CHAPITRE 3 : ANALYSE SYNTAXIQUE : INTRODUCTION, RAPPELS ET COMPLEMENTS

3.1 Rôle d'un analyseur syntaxique

L'analyseur syntaxique (Parser en anglais) a comme rôle l'analyse des séquences d'unités lexicales, obtenues par l'analyseur lexical, conformément à une grammaire qui engendre le langage considéré.

L'analyseur syntaxique reconnaît la structure syntaxique d'un programme source et produit une représentation qui est généralement sous forme d'un arbre syntaxique. Ce dernier peut être décoré par des informations sémantiques puis utilisé pour produire un programme cible.

Pour les langages simples, l'analyseur syntaxique peut ne pas livrer une représentation explicite de la structure syntaxique du programme, mais remplit le rôle d'un module principal qui appelle des procédures d'analyse lexicale, d'analyse sémantique et de génération de code.

Ainsi, les principales fonctions d'un analyseur syntaxique peuvent être résumées comme suit :

1. L'analyse de la chaîne des unités lexicales délivrées par l'analyseur lexical, pour vérifier si cette chaîne peut être engendrée par la grammaire du langage considéré.
2. La détection des erreurs syntaxiques si la syntaxe du langage n'est pas respectée.
3. La construction éventuelle d'une représentation interne qui sera utilisée par les phases ultérieures.

3.2 Approches d'analyse syntaxique

Il existe trois grandes catégories de méthodes pour l'analyse syntaxique :

1. Méthodes universelles : Elles sont généralement tabulaires comme celles de Cocke-Younger-Kasami (1965-1967) et de Earley (1970) qui peuvent analyser une grammaire quelconque. Cependant, ces méthodes sont trop inefficaces pour être utilisées dans les compilateurs industriels.
2. Méthodes descendantes : Ces méthodes sont dites aussi ***Top-Down*** car elles construisent des arbres d'analyse de haut en bas (de la racine aux feuilles).
3. Méthodes ascendantes : Ces méthodes sont dites aussi ***Bottom-up*** car elles construisent des arbres d'analyse de bas en haut (remontent des feuilles à la racine).

Exemple d'analyse

Soit la grammaire $G = \{V_T, V_N, S, P\}$

P étant défini par les règles suivantes :

$S \rightarrow B$
$B \rightarrow R \mid (B)$
$R \rightarrow E = E$
$E \rightarrow a \mid b \mid (E+E)$

$V_T = \{a, b, (,), =, +\}$

$V_N = \{S, B, R, E\}$

L'analyse de la chaîne (a= (b + a)) par une approche descendante débute par l'axiome et s'effectue par dérivations successives comme suit (si on dérive à partir de la gauche) :

$S \rightarrow B$
$\rightarrow (B)$
$\rightarrow (R)$
$\rightarrow (E = E)$
$\rightarrow (a = E)$
$\rightarrow (a = (E + E))$
$\rightarrow (a = (b + E))$
$\rightarrow (a = (b + a))$

L'analyse de la chaîne (a= (b + a)) par une approche ascendante débute au niveau de la chaîne elle-même **(lire de bas en haut)** et remplace, à chaque étape, une partie de la chaîne par un non-terminal, par réductions successives, comme suit (si on réduit à partir de la gauche) :

S
B
(B)
(R)
(E = E)
(E = (E + E))
(E = (E + a))
(E = (b + a))
(a = (b + a))

Les méthodes ascendantes et descendantes sont utilisées fréquemment pour la mise en oeuvre de compilateurs. Dans les deux cas, l'entrée de l'analyseur syntaxique est parcourue de la gauche vers la droite, un symbole à la fois. Les méthodes ascendantes et descendantes les plus efficaces fonctionnent uniquement avec des sous-classes de grammaires telles que les grammaires LL et LR (qui seront étudiées aux chapitres 4 et 5) qui sont suffisamment expressives pour décrire la majorité des structures syntaxiques des langages de programmation.

Les analyseurs syntaxiques mis en œuvre manuellement utilisent généralement des méthodes descendantes et des grammaires LL. C'est le cas, par exemple, pour l'analyse prédictive et la descente récursive.

