

**ECOLE HASSANIA DES TRAVAVAUX PUBLICS**

**THÉORIE DE COMPILATION**

**RAPPORT PROJET THEORIE DE COMPILATION**

**COMPILATEUR C-PASCAL**

**- AMÉLIORATION : AJOUT DU TYPE STRUCT -**

**Réalisé par :**

ABCHIR OUIJDANE

**Encadré par :**

M. Elghazi Souhail

## Table des matières

[Introduction. 3](#_TOC_250012)

[Chapitre 1 : caractéristiques du langage C-Pascal 4](#_TOC_250011)

1. Description du langage C-Pascal 5
2. Grammaire améliorée du langage C-Pascal 6
3. [Exemple d’un programme C-Pascal 7](#_TOC_250010)

Chapitre 2 : analyse lexicale. 8

1. Alphabet. 9
2. Unités lexicales/Lexemes/Attributs/Modèles. 9
3. [Diagramme d’état complet de l’analyseur lexical. 10](#_TOC_250009)
4. Structures utilisées. 13
5. Programme de test lexical et son résultat. 13

[Chapitre 3 : analyse syntaxique. 18](#_TOC_250008)

1. [Rendre la grammaire LL( 1 ). 19](#_TOC_250007)
2. [Les fonctions de l’analyseur syntaxique. 24](#_TOC_250006)
3. L’arbre syntaxique. 13
4. Programme de test syntaxique et son résultat. 13

Chapitre 4 : analyse sémantique. 26

1. [Les fonctions de l’analyseur sémantique. 24](#_TOC_250006)
2. Programme de test sémantique et son résultat. 13

Chapitre 5 :bilan du projet 30

1. Les apports du projet[. 31](#_TOC_250004)
2. Les difficultés rencontrées[. 32](#_TOC_250003)
3. Les perspectives[. 32](#_TOC_250003)

[Conclusion. 39](#_TOC_250000)

## Introduction

Notre projet du module “théorie de la compilation” consiste à réaliser un petit compilateur d’un sous-ensemble du langage C et du langage Pascal appelé “C-Pascal”, en implémentant sa partie lexicale, son syntaxique, son sémantique, son générateur de code et en fin son interpréteur P-Machine.

L’amélioration que nous allons ajouter consiste à intégrer le type struct “struct” à la grammaire du langage “ C-Pascal ” donné. On manipulera donc les structures à l’aide des variables simples uniquement.

Dans le présent rapport nous allons tout d’abord présenter la grammaire améliorée de notre langage, puis nous allons donner les différentes composantes de son lexicale, suivie par sa partie syntaxique dans laquelle par la suite nous allons intégrer la sémantique, suivie par la partie de génération du code, puis l’interpréteur P-Machine et en fin dans la conclusion on va présenter les apports que nous a apporté ce projet.

**CHAPITRE 1 : CARACTÉRISTIQUES DU LANGAGE C-PASCAL**

Dans cette première partie nous allons aborder les aspects généraux de notre projet, à savoir sa description et sa grammaire améliorée.

## Description du langage source

Il s’agit d’un sous ensemble du langage C et du langage Pascal

dont les caractéristiques sont les suivantes :

### Structure de programme :

* La structure d’un programme est la suivante : <déclarations> <fonctions>
* Toute variable doit être déclarée avant utilisation

### Les identificateurs :

* Il n’y a aucune distinction entre les minuscules et les majuscules dans le cas des identificateurs ;
* Chaque identificateur doit commencer par une lettre ;
* Ils ne peuvent porter le nom d’un mot clé réservé : entier, alors, struct, tantque, si, faire, sinon, lire, retour, ecrire, main.
* Ils ne peuvent être déclarés qu'en début de programme (globales) ou qu'au début de la définition d'une fonction (locales).

### Les variables :

* Elles ne peuvent être déclarées qu’en début du programme (on parle de variables

globales) ou qu’au début de la définition d’une fonction (les variables locales).

