors
                Écrire("Erreur :a ou b expected") ;
    sinon
        si (tc ='d') alors
            tc := ts ;
            si (tc ∈ Débuts(S)) alors
                S() ;
                si (tc ='c') alors
                    tc := ts ;
                sinon
                    Écrire("Erreur :c expected") ;
            sinon
                Écrire("Erreur :a expected") ;
        sinon si (tc ∉ Suivants(U)) alors
            Écrire("Erreur :a ou b expected") ;
fin
```

II-2 Méthodes Ascendantes

Exercice 4 :

Soit la grammaire G définie par <T,ℕ,S,P> tels que :
T={ a b}
ℕ={S,A,B} et

$$P : \begin{cases} S \rightarrow BA \\ A \rightarrow aB \mid \epsilon \\ B \rightarrow Bb \mid \epsilon \end{cases}$$

1. G est-elle LR(1) ?

— Soit la grammaire augmentée correspondante à G suivante :

$$\begin{cases} Z \rightarrow S\# \\ S \rightarrow BA \,①\\ A \rightarrow aB \,② \mid \epsilon \,③ \\ B \rightarrow Bb \,④ \mid \epsilon \,⑤ \end{cases}$$

— Les ensembles débuts et suivants associés aux non terminaux de la grammaire G sont donnés par la table suivante :

	Débuts	Suivants
S	a, b ϵ	#
A	a, ϵ	#
B	b, ϵ	a, b, #

— Calcul des items LR(1) :
I_0 = { [Z → .S,#] [S → .BA,#] [B → .Bb,a |#] [B → .,a|#] }
I_1 = GOTO(I_0, S) = { [Z → S.,#] }
I_2 = GOTO(I_0, B) = { [S → B.A,#] [B → B.b,a|#] [A → .aB,#] [A → .,#] }
I_3 = GOTO(I_2, A) = { [S → BA.,#] }
I_4 = GOTO(I_2, b) = { [B → Bb.,a|#] }
I_5 = GOTO(I_2, a) = { [A → a.B,#], [B → .Bb,#] [B → .,#] }
I_6 = GOTO(I_5, B) = { [A → aB.,#] [B → B.b,#] }
I_7 = GOTO(I_6, b) = { [B → Bb.,#] }
La collection des items C ={I_0,I_1,I_2,I_3,I_4,I_5,I_6,I_7}

— Construction de la table d'analyse LR(1) :

États	a	b	#	S	A	B
0	R5		R5	1		2
1			Accept			
2	D,5	D,4	R3		3	
3			R1			
4	R4	R4	R4			
5			R5		6	
6		D,7	R2			
7			R4			

— La table d'analyse LR(1) est mono-définie $\Longrightarrow$ G est LR(1).

2. G est-elle LALR(1) ?

— L'item LR(1) I_4 et l'item LR(1) I_7 peuvent être regroupés car ils possèdent le même cœur :
I_{47} = {[B → Bb.,a | #]}
La collection C' ={I_0,I_1,I_2,I_3,I_{47},I_5,I_6}

— Construction de la table d'analyse LALR(1) :

	a	b	#	S	A	B
0	R5		R5	1		2
1			Accept			
2	D,5	D,47	R3		3	
3			R1			
47	R4	R4	R4			
5			R5		6	
6		D,47	R2			

La table d'analyse LALR(1) est mono-définie $\Longrightarrow$ G est LALR(1).

3. G es-elle SLR(1) ?

— Calcul des items SLR(1) :

I_0 = { [Z → .S] [S → .BA] [B → .Bb] [B → .] }
I_1 = GOTO(I_0,S) = { [Z → S.] }
I_2 = GOTO(I_0,B) = { [S → B.A] [B → B.b] [A → .aB] [A → .] }
I_3 = GOTO(I_2, A) = { [S → BA.] }
I_4 = GOTO(I_2, b) = { [B → Bb.] }
I_5 = GOTO(I_2, a) = { [A → a.B] [B → .Bb] [B → .] }
I_6 = GOTO(I_5, B) = { [A → aB.] [B → B.b] }
I_4 = GOTO(I_6, b)

— La collection des items C ={I_0,I_1,I_2,I_3,I_4,I_5,I_6}

— Construction de la table d'analyse SLR(1) :

	a	b	#	S	A	B
0	R5	R5	R5	1		2
1			Accept			
2	D,5	D,4	R3		3	
3	D,5	D,4	R1			
4	R4	R4	R4			
5	R5	R5	R5		6	
6		D,4	R2			

— La table d'analyse SLR(1) est mono-définie $\Longrightarrow$ G est SLR(1).

Exercice 5 :

Soit la grammaire G définie par <T,ℕ,X,P> tels que :
T={ a b c d}
ℕ={X,M} et

$$P:\begin{cases} X \rightarrow Ma \mid bMc \mid dc \mid bda \\ M \rightarrow d \end{cases}$$

1. G est-elle SLR(1) ? Justifiez
   Soit la grammaire augmentée correspondante à G suivante :

$$\begin{cases} Z \rightarrow X\# \\ X \rightarrow Ma\,\textcircled{1} \mid bMc\,\textcircled{2} \mid dc\,\textcircled{4} \mid bda\,\textcircled{4} \\ M \rightarrow d\,\textcircled{5} \end{cases}$$

   Les ensembles débuts et suivants associés aux non terminaux de la grammaire G sont donnés par la table suivante :

	Débuts	Suivants
X	d b	#
M	d	a c

1. G est-elle SLR(1) ?

— Calcul des items LR(0) :

I_0 = { [Z → .X] [X → .Ma] [X → .bMc] [X → .dc] [X → .bda] [M → .d] }
I_1 = GOTO(I_0, X) = { [Z → X.] }
I_2 = GOTO(I_0, M){ [X → M.a] }
I_3 = GOTO(I_0, b) = { [X → b.Mc] [X → b.da] [M → .d] }
I_4 = GOTO(I_0, d) = { [X → d.c] [M → d.] }
I_5 = GOTO(I_2, a) = { [X → Ma.] }
I_6 = GOTO(I_3, M) = { [X → bM.c] }
I_7 = GOTO(I_3, d) = { [X → bd.a] [M → d.] }
I_8 = GOTO(I_4, c) = { [X → dc.] }
I_9 = GOTO(I_6, c) = { [X → bMc.] }
I_{10} = GOTO(I_7, a) = { [X → bda.] }

— La collection des items C ={I_0,I_1,I_2,I_3,I_4,I_5,I_6,I_7,I_8,I_9,I_{10}}.

— Construction de la table d'analyse SLR(1) :

	a	b	c	d	#	X	M
0		D,3		D,4		1	2
1					Accept		
2	D,5						
3				D,7			6
4	R5		R5 / D,8				
5					R1		
6			D,9				
7	R5 / D,10		R5				
8					R3		
9					R2		
10					R4		

— La table d'analyse SLR(1) $\mathbb{M}$ est multi-définie $\Longrightarrow$ G n'est pas SLR(1).

Remarque 35 *Les deux cas de multi-définitions peuvent être détectés directement à partir des items I_4 et I_7. En effet :*

(a) $[X \rightarrow d.c] \in I_4 \Longrightarrow \mathbb{M}[I_4,c] := D,8$

(b) $[M \rightarrow d.] \in I_4 \Longrightarrow \mathbb{M}[I_4,c] := R_5$

(c) $[X \rightarrow bd.a] \in I_7 \Longrightarrow \mathbb{M}[I_7,a] := D,10$

(d) $[M \rightarrow d.] \in I_7 \Longrightarrow \mathbb{M}[I_7,a] := R_5$

Remarque 36 *Lors d'une analyse SLR(1), les réductions du type* $A \rightarrow \alpha$. *s'effectuent avec les suivants du MGP.*

2. G est-elle LR(1) ? Justifiez.

— Calcul des items LR(1 :

I_0 = { [Z → .X,#] [X → .Ma,#] [X → .bMc,#] [X → .dc,#] [X → .bda,#] [M → .d,a] }
I_1 = GOTO(I_0, X) = { [Z → X.,#] }
I_2 = GOTO(I_0, M){ [X → M.a,#] }
I_3 = GOTO(I_0, b) = { [X → b.Mc,#] [X → b.da,#] [M → .d,c] }
I_4 = GOTO(I_0, d) = { [X → d.c,#] [M → d.,a] }
I_5 = GOTO(I_2, a) = { [X → Ma.,#] }
I_6 = GOTO(I_3, M) = { [X → bM.c,#] }
I_7 = GOTO(I_3, d) = { [X → bd.a,#] [M → d.,c] }
I_8 = GOTO(I_4, c) = { [X → dc.,#] }
I_9 = GOTO(I_6, c) = { [X → bMc.,#] }
I_{10} = GOTO(I_7, a) = { [X → bda.,#] }

— La collection des items C ={I_0,I_1,I_2,I_3,I_4,I_5,I_6,I_7,I_8,I_9,I_{10}}.

— Construction de la table d'analyse LR(1) :

	a	b	c	d	#	X	M
0		D,3		D,4		1	2
1					Accept		
2	D,5						
3				D,7			6
4	R5		D,8				
5					R1		
6			D,9				
7	D,10		R5				
8					R3		
9					R2		
10					R4		

— La table d'analyse LR(1) mono-définie $\Longrightarrow$ G est LR(1).