Les méthodes ascendantes sont souvent utilisées par les outils de construction des analyseurs syntaxiques. C'est le cas avec les analyseurs LR qui sont des analyseurs plus généraux que les analyseurs descendants.

La construction d'analyseurs LR est courante dans les environnements de développement de compilateurs. Notons qu'il existe aussi d'autres méthodes ascendantes telles que la précédence d'opérateurs, la précédence simple, la précédence faible et les matrices de transition.

3.3 Concepts de base

3.3.1 Notion de grammaire

Une grammaire G est un formalisme qui permet la génération des chaînes d'un langage. Elle est formellement définie par un quadruplet $G = (V_T, V_N, S, P)$ où

- V_T est le vocabulaire terminal contenant l'ensemble des symboles atomiques individuels, pouvant être composés en séquence pour former des phrases et souvent notés en lettres minuscules.

- V_N est le vocabulaire non terminal contenant l'ensemble des symboles non terminaux représentant des constructions syntaxiques. Ces symboles sont souvent notés en majuscules.
- S est appelé axiome ou symbole initial de la grammaire.
- P est l'ensemble des règles de production de la forme $\alpha \rightarrow \beta$ où α et β appartiennent à $(V_T \cup V_N)^*$

Selon la forme des règles de production, il existe quatre types de grammaires que nous présentons selon la classification de Chomsky. Dans la suite nous considérons que si V est un vocabulaire alors V* est l'ensemble de toutes les chaînes résultant de la concaténation des éléments de V.

Type de grammaire	**Forme des règles de production**
Type 0 (Grammaires sans restrictions)	$\alpha \rightarrow \beta$ avec $\alpha \in (\mathbf{V_T} \cup \mathbf{V_N})^+$ et $\beta \in (\mathbf{V_T} \cup \mathbf{V_N})^*$ Autrement dit il n' y a aucune restriction sur α et β
Type 1 (Grammaires à contexte lié ou Contextuelles)	$\alpha \rightarrow \beta$ avec $\alpha \in (\mathbf{V_T} \cup \mathbf{V_N})^+$ et $\beta \in (\mathbf{V_T} \cup \mathbf{V_N})^*$ et $\lvert\alpha\rvert \leq \lvert\beta\rvert$ et ε (la chaîne vide) ne peut être généré que par l'axiome avec la règle $S \rightarrow \varepsilon$
Type 2 (Grammaires à contexte libre, non contextuelles, algébriques ou encore de Chomsky)	$\mathbf{A} \rightarrow \alpha$ avec $\mathbf{A} \in \mathbf{V_N}$ et $\alpha \in (\mathbf{V_T} \cup \mathbf{V_N})^*$
Type 3 (Grammaires régulières ou linéaires droites)	$\mathbf{A} \rightarrow \alpha\ \mathbf{B}$ ou $\mathbf{A} \rightarrow \alpha$ avec $\mathbf{A, B} \in \mathbf{V_N}$ et $\alpha \in \mathbf{V_T}^*$

3.3.2 Expression des grammaires

A part l'énumération des règles de production, il est possible de définir une grammaire en utilisant des notations textuelles telles que la **forme BNF** ou la **forme EBNF** ainsi que des représentations graphiques telles que les **diagrammes syntaxiques**.

a- La notation BNF (Backus-Naur Form)

Cette notation est due aux créateurs de Fortran (J. Backus, 1955) et Algol (P. Naur, 1963). Les règles sont écrites avec les conventions suivantes :

- Un symbole $\mathbf{A} \in \mathbf{V_N}$ est noté par **<A>** ou **A** (en utilise des lettres majuscules)
- Un symbole $\mathbf{a} \in \mathbf{V_T}$ est noté sans les **< >** ou encore par **"a"** ou **a** (en utilise des lettres minuscules)
- Le signe $\rightarrow$ devient **::=**
- Un ensemble de règles $\mathbf{A} \rightarrow \alpha$, $\mathbf{A} \rightarrow \beta$, ..., $\mathbf{A} \rightarrow \gamma$ sera remplacé par $\mathbf{A} \rightarrow \alpha|\beta|...|\gamma$

Exemple

<Bloc> ::= **Begin** <List Opt Inst> **End .**

<List Opt Inst> ::= <List Inst> | ε

<List Inst> ::= < List Inst>**;** <Inst> | <Inst>

<Inst> ::= **id :=** <Exp> | **if** <Exp> **then** <Inst> | **if** <Exp> **then** <Inst> **else** <Inst>

b- La notation EBNF (Extended Backus-Naur Form)

C'est une représentation plus réduite que BNF qui permet d'écrire les grammaires sous une forme plus condensée en utilisant les signes {} et [].