* Les variables sont de types entier ou struct ou tableau d’entiers.
* On peut déclarer plusieurs variables en séparant les déclarations les unes des autres par une virgule.

.

### Les constantes :

* Elles sont des entiers exprimés en base 10.

### Les fonctions :

* Le langage C-Pascal permet de définir des fonctions à résultat entier. Leurs arguments sont simples (i.e. des entiers, les tableaux sont des variables globales ) et le passage se fait par valeur. Les fonctions peuvent avoir des variables locales.

Les parties d’une fonction doivent être déclarées dans l’ordre suivant :

* + en-tête,
  + déclaration des variables locales,
  + bloc d’instructions du corps de la fonction.

La structure d'une fonction est donc :

  identificateur (<liste\_paramètres>)

<declarations>

{

<instructions>

}

* A la différence du C, les variables locales sont déclarées entre l’en-tête et le bloc d’instructions.
* L’en- tête d’une fonction est un identificateur pour nommer la fonction, suivi d’une liste de déclarations d’arguments entre parenthèses.
* Une fonction ne peut pas être déclarée à l’intérieur d’une autre : les fonctions sont toutes globales.
* Un programme en C-Pascal est une suite de déclarations de fonctions.
* On ne peut pas déclarer deux fonctions avec le même nom (pas de surcharge) et on doit déclarer toutes les fonctions appelées (sauf les fonctions spéciales lire et ecrire).
* La fonction appelée en premier dans un programme en C-Pascal est celle qui s’appelle main.
* L’appel d’une fonction comme si c’était une procédure – c’est-à-dire en ignorant le résultat rendu – est permis, comme en C.

### Les instructions considérées sont les suivantes :

* Le langage C-Pascal connaît les instructions suivantes :
  + Affectation : a = b + 1; (contrairement à C, une affectation n’est pas une expression, elle ne correspond pas à une valeur).
  + **si** expression **alors** { ... } et **si** expression **alors** { ... } **sinon** { ... }
  + **tantque** expression **faire** { ... }
  + **retour** expression ;
  + Appel de fonction simple: nom\_fonction( liste d’expressions ); et à la fonction prédéfinie ecrire( expression );
  + Bloc d’instructions, délimité par des accolades { ... }
  + instruction vide ;

### Les opérations arithmétiques :

* Tous les opérateurs sont associatifs à gauche (c’est-à-dire, 5 − 3 + 2 équivaut à (5 − 3) + 2). Voici dans l’ordre décroissant la priorité des opérateurs :
* **1 ( expression )**
* **2 ! (non)**
* **3 \*** , **/**
* **4 +** ,  **-**
* **5 =** ,  **< , < = , > , =>**
* **6 & (et)**
* **7 | (ou)**

### Les fonctions prédéfinies :

### Procédures prédéfinies: Les entrées-sorties de base se présentent comme des appels à des fonctions prédéfinies lire et ecrire.

### Les commentaires :

* Les commentaires débutent par # et sont sur une seule ligne.

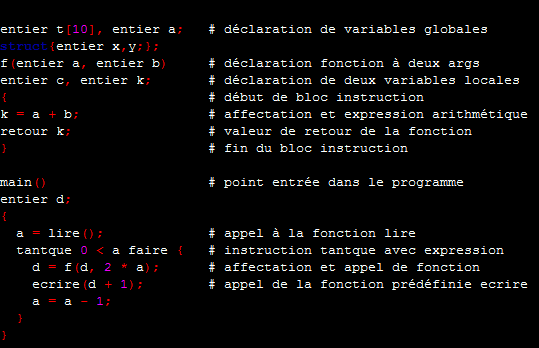
## La grammaire améliorée du langage C-PASCAL