Remarque 37 *Dans le cas d'une analyse LR(), les réductions du type A→ α.,ω s'effectuent avec les terminaux ω.*

Remarque 38 *Les deux cas de multi-définitions présents dans la tables d'analyse SLR() ont été résolus dans le cas de l'analyse LR(1) à l'aide de l'entité supplémentaire représentée par ω prise en compte en plus de l'entité en cours du traitement.*

3. G est-elle LALR(1) ? Justifiez.
   Il n' y a pas de regroupements des ensembles des items LR(1) ce qui implique que la table d'analyse LALR(1) est identique à celle de LR(1).

Exercice 6 :

Soit la grammaire G définie par <T,ℕ,E,P> tels que :
T={ ? @ op nb}
ℕ={E} et

$$P : \left\{ E \rightarrow ?E\ \textcircled{1} \mid E@E\ \textcircled{2} \mid EopE\ \textcircled{3} \mid nb\ \textcircled{4} \right.$$

a. G est-elle ambiguë ?
   La grammaire G est ambigüe. A titre d'exemple, à la chaîne 'nb op nb op nb', deux arbres syntaxiques lui sont associés comme indiqué dans la Figure 7.8.

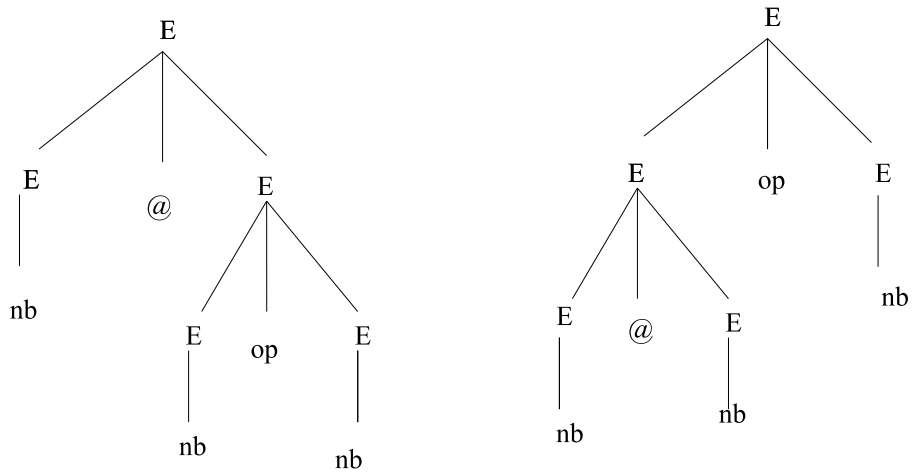


FIGURE 7.8 – arbres syntaxiques associés à la chaine *nb @ nb op nb*

La grammaire G est donc ambigüe $\Longrightarrow$ la grammaire G n'est pas SLR(1).

b. Construisez la table d'analyse SLR(1).

— Les ensembles débuts et suivants associés aux non terminaux de la grammaire G sont donnés par la table suivante :

	Débuts	Suivants
E	?, nb	@, op, #

— Calcul des items LR(0) :
I_0 = { [Z → .E] [E → . ?E] [E → .E @ E] [E → .E op E] [E → .nb]}
I_1= GOTO(I_0, E) = {[Z → E.][E → E .@ E][E → E .op E]}
I_2 = GOTO(I_0, ?) ={[E → ?.E] [E → . ?E] [E → .E @ E] [E → .E op E] [E → .nb]}
I_3= GOTO(I_0, nb) = {[E → nb.]}
I_4= GOTO(I_1, @) = {[E → E@.E][E → . ?E] [E → .E @ E] [E → .E op E] [E → .nb]}
I_5= GOTO(I_1, op) = {[E → Eop.E][E → . ?E] [E → .E @ E] [E → .E op E] [E → .nb]}
I_6= GOTO(I_2, E) = {[E → ?E.][E → E .@ E][E → E .op E]}
I_2 = GOTO(I_2, ?)
I_3= GOTO(I_2, nb)
I_7= GOTO(I_4, E) = {[E → E@E.][E → E .@ E][E → E .op E]}
I_2 = GOTO(I_4, ?)
I_3= GOTO(I_4, nb)
I_8= GOTO(I_5, E) = {[E → EopE.][E → E .@ E][E → E .op E]}
I_2 = GOTO(I_5, ?)
I_3= GOTO(I_5, nb)
I_4 = GOTO(I_6, @)
I_5= GOTO(I_6, op)
I_4 = GOTO(I_7, @)
I_5= GOTO(I_7, op)
I_4 = GOTO(I_8, @)
I_5= GOTO(I_8, op)
Ainsi, La collection des items C ={I_0,I_1,I_2,I_3,I_4,I_5,I_6,I_7,I_8}.

— Construction de la table d'analyse SLR(1) :

	?	@	op	nb	#	E
0	D,2			D,3		1
1		D,4	D,5		Accept	
2	D,2			D,3		6
3		R4	R4		R4	
4	D,2			D,3		7
5	D,2			D,3		8
6		R1 D,4	R1 D,5		R1	
7		R2 D,4	R2 D,5		R2	
8		R3 D,4	R3 D,5		R3	

La table d'analyse SLR(1) $\mathbb{M}$ est multi-définie $\Longrightarrow$ G n'est pas SLR(1).

Remarque 39 *Les six cas de multi-définitions peuvent être détectés directement à partir des items I_6, I_7 et I_8. En effet :*

(a) $[E \rightarrow ?E.] \in I_6 \Longrightarrow \mathbb{M}[I_6, @] := R_1$

(b) $[E \rightarrow E.@E] \in I_6 \Longrightarrow \mathbb{M}[I_6, @] := D,4$

(c) $[E \rightarrow ?E.] \in I_6 \Longrightarrow \mathbb{M}[I_6, op] := R_1$

(d) $[E \rightarrow E.opE] \in I_6 \Longrightarrow \mathbb{M}[I_6, op] := D,5$

(e) $[E \rightarrow E@E.] \in I_7 \Longrightarrow \mathbb{M}[I_7, @] := R_2$

(f) $[E \rightarrow E.@E] \in I_7 \Longrightarrow \mathbb{M}[I_7, @] := D,4$

(g) $[E \rightarrow E@E.] \in I_7 \Longrightarrow \mathbb{M}[I_7, op] := R_2$

(h) $[E \rightarrow E.opE] \in I_7 \Longrightarrow \mathbb{M}[I_7, op] := D,5$

(i) $[E \rightarrow EopE.] \in I_8 \Longrightarrow \mathbb{M}[I_8, @] := R_3$

(j) $[E \rightarrow E.@E] \in I_8 \Longrightarrow \mathbb{M}[I_8, @] := D,4$

(k) $[E \rightarrow EopE.] \in I_8 \Longrightarrow \mathbb{M}[I_8, op] := R_3$

(l) $[E \rightarrow E.opE] \in I_8 \Longrightarrow \mathbb{M}[I_8, op] := D,5$

c. Est-il possible d'éliminer les multi-définitions en adoptant les conventions suivantes ?

— ? plus prioritaire que @,

— @ plus prioritaire que op,

— l'associativité est de droite à gauche.

Ces conventions permettent de résoudre les cas de conflits induits par la grammaire G comme suit :

- cas de M[I_6,@]=D,4 et M[I_6,@]=R1
  Il correspond au conflit ?nb @ nb impliquant l'opérateur ? suivi de l'opérateur @. A la lecture du second opérateur @, deux actions sont envisageables :
  — soit avancer afin d'évaluer d'abord l'opérateur @ (D,4),
  — ou encore évaluer en premier l'opérateur ? en effectuant la réduction (R1).

  Comme l'opérateur ? est plus prioritaire que l'opération @, l'action de réduction R1 sera sélectionnée.
- cas de M[I_6,op]=D,5 et M[I_6,op]=R1
  Il correspond au conflit ?nb op nb impliquant l'opérateur ? suivi de l'opérateur op. A la lecture du second opérateur op, deux actions sont envisageables :
  — soit avancer afin d'évaluer d'abord l'opérateur op (D,5),
  — ou encore évaluer en premier l'opérateur ? en effectuant la réduction (R1).

  Comme l'opérateur ? est plus prioritaire que l'opération op, l'action de réduction R1 sera sélectionnée.
- cas de M[I_7,@]=D,4 et M[I_7,@]=R2
  Il correspond au conflit nb @ nb @ nb impliquant l'opérateur @ suivi de l'opérateur @. A la lecture du second opérateur @, deux actions sont envisageables :
  — soit avancer afin d'évaluer d'abord l'expression impliquant le second opérateur @ (D,4),
  — ou encore évaluer en premier la première expression en effectuant la réduction (R2).

  Comme l'opérateur @ est associatif de droite à gauche, l'action de décalage D,4 sera sélectionnée.
- cas de M[I_7,op]=D,5 et M[I_7,op]=R2
  Il correspond au conflit nb @ nb op nb impliquant l'opérateur @ suivi de l'opérateur op. A la lecture du second opérateur op, deux actions sont envisageables :
  — soit avancer afin d'évaluer d'abord l'expression impliquant l'opérateur op (D,5),
  — ou encore évaluer en premier l'expression impliquant l'opérateur @ en effectuant la réduction (R2).

Comme l'opérateur @ est plus prioritaire que l'opération op, l'action de réduction R2 sera sélectionnée.