- Une partie optionnelle est notée entre [] (apparaît 0 ou 1 fois)
- Une partie qui peut apparaître de façon répétée est notée entre {} (apparaît 0 ou N fois)

Notons que, dans les notations BNF et EBNF, les parenthèses peuvent être utilisées pour éviter les ambiguïtés, comme c'est le cas dans la règle suivante :

<Inst> ::= For **id :=** <Exp> (**to** | **downto**) <Exp> **do** <Bloc>

la partie (**to** | **downto**) exprime un choix entre **to** et **downto**

Exemple

L'exemple présenté avec la notation BNF peut être noté comme suit en utilisant EBNF :

<Bloc> ::= **Begin** [<List Inst>] **End .**

<List Inst> ::= <Inst> {**;** <Inst>}

<Inst> ::= **id :=** <Exp> | **if** <Exp> **then** <Inst> [**else** <Inst>]

Cette représentation est la même que la suivante :

<Bloc> ::= **Begin** [<Inst> {**;** <Inst>}] **End .**

<Inst> ::= **id :=** <Exp> | **if** <Exp> **then** <Inst> [**else** <Inst>]

c- Les diagrammes syntaxiques

C'est une représentation sous forme graphique des règles du langage où les terminaux sont **encerclés** et les non terminaux sont **encadrés**. La figure 3.1 donne les diagrammes syntaxiques correspondant à l'exemple précédent.

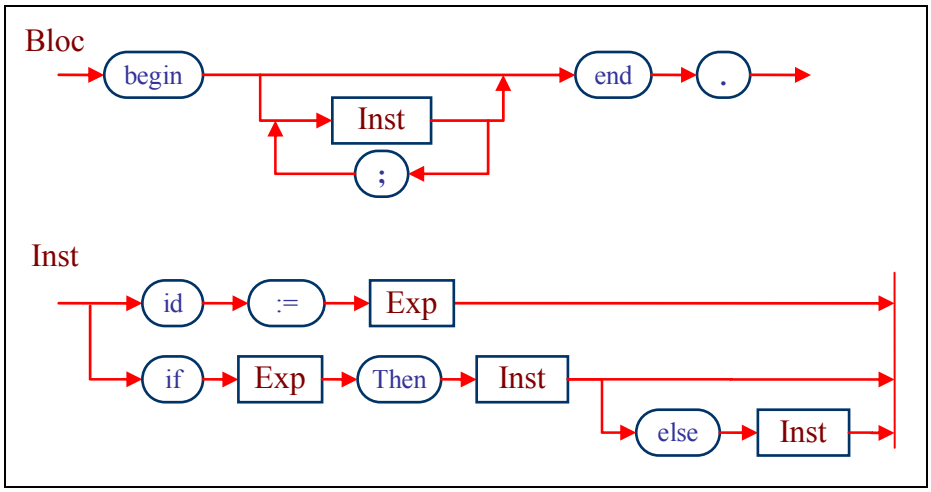

Figure 3.1. Exemple de diagrammes syntaxiques

3.3.3 Dérivation et arbres syntaxiques

a- Dérivation

On appelle dérivation, l'application d'une ou plusieurs règles à partir d'une phrase (chaîne) de $(V_T \cup V_N)^+$.

On note $\rightarrow$ une dérivation obtenue par l'application d'une seule règle de production et on note $\overset{*}{\rightarrow}$ une dérivation obtenue par l'application de plusieurs règles de production.

Exemple

Soit les règles :

$S \rightarrow A$

$A \rightarrow bB$

$B \rightarrow c$

La chaîne bc peut être dérivée par l'application de ces règles comme suit :

$S \rightarrow A$
$\rightarrow bB$
$\rightarrow bc$

On peut noter ces dérivations par une expression plus condensée $S \overset{*}{\rightarrow} bc$

D'une manière générale, si G est une grammaire définie par (V_T, V_N, S, P) alors le langage généré peut être noté par : $L(G) = \{x \ V_T$ tel que $S \rightarrow x \}$.