1. <Programme> : : = < liste-declarations>< liste-fonctions> **main** ( )<bloc>
2. < liste-declarations> : : = < liste-declarations><declaration> | **epsilon**
3. <declaration> : : = <declaration>, < declarateur-cp> | < declarateurcp>;
4. <declaration-sp> : : = <declaration-sp>,< declarateur-sp>

|<declarateursp>;

1. <declarateur-sp> : : = **entier** <variable> | **entier** <variable> [<nombre]
2. <declarateur-cp> : : = **entier** <variable> | **entier** <variable> [<nombre>] **| struct** <variablestruct>**{**<declaration-sp>}
3. < liste-fonctions> : : = < liste-fonctions> <fonction> | **epsilon**
4. <fonction> : : = <identificateur> ( <liste-parms> ) < liste-declarations><bloc>
5. <liste-parms> : : = <liste-parms> , <parm> | <parm>
6. <parm> : : = **entier** <variable> | **entier** <variable> [<nombre>]
7. <bloc> : : = { <liste-instructions> }
8. <liste-instructions> : : = <liste-instructions>< instruction > | **epsilon**
9. <instruction> : : = < iteration> I <selection> | <saut> | <affectation><bloc> I <appel>| **epsilon** ;
10. <iteration> : : = **tanque** <condition> **faire** <bloc>
11. <selection> : : = **si** <condition> **alors** <bloc> | **si** <condition> **alors** <bloc> **sinon** <bloc>
12. <saut> : : = **retour** <expression>
13. <affectation> : : = <variable> = <expression> | <variablestruct>. <variable>= <expression> | <variable> = **lire** ( )
14. <appel> : : = <identificateur> ( <liste-expressions>) | **ecrire** (<expression>)
15. <liste-expressions> : : = <liste-expressions>,<expression> | <expression>
16. <condition> : : = !(<expression>) | <expression><binary-rel><expression> | <expression><binary-comp><expression>
17. <expression> : : = <terme> | <terme><binary-op><terme >
18. <terme>  : : = <variable> | <nombre> | <expression>
19. <variable>  : : = <variable>|<lettre>
20. <variablestruct>  : : = <variablestruct>|<lettre>
21. <nombre>  : : = <nombre><constante> | <constante>
22. <identificateur>  : : = <identificateur><lettre> | <lettre>
23. <lettre>  : : = A|B|C|...|Z|a|b|c|...­|z
24. <constante>  : : = 0|1|…­|9
25. <binary-op> : : = +|-|\*|/
26. <binary-rel> : : = **&** | |
27. <binary-comp> : : = <|>|>=|<=|==|!=

## Exemple d’un programme C-PASCAL



**CHAPITRE 2: ANALYSE LEXICALE**

1. **L’alphabet**

Voici l’alphabet du grammaire :

∑ = {A , B , C , D , E , F , G , H , I , J , K , L , M , N ,O , P , Q , R , S , T , U , V ,

W , X , Y , Z, a , b , c , d , e , f , g , h , i , j , k , l , m , n , o , p , q , r , s , t , u , v ,

w , x , y , z, 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9, +, /, \*, -, >, <, |, &, =, !, ,, ), (, [, ], }, {, ;, #, . }