- cas de M[I_8,@]=D,4 et M[I_8,@]=R3
  Il correspond au conflit nb op nb @ nb impliquant l'opérateur op suivi de l'opérateur @. A la lecture du second opérateur @, deux actions sont envisageables :
  — soit avancer afin d'évaluer d'abord l'expression impliquant le second opérateur @ (D,4),
  — ou encore évaluer en premier la première expression en effectuant la réduction (R3).
  Comme l'opérateur @ est plus prioritaire que l'opération op, l'action de décalage (D,4) sera sélectionnée.
- cas de M[I_8,op]=D,5 et M[I_8,op]=R3
  Il correspond au conflit nb op nb op nb impliquant l'opérateur op suivi de l'opérateur op. A la lecture du second opérateur op, deux actions sont envisageables :
  — soit avancer afin d'évaluer d'abord l'expression impliquant le second opérateur op (D,5),
  — ou encore évaluer en premier l'expression impliquant le premier opérateur op en effectuant la réduction (R3).
  Comme l'opérateur op est associatif de droite à gauche, l'action de décalage D,5 sera sélectionnée.

La table d'analyse SLR(1) mono-définie qui en résulte après la phase de résolution de conflits est la suivante :

	?	@	op	nb	#	E
0	D,2			D,3		1
1		D,4	D,5		Accept	
2	D,2			D,3		6
3		R4	R4		R4	
4	D,2			D,3		7
5	D,2			D,3		8
6		R1	R1		R1	
7		D,4	R2		R2	
8		D,4	D,5		R3	

Soit à analyser la chaine **a op b @ a#**

Pile	tc	Chaine	Action
#⓪	a	a op b @ a#	D,3
#⓪$\underbrace{a③}$	op	op b @ a#	R4 : E → nb
#⓪E①	op	op b @ a#	D,5
#⓪E①op⑤	b	b @ a#	D,3
#⓪E①op⑤ $\underbrace{b③}$	@	@ a#	R4 : E → nb
#⓪E①op⑤E⑧	@	@a#	D,4
#⓪E①op⑤E⑧@④	a	a#	D,3
#⓪E①op⑤E⑧@④$\underbrace{a③}$	#	#	R4 : E → nb
#⓪E①op⑤$\underbrace{E⑧@④E⑦}$	#	#	R2 : E → E@E
#⓪$\underbrace{E①op⑤E⑧}$	#	#	R3 : E → EopE
#⓪E①	#	#	ACCEPT

Exercice 7 :

Soit la grammaire G définie par <T,ℕ,E,P> tels que T={id sep par entier ser ( )} ; ℕ={S D C} et P :

$$\begin{cases} S \rightarrow D\ sep\ C \\ D \rightarrow id\ entier \mid D\ id\ entier \\ C \rightarrow id \mid C\ par\ C \mid C\ ser\ C \mid (C) \end{cases}$$

a. G est-elle ambiguë ?
La grammaire G est ambigüe. A titre d'exemple, à la chaine "id entier sep id par id par id" deux arbres syntaxiques lui sont associés représentés par la Figure 7.9.

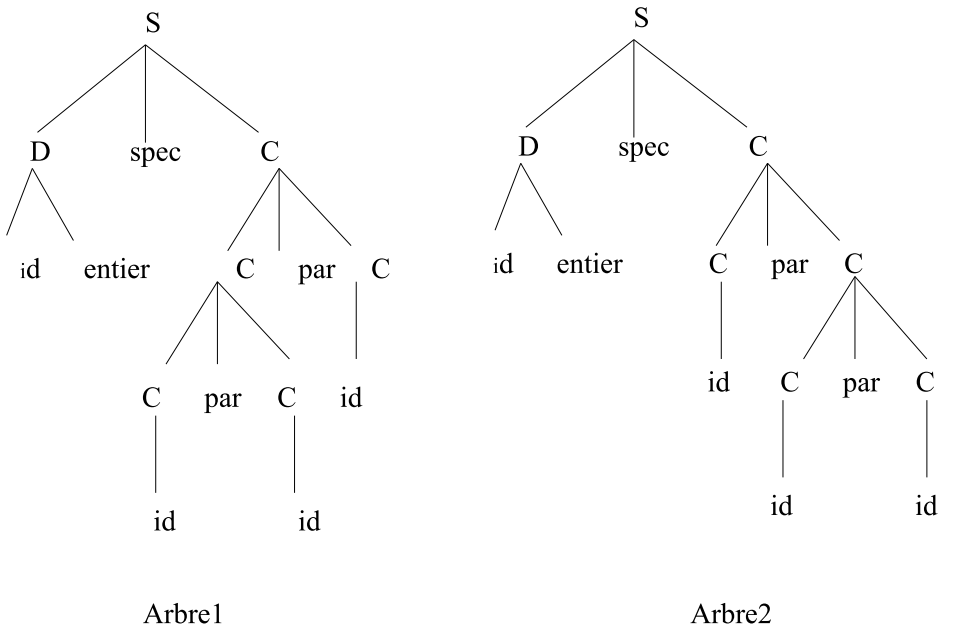


FIGURE 7.9 – arbres syntaxiques associés à la chaine *id entier sep id par id par id*

b. G n'est pas SLR(1) car la grammaire est ambigüe.

Soit la grammaire augmentée correspondante à G suivante :

$$\begin{cases} Z \rightarrow S\# \\ S \rightarrow D\ sep\ C \\ D \rightarrow id\ entier \mid D\ id\ entier \\ C \rightarrow id \mid C\ par\ C \mid C\ ser\ C \mid (C) \end{cases}$$

Les ensembles débuts et suivants associés aux non terminaux de la grammaire G sont donnés par la table suivante :

	Débuts	Suivants
S	id	#
D	id	sep id
C	id (	par sep) #

— Calcul des items LR(0) :
I_0 = { [Z → .S] [S → .D sep C] [D → .id entier] [D → .D id entier]}
I_1= GOTO(I_0, S) = {[Z → S.]}
I_2 = GOTO(I_0, D){[S → D.sep C] [D → D. id entier]}
I_3 = GOTO(I_0, id) = {[D → id.entier]}
I_4 = GOTO(I_2, sep) ={[S → D sep.C] [C → .id] [C → .C par C] [C → .C ser C] [C → .(C)]}
I_5 = GOTO(I_2, id) = {[D → D id.entier]}
I_6 = GOTO(I_3, entier) = {[D → id entier.]}
I_7 = GOTO(I_4, C) = {[D → D sep C.] [C → C.par C] [C → C.ser C]}
I_8 = GOTO(I_4, id) = {[C → id.]}
I_9 = GOTO(I_4, () = {[C → (.C)] [C → .id] [C → .C par C] [C → .C ser C] [C → .(C)]}
I_{10} = GOTO(I_5, entier) = {[D → D id entier.]}
I_{11} = GOTO(I_7, par) = {[C → (C par.C)] [C → .id] [C → .C par C] [C → .C ser C] [C → .(C)]}
I_{12} = GOTO(I_7, ser) = {[C → (C ser.C)] [C → .id] [C → .C par C] [C → .C ser C] [C → .(C)]}
I_{13} =GOTO(I_9, C) = {[C → (C.)] [C → C.par C] [C → C.ser C]}
I_8 = GOTO(I_9, id)
I_9 = GOTO(I_9, ()
I_{14} = GOTO(I_{11}, C) = {[C → (C par C.)] [C → C.par C] [C → C.ser C]}
I_8 = GOTO(I_{11}, id)
I_9 = GOTO(I_{11}, ()
I_{15} = GOTO(I_{12}, C) = {[C → (C ser C.)] [C → C.par C] [C → C.ser C]}
I_8 = GOTO(I_{12}, id)
I_9 = GOTO(I_{12}, ()
I_{16} = GOTO(I_{13},))= {[C → (C).]}
I_{11} = GOTO(I_{13}, par)
I_{12} = GOTO(I_{13}, ser)
I_{11} = GOTO(I_{14}, par)
I_{12} = GOTO(I_{14}, ser)
I_{11} = GOTO(I_{15}, par)
I_{12} = GOTO(I_{15}, ser)

— La collection des items C ={I_0,I_1,I_2,I_3,I_4,I_5,I_6,I_7,I_8,I_9,I_{10},I_{11},I_{12},I_{13},I_{14},I_{15},I_{16}}.

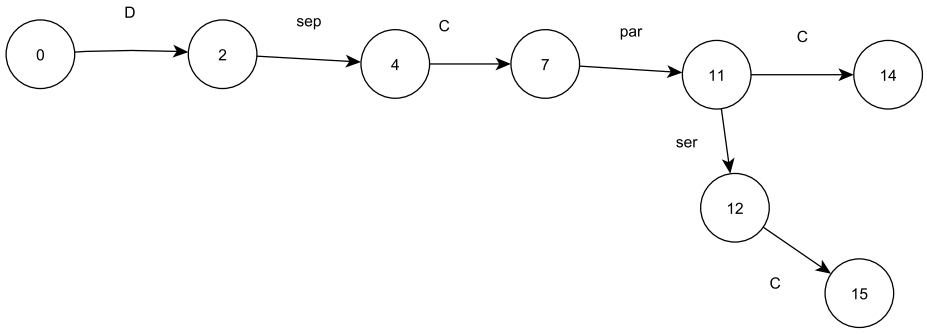
— Construction de la table d'analyse SLR(1) :

	sep	id	entier	par	ser	(	)	#	S	D	C
0		D,3							1	2	
1								Accept			
2	D,4	D,5									
3			D,6								
4		D,8				D,9					7
5			D,10								
6	R2	R2									
7				D,11	D,12			R1			
8				R4	R4		R4	R4			
9		D,8				D,9					13
10	R3	R3									
11		D,8				D,9					14
12		D,8				D,9					15
13				D,11	D,12		D,16				
14				R5 D,11	R5 D,12		R5	R5			
15				R6 D,11	R6 D,12		R6	R6			
16				R7	R7		R7	R7			

— La table d'analyse SLR(1) $\mathbb{M}$ est multi-définie $\Longrightarrow$ G n'est pas SLR(1).