On distingue deux façons d'effectuer une dérivation :

- Dérivation gauche (la plus à gauche ou « left most ») : Elle consiste à remplacer toujours le symbole non terminal le plus à gauche.
- Dérivation droite (la plus à droite ou« right most ») : Elle consiste à remplacer toujours le symbole non terminal le plus à droite.

Exemple

Soit G = ({a, +, (,)}, {E, F}, E, P)

P est défini par :

$E \rightarrow E + F$
$E \rightarrow F$
$F \rightarrow (F) \mid a$

Peut-on affirmer que a+a appartient au langage L(G) ?

Par dérivations gauches, on a :

$E \rightarrow E + F \rightarrow F + F \rightarrow a + F \rightarrow a + a$ donc $E \overset{*}{\rightarrow} a+a$ et la chaîne appartient au langage

Par dérivations droites, on a :

$E \rightarrow E + F \rightarrow E + a \rightarrow F + a \rightarrow a + a$ donc $E \overset{*}{\rightarrow} a+a$ et la chaîne appartient au langage

b- Arbre de dérivation (Arbre syntaxique)

On appelle **arbre de dérivation** ou **arbre syntaxique concret** un arbre tel que

1. La racine est l'axiome de la grammaire
2. Les feuilles de l'arbre sont des symboles terminaux
3. Les nœuds sont les symboles non terminaux
4. Pour un nœud interne d'étiquette A, le mot $\alpha_1\alpha_2 \quad \alpha_n$ obtenu en lisant de gauche à droite les étiquettes des nœuds fils de A correspond à une règle de la grammaire $A \rightarrow \alpha_1\alpha_2 \quad \alpha_n$
5. Le mot m, dont en fait l'analyse, est constitué des étiquettes des feuilles, lues de gauche à droite

Exemple

Soit G = ({a, b, c}, {S, T}, S, P)

P est défini par :

$S \rightarrow aTb \mid c$
$T \rightarrow cSS \mid S$

L'arbre de dérivation de accacbb est donné par la figure 3.2.

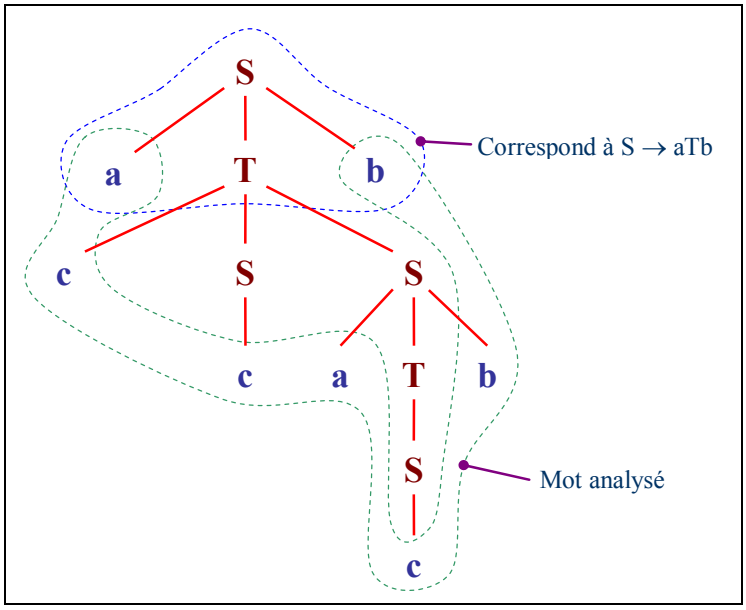

Figure 3.2. Exemple d'arbre de dérivation

c- Arbre abstrait

On appelle **arbre syntaxique abstrait** ou **arbre abstrait** une forme condensée d'arbre syntaxique qui est utilisée dans les compilateurs. L'arbre abstrait est obtenu par des transformations simples de l'arbre de dérivation.

L'arbre syntaxique abstrait est plus compact que l'arbre syntaxique concret et contient des informations sur la suite des actions effectuées par un programme. Dans un arbre abstrait, les opérateurs et les mots clés apparaissent comme des nœuds intérieurs et les opérandes apparaissent comme des feuilles.