## Unités lexicales / Lexèmes / Attributs / Modèles

|  |  |  |  |
| --- | --- | --- | --- |
| Unité lexicale | Lexème | Attribut | Modèle |
| COMMENT | ‘’# ‘’ | X | # |
| PT | ‘’.’’ | X | . |
| BIN\_OP | Ex: “+”,”/”…. | Indice dans la table des opérateurs binaires | +|/|\*|- |
| BIN\_COMP | Ex:“<”,”>”,== | Indice dans la table des opérateurs binaires de comparaisons | <|>|>=|<=|==|!= |
| AND | “&” | X | & |
| OR | “|” | X | | |
| CROCH\_OUV | “[“ | X | [ |
| CROCH\_FER | “]” | X | ] |
| PAR\_OUV | “(“ | X | ( |
| PAR\_FER | ‘’)’’ | X | ) |
| COMMENT | ‘’#’’ | X | # |
| NON | ‘’ !’’ | X | ! |
| VERG | *‘’,’’* | X | , |
| AFF | *‘’=’’* | X | = |
| ACC\_OUV | *‘’{‘’* | X | { |
| ACC\_FER | *‘’}’’* | X | } |
| PT\_VER | *‘’;‘’* | X | ; |
| MOT\_CLE | *Ex :retour*  *Ex : entier* | Indice dans la table des mots clés | sinon|alors|si|tanque| entier|retour|faire |
| NOMBRE | ex : 15 | La valeur de l’entier | (0)|(1|2….|9)\*(0|1|…..9)\* |
| MOT\_RESERVE | *Ex :main* | Indice dans la table des mots réservés | lire|ecrire|main |
| IDENT | EX : MACHAN | L’indice dans la table des identificateurs | Lettre(lettre|chiffre)\* |

## Diagramme d’état complet de l’analyseur lexical

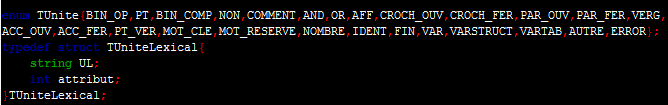
-> Les erreurs :

▶ 1 : “ Chiffre attendu “

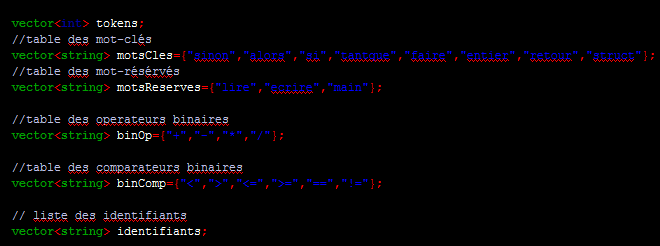
▶ 2 : “ Charactère non appartenant au langage “