Les cas de multi-définitions peuvent être éliminés en adoptant les conventions suivantes :

(a) associativité de gauche à droite pour les opérateurs par et ser

(b) les priorités dans l'ordre décroissant sont : ( ), ser, par.



— à l'état 14, le contenu de la pile est le suivant : D sep C par C

début
  si *(tc = 'par')* **alors**
    l'action de réduction est retenue car l'opérateur **par** est associatif de gauche à droite ;
  si *(tc = 'ser')* **alors**
    l'action de décalage est retenue car l'opérateur **ser** est plus prioritaire que l'opérateur **par**
fin

— à l'état 15, le contenu de la pile est le suivant : D sep C ser C

début
  si *(tc = 'par')* **alors**
    l'action de réduction est retenue car l'opérateur **ser** est plus prioritaire que l'opérateur **par**
  si *(tc = 'ser')* **alors**
    l'action de réduction est retenue car l'opérateur **ser** est associatif de gauche à droite ;
fin

— Après résolution des 4 cas de conflits, la table d'analyse SLR(1) devient mono-définie comme suit :

	sep	id	entier	par	ser	(	)	#	S	D	C
0		D,3							1	2	
1								Accept			
2	D,4	D,5									
3			D,6								
4		D,8				D,9					7
5			D,10								
6	R2	R2									
7				D,11	D,12			R1			
8				R4	R4		R4	R4			
9		D,8				D,9					13
10	R3	R3									
11		D,8				D,9					14
12		D,8				D,9					15
13				D,11	D,12		D,16				
14				R5	D,12		R5	R5			
15				R6	R6		R6	R6			
16				R7	R7		R7	R7			

Exercice 1 :

Les entités lexicales relatives à la description d'une facture sont :

1. les mots clés : FACTURE et LABEL,
2. le numéro de la facture composé de deux à quatre lettres représentant les deux premières lettres du mois suivies de trois chiffres représentant le numéro de la facture du mois correspondant,
3. un libellé qui est représenté par une suite de lettres,
4. un nombre entier représentant la quantité,
5. le prix représenté par un réel.

La reconnaissance de chaque entité lexicale débute par l'état initial. Afin de rendre l'automate déterministe, il faut :

— pour les mois qui commencent par les mêmes caractères (tels que juin et juillet), il faut regrouper les transitions jusqu'à la lettre qui les différencie (jui pour les deux mois en question)

— pour différencier le label et le numéro de la facture, il faut opérer de la même manière pour différencier les mots clés des identificateurs. L'état 24 permet de poursuivre la reconnaissances des labels avec la reconnaissances des lettres.

L'automate déterministe permettant la reconnaissances des différentes entités citées est représenté par la Figure suivante :

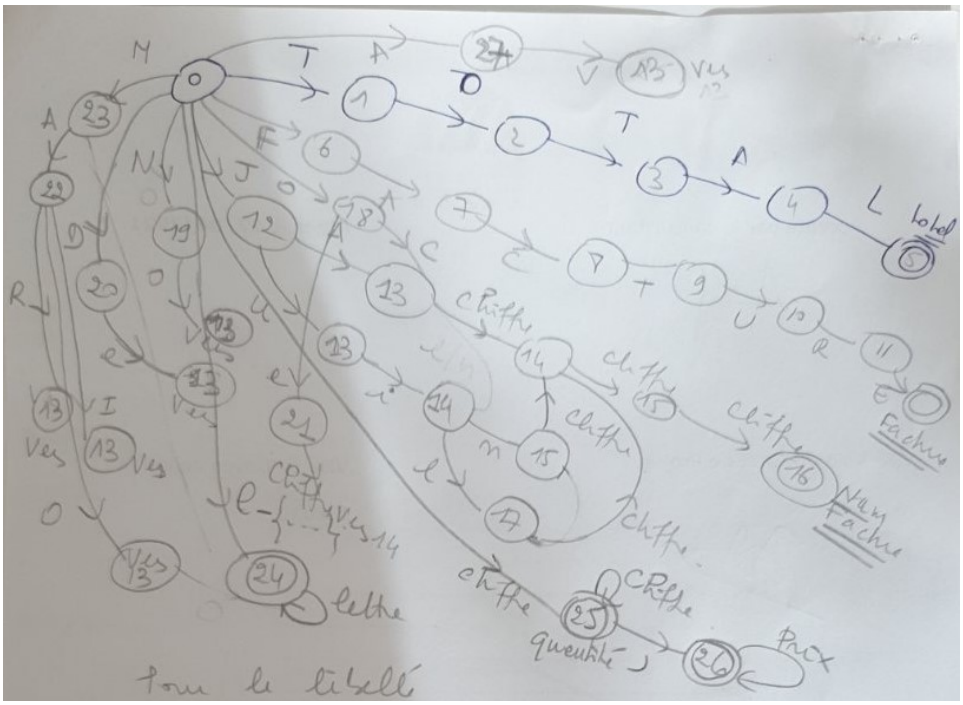


FIGURE 7.10 – arbres syntaxiques associés à la chaine *Automate déterministe reconnaissant les entités*

Exercice 2 :

Soit la grammaire G définie par <T,ℕ,R,P> tels que T={join, union, select, (,), id} ; ℕ={R} et P :

$$\begin{cases} R \rightarrow R\ join\ R \mid R\ union\ R \mid select(R) \mid id \end{cases}$$

a. G est-elle ambiguë ?
La grammaire G est ambigüe. A titre d'exemple, à la chaine "id join id join id " deux arbres syntaxiques lui sont associés représentés par la Figure 7.11.

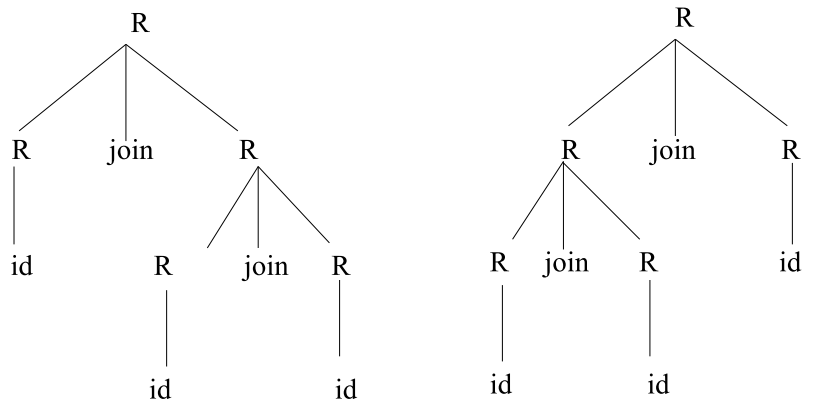


FIGURE 7.11 – arbres syntaxiques associés à la chaine *id join id join id*

b. G est-elle SLR(1) ?
La grammaire est ambigüe $\Longrightarrow$ la grammaire G n'est pas SLR(1).

c. Construction de la table d'analyse SLR(1).

— Soit la grammaire augmentée correspondante à G suivante :

$$\begin{cases} Z \rightarrow R\# \\ R \rightarrow R\ join\ R \mid R\ union\ R \mid select(R) \mid id \end{cases}$$

— Les ensembles débuts et suivants associés aux non terminaux de la grammaire G sont donnés par la table suivante :

	Débuts	Suivants
R	select id	join, union,),#

— Calcul des items LR(0) :
I_0 = { [Z → .R] [R → .R join R] [R → .R union R] [R → .select(R)] [R → .id]}
I_1= GOTO(I_0, R) = {[Z → R.] [R → R.join R] [R → R.union R]}
I_2 = GOTO(I_0, select){[R → select.(R)]}
I_3 = GOTO(I_0, id) = {[R → id.]}

I_4 = GOTO(I_1, join) ={[R → R join.R] [R → .R join R] [R → .R union R] [R → .select(R)] [R → .id]}
I_5 = GOTO(I_1, union) ={[R → R union.R] [R → .R join R] [R → .R union R] [R → .select(R)] [R → .id]}
I_6 = GOTO(I_2,() = {[R → select(.R)][R → .R join R] [R → .R union R] [R → .select(R)] [R → .id]}
I_7 = GOTO(I_4, R) = {[R → R join R.] [R → R.join R] [R → R.union R]}
I_2 = GOTO(I_4, select)
I_3 = GOTO(I_4, id)
I_8 = GOTO(I_5, R) = {[R → R union R.] [R → R.join R] [R → R.union R]}
I_2 = GOTO(I_5, select)
I_3 = GOTO(I_5, id)
I_9 = GOTO(I_6,R) = {[R → select(R.)][R → R. join R] [R → R. union R]}
I_2 = GOTO(I_6, select)
I_3 = GOTO(I_6, id)
I_4 = GOTO(I_7, join)
I_5 = GOTO(I_7, union)
I_4 = GOTO(I_8, join)
I_5 = GOTO(I_8, union)
I_{10} = GOTO(I_9,)) = {[R → select(R).]}
I_4 = GOTO(I_9, join)
I_5 = GOTO(I_9, union)

— La collection des items C ={I_0,I_1,I_2,I_3,I_4,I_5,I_6,I_7,I_8,I_9,I_{10}}.