Exemple

Si on considère, la grammaire G = ({a, +, (,)}, {E, T, F}, E, P)

P étant défini par :

E → T | E + T

T → F | T * F

F → a | (E)

Les arbres syntaxiques (concret et abstrait) pour la chaîne a + a * a sont donnés par la figure 3.5.

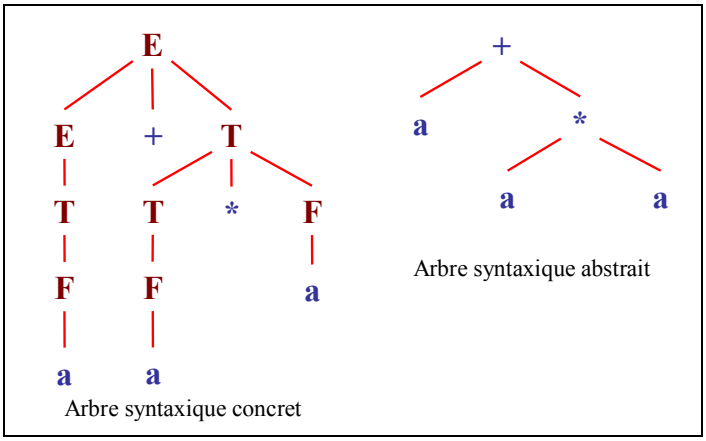

Figure 3.4. Arbre syntaxique concret et abstrait pour la chaîne a + a * a

d- Grammaire ambiguë

On dit qu'une grammaire est ambiguë si elle produit plus d'un arbre syntaxique pour une chaîne donnée.

Si on considère la grammaire G = ({+, -, *, /, id}, {E}, E, P) où P est défini par :

E → E + E | E – E | E * E | E / E | (E) | -E | id

La figure 3.3 donne deux arbres de dérivation pour la chaîne id+id* id

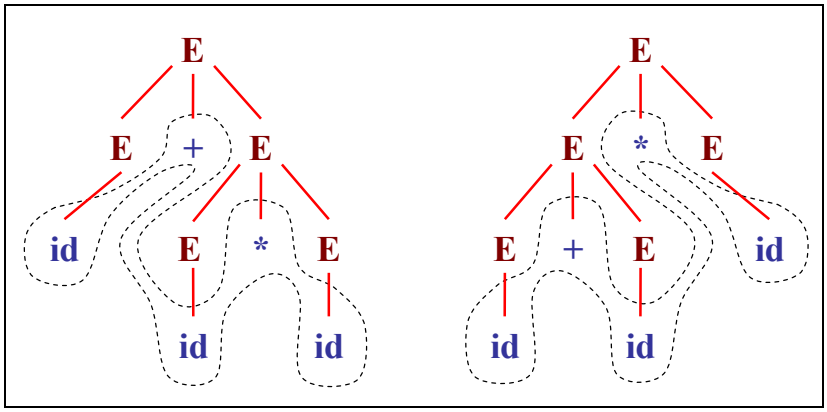

Figure 3.3. Deux arbres de dérivation pour la chaîne id + id * id

L'élimination de l'ambiguïté est une tâche importante qu'on exige généralement pour les langages de programmation qui sont basés sur des grammaires non contextuelles. L'élimination de l'ambiguïté peut avoir lieu de deux façons :

1. Elimination de l'ambiguïté par ajout de règles. Dans cette approche, la grammaire reste inchangée, on ajoute, à la sortie de l'analyseur syntaxique, quelques règles dites de désambiguïté, qui éliminent les arbres syntaxiques non désirables et conservent un arbre unique pour chaque chaîne. On peut, par exemple, éliminer l'ambiguïté en affectant des priorités aux opérateurs arithmétiques. Dans les expressions arithmétiques, la priorité est affectée aux parenthèses, puis au signe mois unaire, puis à la multiplication/division et enfin à l'addition/soustraction.