## Les structures utilisées

* **TUnite**
* **TUniteLexicale**



* **Tables utilisées**

****

* **Fichier Lexical.h**

**#ifndef LEXICAL\_H**

**#define LEXICAL\_H**

**#include <string>**

**#include <vector>**

**using namespace std;**

**enum TUnite{BIN\_OP,PT,BIN\_COMP,NON,COMMENT,AND,OR,AFF,CROCH\_OUV,CROCH\_FER,PAR\_OUV,PAR\_FER,VERG,**

**ACC\_OUV,ACC\_FER,PT\_VER,MOT\_CLE,MOT\_RESERVE,NOMBRE,IDENT,FIN,VAR,VARSTRUCT,VARTAB,AUTRE,ERROR};**

**typedef struct TUniteLexical{**

**string UL;**

**int attribut;**

**}TUniteLexical;**

**class Lexical**

**{**

**public:**

**Lexical(string s);**

**void readfichier();**

**void fin();**

**void affiche(TUniteLexical ul1);**

**void ULIdentif(string lexeme);**

**void uniteSuivante();**

**bool estBlanc(char c);**

**int getIndexIdentif(string stringToFind, vector<string> stringArray);**

**int getIndex(string stringToFind, vector<string> stringArray);**

**int estMotcle(string lexeme);**

**int estMotreserve(string lexeme);**

**char lc();**

**protected:**

**private:**

**char c=' ';string Str;int test=0;**

**vector<int> tokens;**

**//table des mot-clés**

**vector<string> motsCles={"sinon","alors","si","tantque","faire","entier","retour","struct"};**

**//table des mot-résérvés**

**vector<string> motsReserves={"lire","ecrire","main"};**

**//table des operateurs binaires**

**vector<string> binOp={"+","-","\*","/"};**

**//table des comparateurs binaires**

**vector<string> binComp={"<",">","<=",">=","==","!="};**

**// liste des identifiants**

**vector<string> identifiants;**

**};**

**#endif // LEXICAL\_H**

* **Code de la méthode uniteSuivante() :**

**void Lexical::affiche(TUniteLexical ul1)**

**{**

**char car;**

**for(int i=0;i<ul1.UL.size();i++){car=toupper(ul1.UL[i]);cout<<car;}**

**cout<<"\t\t"<<ul1.attribut<<endl;**

**}**

**void Lexical::uniteSuivante()**

**{**

**string lexeme="";TUniteLexical ul;**

**//if(c=='/0')**

**while(estBlanc(c)) {c=lc();test++;}**

**switch(c){**

**case '+': {**

**c=lc();test++;tokens.push\_back(BIN\_OP);**

**ul.UL="BIN\_OP";ul.attribut=getIndex("+",binOp);cout<<"+"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}case '.': {**

**c=lc();test++;tokens.push\_back(PT);**

**ul.UL="PT";ul.attribut=-1;cout<<"."<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '-':{**

**c=lc();test++;tokens.push\_back(BIN\_OP);**

**ul.UL="BIN\_OP";ul.attribut=getIndex("-",binOp);cout<<"-"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '<':{**

**c=lc();test++;**

**if(c=='='){**

**c=lc();test++;tokens.push\_back(BIN\_COMP);**

**ul.UL="BIN\_COMP";ul.attribut=getIndex("<=",binComp);cout<<"<="<<"\t";**

**}**

**else{**

**tokens.push\_back(BIN\_COMP);ul.UL="BIN\_COMP";ul.attribut=getIndex("<",binComp);cout<<"<"<<"\t";**

**}**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '>':{**

**c=lc();test++;**

**if(c=='='){**

**c=lc();test++;**

**tokens.push\_back(BIN\_COMP);ul.UL="BIN\_COMP";ul.attribut=getIndex(">=",binComp);cout<<">="<<"\t";**

**}**

**else{**

**tokens.push\_back(BIN\_COMP);ul.UL="BIN\_COMP";ul.attribut=getIndex(">",binComp);cout<<">"<<"\t";**

**}**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '\*': {**

**c=lc();test++;tokens.push\_back(BIN\_OP);**

**ul.UL="BIN\_OP";ul.attribut=getIndex("\*",binOp);cout<<"\*"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '&': {**