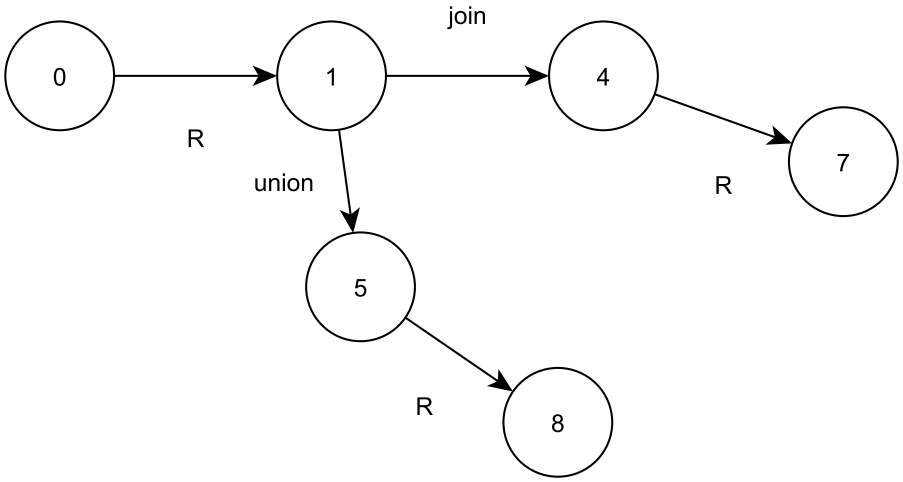
— Construction de la table d'analyse SLR(1) :

	join	union	select	(	)	id	#	R
0			D,2			D,3		1
1	D,4	D,5					Accept	
2				D,6				
3	R4	R4			R4		R4	
4			D,2			D,3		7
5			D,2			D,3		8
6			D,2			D,3		9
7	R1 D,4	R1 D,5			R1		R1	
8	R2 D,4	R2 D,5			R2		R2	
9	D,4	D,5			D,10			
10	R3	R3			R3		R3	

— La table d'analyse SLR(1) $\mathbb{M}$ est multi-définie $\Longrightarrow$ G n'est pas SLR(1). Les cas de multi-définitions peuvent être éliminés en adoptant les conventions suivantes :

(a) associativité de droite à gauche pour les opérateurs join et union

(b) les priorités dans l'ordre décroissant sont : join, union.



— à l'état 7, le contenu de la pile est le suivant : R join R

début
  si *(tc = 'join')* **alors**
    l'action de décalage est retenue car l'opérateur **join** est associatif de droite à gauche ;
  si *(tc = 'union')* **alors**
    l'action de réduction est retenue car l'opérateur **join** est plus prioritaire que l'opérateur **union**
fin

— à l'état 8, le contenu de la pile est le suivant : R union R

début
  si *(tc = 'join')* **alors**
    l'action de décalage est retenue car l'opérateur **join** est plus prioritaire que l'opérateur **union**
  si *(tc = 'union')* **alors**
    l'action de décalage est retenue car l'opérateur **union** est associatif de droite à gauche ;
fin

— Après résolution des 4 cas de conflits, la table d'analyse SLR(1) devient mono-définie comme suit :

	join	union	select	(	)	id	#	R
0			D,2			D,3		1
1	D,4	D,5					Accept	
2				D,6				
3	R4	R4			R4		R4	
4			D,2			D,3		7
5			D,2			D,3		8
6			D,2			D,3		9
7	D,4	R1			R1		R1	
8	D,4	D,5			R2		R2	
9	D,4	D,5			D,10			
10	R3	R3			R3		R3	

d. Soit à analyser la chaine **select(id)join id union id union id #**

Pile	**tc**	**Chaine**	**Action**
#⓪	select	select(id)join id union id union id #	D,2
#⓪select②	(	(id)join id union id union id #	D,6
#⓪select②(⑥	id	id)join id union id union id #	D,3
#⓪select②(⑥$\underbrace{id③}$	)	)join id union id union id #	R4
#⓪select②(⑥R ⑨	)	)join id union id union id #	D,10
#⓪$\underbrace{select②(⑥R⑨)⑩}$	join	join id union id union id #	R3
#⓪R①	join	join id union id union id #	D,4
#⓪R① join ④	id	id union id union id #	D,3
#⓪R① join ④ $\underbrace{id③}$	union	union id union id #	R4
#⓪$\underbrace{R①join④R⑦}$	union	union id union id #	R1
#⓪R ①	union	union id union id #	D,5
#⓪R ①union ⑤	id	id union id #	D,3
#⓪R ①union ⑤$\underbrace{id③}$	union	union id #	R4
#⓪R ①union ⑤R ⑧	union	union id #	D,5
#⓪R ①union ⑤R ⑧union ⑤id	id	id#	D,3
#⓪R ①union ⑤R ⑧union ⑤$\underbrace{id③}$	#	#	R4
#⓪R ①union ⑤$\underbrace{R⑧union⑤R⑧}$	#	#	R2
#⓪$\underbrace{R①union⑤R⑧}$	#	#	R2
#⓪R①	#	#	ACCEPT

Exercice 3 :

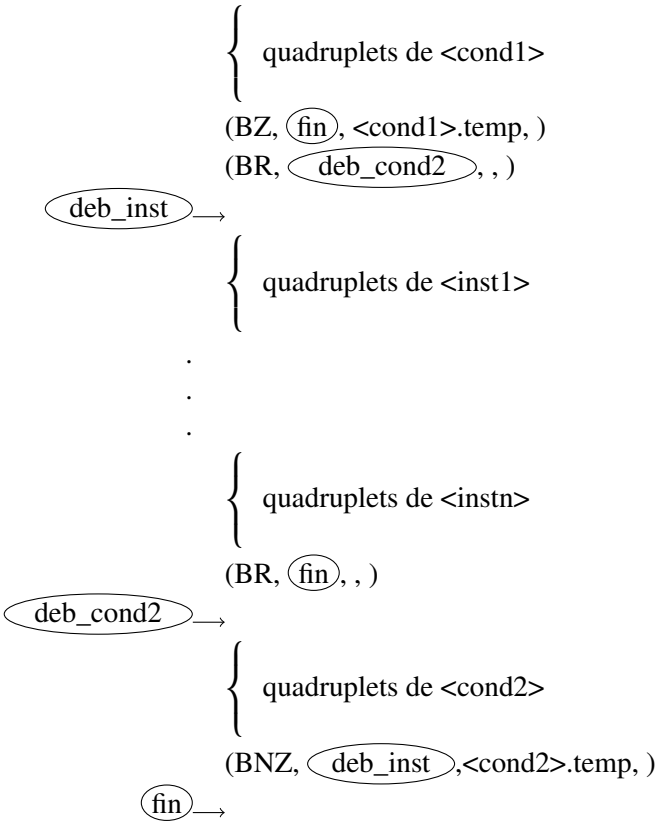
Soit l'instruction suivante :
TEST condition$_1$,(instruction$_1$,...,instruction$_n$),condition$_2$
La sémantique de l'instruction est comme suit :

> Le bloc d'instruction (instruction$_1$,...,instruction$_n$) est exécuté si condition$_1$ et condition$_2$ sont vérifiées.