2. Réécriture de la grammaire. Il s'agit de changer la grammaire en incorporant des règles de priorité et d'associativité. Autrement dit, on ajoute de nouvelles règles et de nouveaux symboles non terminaux. La grammaire ambiguë donnée précédemment peut être désambiguïsée comme suit :

G = ({+, -, *, /, id}, {E, T, F, L}, E, P) où P est défini par :

E → E + T | E – T | T
T → T * F | T / F | F
F → -F | L
L → (E) | id

La chaîne id + id * id a maintenant une seule séquence de dérivations (gauches) :

E → E + T → T + T → F + T → L + T → id+ T → id + T*F
→ id + F*F → id + L*F→ id + id*F→ id + id*L → id + id*id

et l'unique arbre syntaxique donné par la figure 3.4.

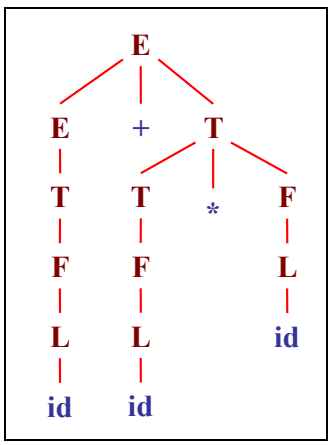

Figure 3.4. Arbre syntaxique de id + id * id

SERIE DE TD N°:2 COMPILATION
ANALYSE SYNTAXIQUE : INTRODUCTION, RAPPELS ET COMPLEMENTS

Exercice 1

1) Soit la grammaire G = ({a, b }, {S, A, B, C}, S, P) où P est défini par :

S → AB | BC

A → BA | a

B → CC | b

C → AB | a

Analyser la chaîne **babaab** de manière descendante puis ascendante, en construisant, à chaque fois, son arbre de dérivation.

Exercice 2

1) Soit la grammaire des expressions arithmétiques G = ({+, -, *, /, a, b, c, (,) }, {E}, E, P) où P est défini par :

E → E + E | E – E | E * E | E / E | (E) | -E | a | b | c

Donner l'arbre de dérivation de la chaîne **b + a * b - c** ? Que peut-on en déduire ?

2) Soit la grammaire G' = = ({+, -, *, /, a, b, (,) }, {E, T, F, L}, E, P) où P est défini par :

E → E + T | E – T | T

T → T * F | T / F | F

F → -F | L

L → (E) | a | b | c

a) Donner l'arbre de dérivation de la chaîne **b + a * b - c**. Que peut-on en déduire ?

b) Donner l'arbre abstrait correspondant à la chaîne **b + a * b - c**.

c) Donner les arbres syntaxiques concret et abstrait de la chaîne **b - a + c**.

Exercice 3

1) Soit la grammaire des expressions booléennes G = ({ou , et, non, vrai, faux, (,) }, {A}, A, P) où P est défini par :

A → A ou A | A et A | non A | (A) | vrai | faux

Donner l'arbre de dérivation de la chaîne **non faux ou vrai et vrai** ? Que peut-on en déduire ?

2) Soit la grammaire G' = ({ou , et, non, vrai, faux, (,) }, {A, B, C, D}, A, P) où P est défini par :

A → A ou B | B

B → B et C | C

C → non C | D

D → (A) | vrai | faux

Donner l'arbre de dérivation de la chaîne **non faux ou vrai et vrai** ? Conclusion ?

Exercice 4

Soit la grammaire G d'un langage proche de Pascal, exprimée sous forme EBNF de la manière suivante, exprimer la grammaire G sous forme de diagrammes syntaxiques.

<Programme> ::= Program ident ; <Bloc>.

<Bloc> ::= [Const <SuitConst> ;] [Var <SuitVar> ;] { Procedure ident ; <Bloc> ;}
Begin <Inst> {; <Inst> } End

<SuitConst> ::= <DecConst> { , < DecConst>}

<DecConst> ::= ident = nbEnt

<SuitVar> ::= ident {, ident}

<Inst> ::= ident := <Exp> | If <Cond> Then <Inst> [Else <Inst>]
| Repeat <Inst> Until <Cond> | While <Cond> Do <Inst> | Begin <Inst> {; <Inst> } End

<Cond> ::= <Exp> (= | <> | < | > | <= | >=) <Exp>

<Exp> ::= <Terme> { (+ | -) <Terme>}

<Terme> ::= <Facteur> { (* | /) <Facteur>}

<Facteur> ::= ident | nbEnt | (<Exp>)