1. La grammaire syntaxique :

<inst-test> → TEST <cond>,(<list-inst>),<cond>
<list-inst> → <list-inst>, <inst> | <inst>

2. Le schéma de traduction utilisant les quadruplets dans le cas d'une analyse ascendante est défini par :
a. Le code intermédiaire généré sous forme de quadruplets :



b. La grammaire sémantique :

<Inst-test> →	<A> <cond> (R3)
<A> →	<B> (<list-cond>),(R2)
<B> →	TEST <cond>,(R1)
<list-cond> →	<list-cond>,<cond> | <cond>

c. Les routines sémantiques :

Routine (R1)

début
QUAD(qc) := (BZ, , <cond>.temp,)
sauv_BZ := qc ;
qc := qc+1 ;
QUAD(qc) := (BR, , ,)
sauv_BR := qc ;
qc := qc+1 ;
sauv_ debinst :=qc ;
fin

Routine (**R2**)

début
QUAD(qc) := (BR, , ,)
sauv-fin := qc ;
qc := qc+1 ;
QUAD(sauv_BR, 2) := qc ;
fin

Routine (R3)

début
QUAD(qc) := (BNZ,sauv_debinst ,<cond>.temp ,)
qc := qc+1 ;
QUAD(sauv_BZ, 2) := qc ;
QUAD(sauv_fin, 2) := qc ;
fin

III-Les formes intermédiaires

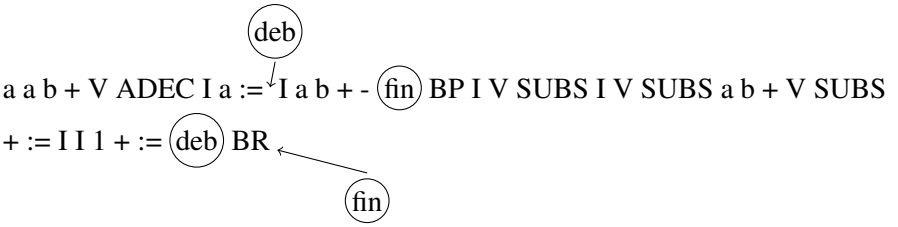
Exercice 1 :

Traduire les expressions suivantes en code postfixé, quadruplets et arbres abstraits :

1. Begin Integer Array V[a :a+b] ;
   For I := a to a+b Step 1
   Do V[I] := V[I] + V[a+b] ;
   End ;

   (a) Begin Integer Array V[a :a+b] ;
   For I := a to a+b Step 1
   Do V[I] := V[I] + V[a+b] ;
   End ;

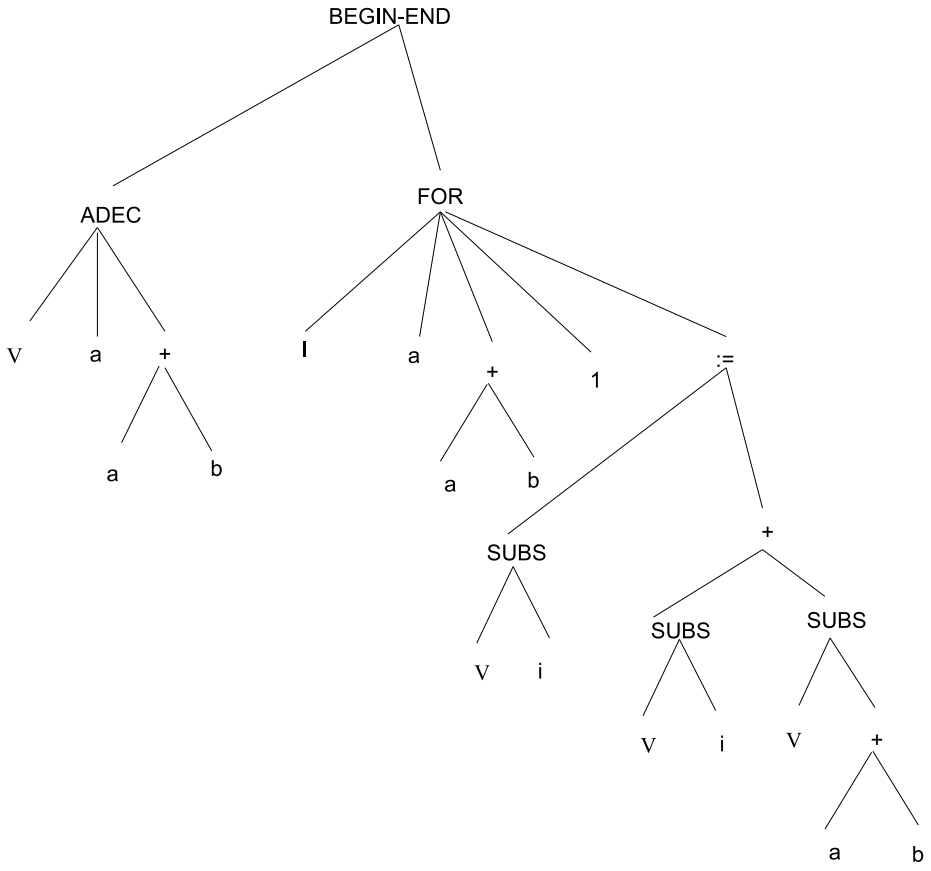
   • code postfixé :

   • quadruplets :

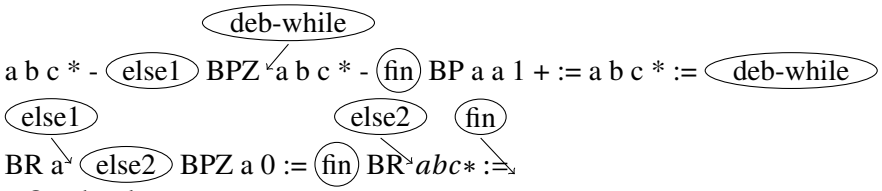
   1-(+,a,b,T_1)
   2-(BOUNDS, a, T_1,)
   3-(ADEC, V, ,)
   4-(:=, a, , I)
   deb → 5-(+, a, b, T_2)
   6-(BG, fin, I, T_2)
   7-(+, a, b, T_2)
   8-(+, V[I], V[T_2], T_3)
   9-(:=, T_3, , V[I])
   10-(+, I, 1, T_4)
   11-(:=, T_4, , I)
   12-(BR, deb, ,)
   fin → 13-

- arbre abstrait :



2. Begin If(a<b*c)
   Then While a<=b*c
   Do Begin a := a+1 ;
   a :=b*c ;
   End ;
   Else If a<0 Then a :=0 ;
   Else a :=b*c ;
   End ;

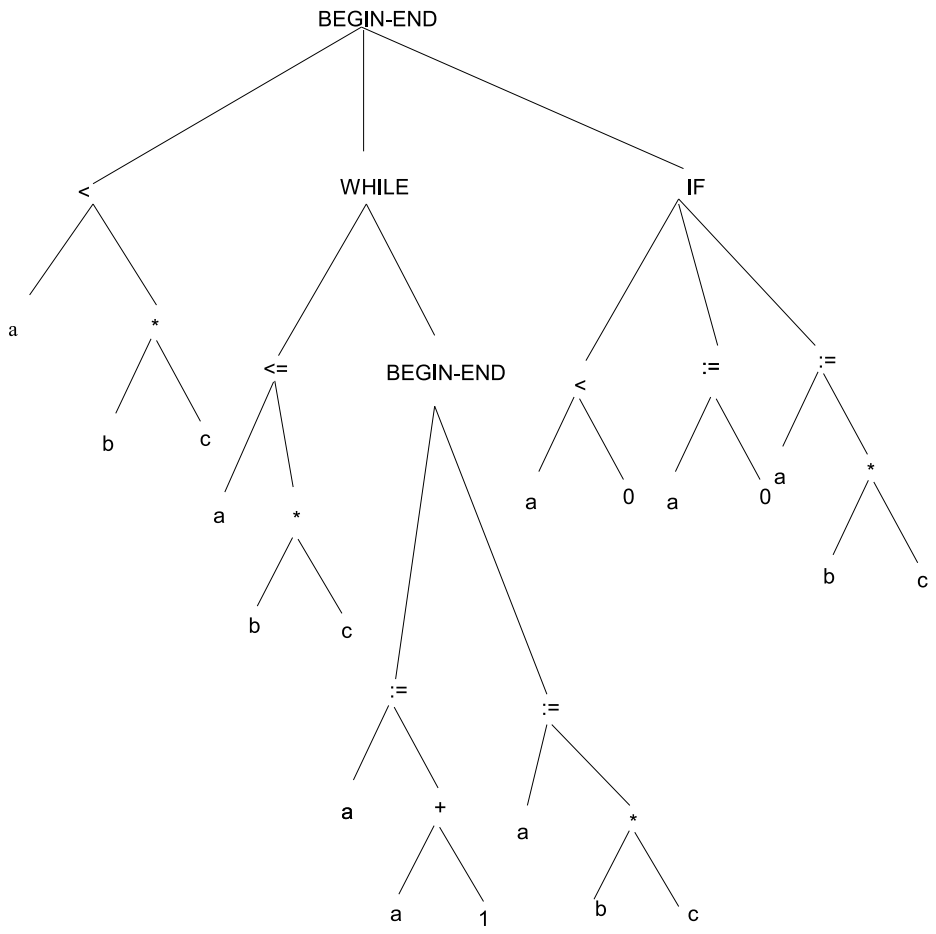
• Code postfixé :



• Quadruplets :

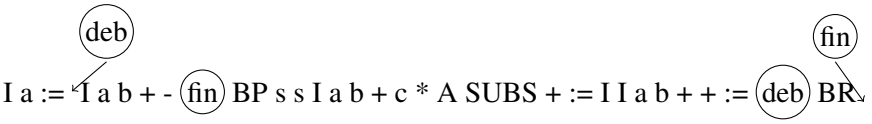
1-(*,b,c,T_1)
2-(BGE, else1, a, T_1)
deb-while → 3-(*, b, c, T_2)
4-(BG, fin, a, T_2)
5-(+, a, 1, T_3)
6-(:=, T_3, , a)
7-(*, b, c, T_4)
8-(:=, T_4, , a)
9-(BR, deb-while, ,)
else1 → 10-(BGE, else2, a, 0)
11-(:=, 0, , a)
12-(BR, fin, ,)
else2 → 13- (*, b, c, T_5)
14-(:=, T_5, , a)
fin → 15-

• Arbre abstrait :



(a) For I := a to (a+b) Step a+b
Do s := s + A[I, (a+b)*c] ;

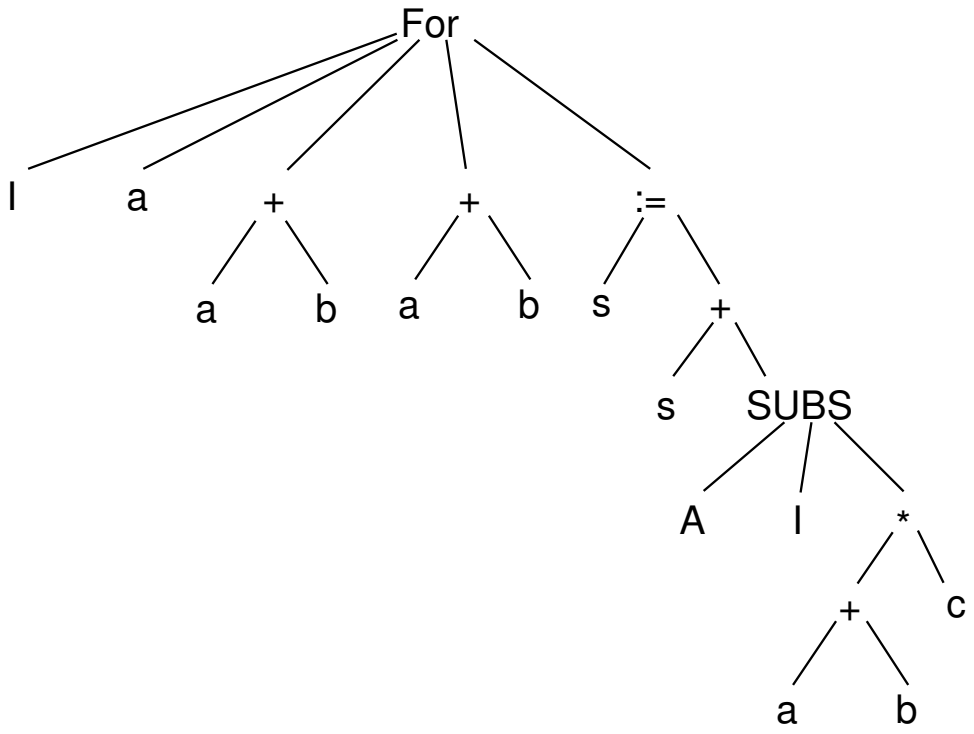
- code postfixé :



• quadruplets :

1-(:=, a, , I)
deb → 2-(+, a, b, T_1)
3-(BG, fin, I, T_1)
4-(+, a, b, T_2)
5-(*, T_2, c, T_3)
6-(+, s, A[I, T_3], T_4)
7-(+, s, T_4, T_5)
8-(:=, T_5, ,s)
9-(+, a, b, T_6)
10-(+, I, T_6, T_7)
11-(:=, T_7, , I)
12-(BR, deb, ,)
fin → 13-

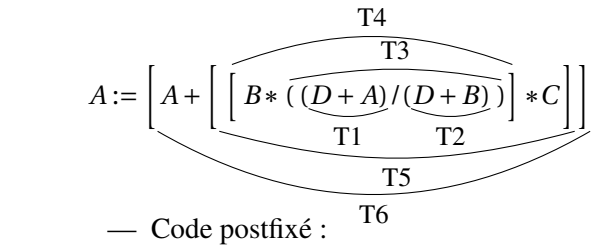
• arbre abstrait :



Exercice 2 :

Traduire les expressions suivantes en code postfixé, en quadruplets et sous forme d'arbres abstraits :

1. A := A+B*((D+A)/(D+B))*C
   L'ordre d'évaluation des sous expressions est comme suit :

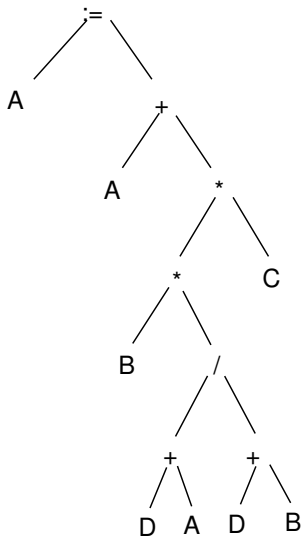


   — Code postfixé :
   A A B D A + D B + / * C * + :=

   — Quadruplets :

1-(+, D, A, T_1)
2-(+, D, B, T_2)
3-(/, T_1, T_2, T_3)
4-(*, B, T_3, T_4)
5-(*, T_4, C, T_5)
6-(+, A, T_5, T_6)
7-(:=, T_6, , A)

— Arbre abstrait :



2. P := (P*(S/2))+(X-X/Y)
L'ordre d'évaluation des sous expressions est comme suit :

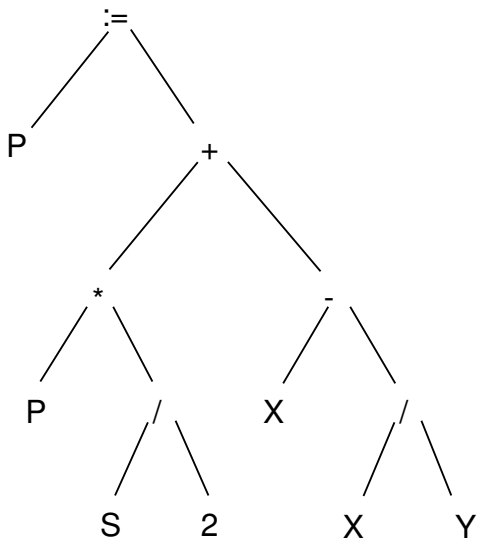
$$P := \overbrace{\Big[\overbrace{(P * \underbrace{(S/2)}_{T1})}^{T2} + \overbrace{(X - \underbrace{(X/Y)}_{T3})}^{T4} \Big]}^{T4}$$

— Code postfixé :
P P S 2 / * X X Y / - + :=

— Quadruplets :

1-(/, S, 2, T_1)
2-(*, P, T_1, T_2)
3-(/, X, Y, T_3)
4-(-, X, T_3, T_4)
5-(+, T_2, T_4, T_5)
6-(+, T_5, , P)

— Arbre abstrait :



IV- Traduction dirigée par la syntaxe

Exercice 1 :

Soit l'instruction suivante :
SI <exparithm> = NEG <suite1> NUL <suite2> POS <suite3> FINSI
La sémantique de l'instruction est comme suit :

> Si *exparithm* est négative, exécuter suite1 et aller à finisi, sinon si *exparithm* est nulle, exécuter suite2 et aller à finsi, si elle est positive alors exécuter suite3.

1. La grammaire syntaxique :

<Inst-exparithm> → Si <expartithm> = neg <Suite1> nul <Suite2> pos <Suite3> finsi

2. Le schéma de traduction utilisant les quadruplets dans le cas d'une analyse ascendante :

a. Le code intermédiaire généré :

{ quadruplets de <exparithm>

(BP, (suite3), <exparithm>.temp,)
(BZ, (suite2), <exparithm>.temp,)

{ quadruplets de <suite1>

(BR, (fin), ,)

(suite2) →

{ quadruplets de <suite2>

(BR, (fin), ,)

(suite3) →

{ quadruplets de <suite3>

(fin) →

b. La grammaire associée :

$$\begin{cases} \text{<Inst-exparithm>} \rightarrow & \text{<A> POS <suite3> FINSI } (R4) \\ \text{<A>} \rightarrow & \text{<B> NUL <suite2> } (R3) \\ \text{<B>} \rightarrow & \text{<C> = NEG <suite1> } (R2) \\ \text{<C>} \rightarrow & \text{SI <exparithm> } (R1) \end{cases}$$

c. Les routines sémantiques :

Routine (R1)

début
QUAD(qc) := (BP, , <exparithm>.temp,)
sauv-BP := qc ;
qc := qc+1 ;
QUAD(qc) := (BZ, , <exparithm>.temp,)
sauv-BZ := qc ;
qc := qc+1 ;
fin

Routine (**R2**)

début
QUAD(qc) := (BR, , ,)
sauv-BR1 := qc ;
qc := qc+1 ;
QUAD(sauv_BZ, 2) := qc ;
fin

Routine (R3)

début
QUAD(qc) := (BR, , ,)
sauv_BR2 := qc ;
qc := qc+1 ;
QUAD(sauv_BP, 2) := qc ;
fin

Routine (**R4**)

début
QUAD(sauv_BR1, 2) := qc ;
QUAD(sauv_BR2, 2) := qc ;
fin

Exercice 2 :

Soit l'instruction :
id :=MOYENNE (<$Expression_1$ >,< $Expression_2$ >,... < $Expression_p$>)

1. La grammaire syntaxique :

$$\begin{cases} \text{<Inst-moyenne>} \rightarrow & \text{id := moyenne(<list-exp>)} \\ \text{<list-exp>} \rightarrow & \text{<list-exp>, <exp> | <exp>} \end{cases}$$

2. Le schéma de traduction utilisant les quadruplets dans le cas d'une analyse ascendante :
   2- Schéma de traduction utilisant :
   a. Le code intermédiaire sous forme de quadruplets :
   (:=, 0, ,id)
   { quadruplets <Expression_1>
   (+, id, <Expression_1>.temp, id)
   .
   .
   .
   .
   .
   { quadruplets <Expression_n>
   (+, id, <Expression_n>.temp, id)
   (/, id, n, id)

   b. La grammaire associée :

{ <Inst-moyenne> → <A> := MOYENNE(<list-exp>) (R3)
{ <A> → Id (R1)
{ <list-exp> → <list-exp>, <Expression> (R2) | <Expression> (R2)

   c. Les routines sémantiques :
   Routine (R1)
   /* Elle effectue la recherche de l'identificateur dans la TS, récupère ses paramètres et initialise le compteur */
   ↓ ↓ ↑ ↑

```
début
    Lookup(tc, P) ;
    si (P=0) alors
        Écrire("Erreur : Identificateur non déclaré") ;
    sinon
        QUAD(qc) := ( :=, 0, , P.nom) ;
        qc := qc+1 ;
        id.nom := P.nom ;
        id.type := P.type ;
        compteur := 0 ;
fin
```

Routine (R2)
/* Elle contrôle la compatibilité de types entre l'expression et l'identificateur puis effectue la somme */

```
début
    si (<exp>.type et id.type sont compatibles) alors
        QUAD(qc) := (+, id.nom, <exp>.temp, id.nom) ;
        qc := qc+1 ;
        compteur := compteur+1 ;
    sinon
        Écrire("Erreur :Incompatibilité des types") ;
fin
```

Routine (R3)
/* Elle permet de calculer la moyenne */

```
début
    QUAD(qc) := (/, id.nom, compteur, id.nom)
    qc := qc+1 ;
fin
```

Exercice 3 :

Soit l'instruction suivante :
SELECT (Instruction$_1$, Instruction$_2$) Expression
La sémantique de l'instruction est comme suit :

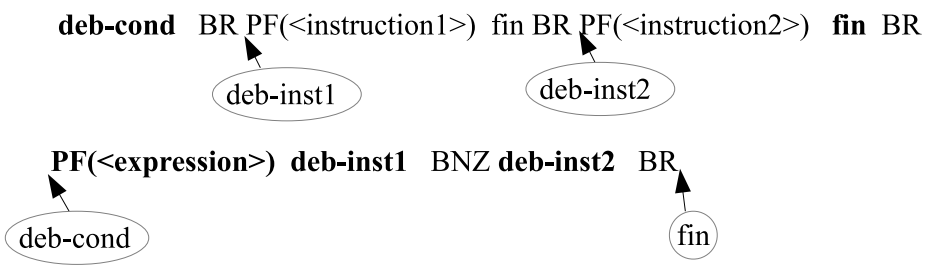
Si l'expression est vraie, exécuter Instruction$_1$ et sortir.
Si l'expression est fausse, exécuter Instruction$_2$ et sortir.

Le schéma de traduction sous forme postfixée, dans le cas d'une analyse descen-

dante est défini par :

1. La grammaire syntaxique :
   <Inst-select> → SELECT(<Instruction1>, <Instruction2>) <Expression>

2. Le schéma de traduction dans le cas d'une analyse descendante en utilisant la forme postfixée :
   a. Le code intermédiaire généré :



   b. La grammaire associée :

$$\begin{cases} \text{<Inst-select>} \rightarrow & \text{SELECT <A> (<Instuction> <B>, <Instruction> <C>) <Expression> <D>} \\ \text{<A>} \rightarrow & \epsilon \\ \text{<B>} \rightarrow & \epsilon \\ \text{<C>} \rightarrow & \epsilon \\ \text{<D>} \rightarrow & \epsilon \end{cases}$$

   c. Les routines sémantiques :
   Routine A → ϵ

```
début
    Sauv-BR := i ;
    i := i+1 ;
    PF[i] := 'BR' ;
    i := i+1 ;
    Sauv-deb-inst1 := i ;
fin
```

   Routine B → ϵ

```
début
    Sauv-Fin1 := i ;
    i := i+1 ;
    PF[i] := 'BR' ;
    i := i+1 ;
    Sauv-deb-inst2 := i ;
fin
```

Routine C $\rightarrow \epsilon$

```
début
    Sauv-Fin2 := i ;
    i := i+1 ;
    PF[i] := 'BR' ;
    i := i+1 ;
    PF(sauv-BR) := i ;
fin
```

Routine D $\rightarrow \epsilon$

```
début
    PF[i] := Sauv-deb-inst1 ;
    i := i+1 ;
    PF[i] := 'BNZ' ;
    i := i+1 ;
    PF[i] := Sauv-deb-inst2 ;
    i := i+1 ;
    PF[i] := 'BR' ;
    i := i+1 ;
    PF(Sauv-Fin1) := i ;
    PF(Sauv-Fin2) :=i ;
fin
```

Exercice 4 :

Soit l'instruction suivante :
COMBINELOOP(<Instruction1>,<Condition1>;<Instruction2>,<Condition2>;<Instruction3>)
, La sémantique associée à cette instruction est comme suit :

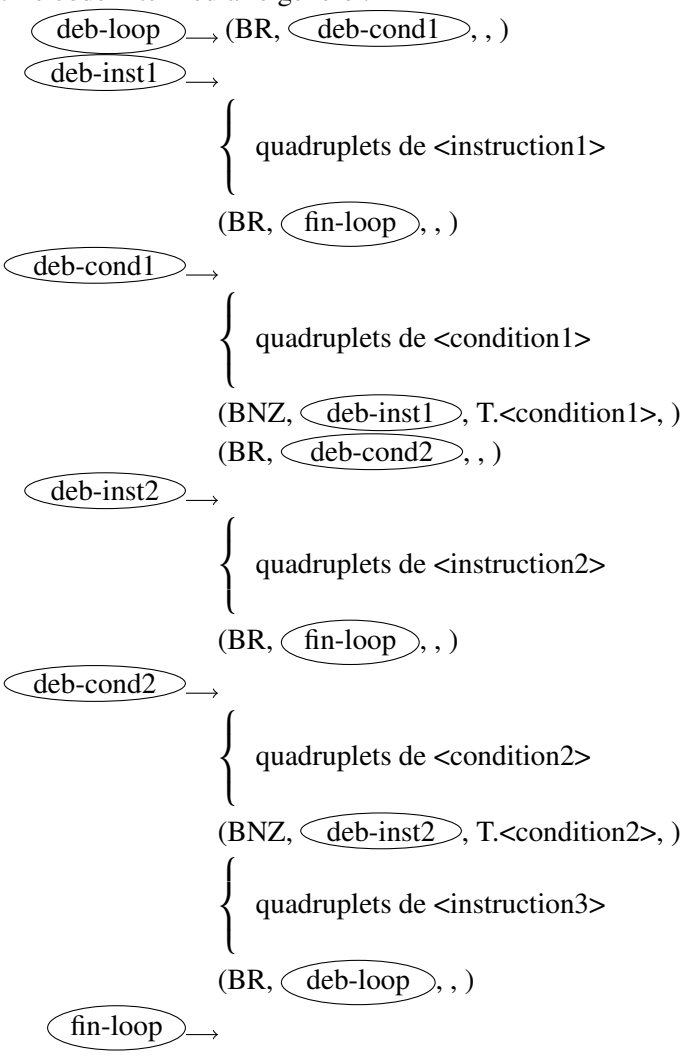
> Si la condition1 est vraie alors seule l'instruction1 est exécutée. Sinon, si la condition2 est vraie alors seule l'instruction2 est exécutée. Sinon, l'instruction3 est exécutée puis il y a un branchement vers le début de

l'instruction COMBINELOOP.

Le schéma de traduction sous forme se quadruplets, dans le cas d'une analyse ascendante est défini par :

1. La grammaire syntaxique :
   <instcombineloop> →
   COMBINELOOP(<Instruction1>,<Condition1>;<Instruction2>,<Condition2>;<Instruction3>)

2. Le schéma de traduction dans le cas d'une analyse ascendante en utilisant les quadruplets :
   a. Le code intermédiaire généré :



b. La grammaire associée :
<instcombineloop> → COMBINELOOP↓(<Instruction1>↓,<Condition1>↓;<Instruction2>↓,<Condition2>↓;<Instruction3>↓)

Dans le cas ascendant, il est nécessaire d'opérer une série de découpage de la grammaire car les routines sémantiques ne peuvent s'exécuter que lors des réductions. Le découpage de la grammaire produit de nouvelles règles de production comme suit :

<instcombineloop> → <debut1>;<instruction3>) (R6)

<debut1> → <debut2>,<condition2> (R5)

<debut2> → <debut3>;instruction2> (R4)

<debut3> → <debut4>,<condition1> (R3)

<debut4> → <debut5>(<instruction1> (R2)

<debut5> → COMBINELOOP (R1)

c. Les routines sémantiques :

Routine (R1)

```
début
    deb-loop :=qc ;
    QUAD(qc) := (BR, , ,) ;
    deb-cond1 :=qc ;
    qc := qc+1 ;
    deb-inst2 :=qc ;
fin
```

Routine (R2)

```
début
    QUAD(qc) := (BR, , ,) ;
    br-fin1 :=qc ;
    qc := qc+1 ;
    QUAD(deb-cond1,2) :=qc ;
fin
```

Routine (R3)

```
début
    QUAD(qc) := (BNZ,deb-inst1 ,T.condition1 ,) ;
    qc := qc+1 ;
    QUAD(qc) := (BR, , ,) ;
    deb-cond2 :=qc ;
    qc := qc+1 ;
    deb-inst2 :=qc ;
fin
```

Routine (R4)

```
début
    QUAD(qc) := (BR, , ,) ;
    br-fin2 :=qc ;
    qc := qc+1 ;
    QUAD(deb-cond2,2) :=qc ;
fin
```

Routine (R5)

```
début
    QUAD(qc) := (BNZ,deb-inst2 ,T.condition2 ,) ;
    qc := qc+1 ;
fin
```

Routine (R6)

```
début
    QUAD(qc) := (BR,deb-loop , ,) ;
    qc := qc+1 ;
    QUAD(br-fin1,2) :=qc ;
    QUAD(br-fin2,2) :=qc ;
fin
```