**c=lc();test++;tokens.push\_back(AND);**

**ul.UL="AND";ul.attribut=-1;cout<<"&"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '|': {**

**c=lc();test++;tokens.push\_back(OR);**

**ul.UL="OR";ul.attribut=-1;cout<<"|"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '/': {**

**c=lc();test++;tokens.push\_back(BIN\_OP);**

**ul.UL="BIN\_OP";ul.attribut=getIndex("/",binOp);cout<<"/"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '=':{**

**c=lc();test++;**

**if(c=='='){**

**c=lc();test++;tokens.push\_back(BIN\_COMP);**

**ul.UL="BIN\_COMP";ul.attribut=getIndex("==",binComp);cout<<"=="<<"\t";**

**}**

**else{**

**tokens.push\_back(AFF);ul.UL="AFF";ul.attribut=-1;cout<<"="<<"\t";**

**}**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '!':{**

**c=lc();test++;**

**if(c=='='){**

**c=lc();test++;tokens.push\_back(BIN\_COMP);**

**ul.UL="BIN\_COMP";ul.attribut=getIndex("=!",binComp);cout<<"!="<<"\t";**

**}**

**else{**

**tokens.push\_back(NON);ul.UL="NON";ul.attribut=-1;cout<<"!"<<"\t";**

**}**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '[':{**

**c=lc();test++;tokens.push\_back(CROCH\_OUV);**

**ul.UL="CROCH\_OUV";ul.attribut=-1;cout<<"["<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case ']':{**

**c=lc();test++;tokens.push\_back(CROCH\_FER);**

**ul.UL="CROCH\_FER";ul.attribut=-1;cout<<"]"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '(':{**

**c=lc();test++;tokens.push\_back(PAR\_OUV);**

**ul.UL="PAR\_OUV";ul.attribut=-1;cout<<"("<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case ')':{**

**c=lc();test++;tokens.push\_back(PAR\_FER);**

**ul.UL="PAR\_FER";ul.attribut=-1;cout<<")"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case ',':{**

**c=lc();test++;tokens.push\_back(VERG);**

**ul.UL="VERG";ul.attribut=-1;cout<<","<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '{':{**

**c=lc();test++;tokens.push\_back(ACC\_OUV);**

**ul.UL="ACC\_OUV";ul.attribut=-1;cout<<"{"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '}':{**

**c=lc();test++;tokens.push\_back(ACC\_FER);**

**ul.UL="ACC\_FER";ul.attribut=-1;cout<<"}"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case ';':{**

**c=lc();test++;tokens.push\_back(PT\_VER);**

**ul.UL="PT\_VER";ul.attribut=-1;cout<<";"<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**case '#':{**

**c=lc();test++;tokens.push\_back(COMMENT);**

**ul.UL="COMMENT";ul.attribut=-1;cout<<'#'<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**default:**

**{**

**if(isdigit(c)){**

**while(isdigit(c)){**

**lexeme += c; //Aj**

**c=lc();test++;**

**}**

**tokens.push\_back(NOMBRE);ul.UL="NOMBRE";ul.attribut=stoi(lexeme);cout<<lexeme<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**else if(isalpha(c)){**

**while(isalpha(c)||isdigit(c)){**

**lexeme += c; //Aj**

**c=lc();test++;**

**}**

**int pos = estMotcle(lexeme);**

**int pos1 = estMotreserve(lexeme);**

**if(pos!=-1){**

**tokens.push\_back(pos);ul.UL=motsCles[pos];ul.attribut=pos;cout<<lexeme<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**if(pos1!=-1){**

**tokens.push\_back(pos1);ul.UL=motsReserves[pos1];ul.attribut=pos1;cout<<lexeme<<"\t";**

**affiche(ul);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**ULIdentif(lexeme);if(c=='\0'){fin();exit(0);}else uniteSuivante();**

**}**

**else{**

**c=lc();test++;tokens.push\_back(ERROR);**

**ul.UL="ERROR";ul.attribut=2;cout<<lexeme<<"\t";**

**affiche(ul);if(c=='\0')exit(0);uniteSuivante();**

**}**

**}}**

**}**

## Programme de test de lexical et son résultat

## Programme de test :

## entier e;

## main()

## {

## e = 153;

## }

## Résultat :

## 

# CHAPITRE 3: ANALYSE SYNTAXIQUE

## Rendre la grammaire LL(1)

Dans cette partie on rendra la grammaire du language C-Pascal LL( 1 ) en élevant toute éventuelle récursivité à gauche, non factorisation ou ambiguïté, dans le but de la rendre analysable par la méthode descendante récursive. Pour ce, on vérifiera ces conditions pour chaque production du grammaire, en évoquant toutes les justifications possibles .

La grammaire du langage C -Pascal avec l’amélioration proposée est la suivante :

1. <Programme> : : = <liste-declarations>< liste-fonctions>|< liste-fonctions>
2. <liste-declarations> : : = <declaration> ;<liste-declarations’>
3. <liste-declarations’>: : = <declaration> ;<liste-declarations’>  | **epsilon**
4. <liste-declarations-sp> : : = <declaration-sp> ;<liste-declarations-sp’>
5. <liste-declarations-sp’> : : = <declaration-sp> ;<liste-declarations-sp’>  | **epsilon**
6. <declaration> : : = <declarateur-cp><declaration’>
7. <declaration’> : : = ,< declarateur-cp><declaration’>**|epsilon**
8. <declaration-sp> : : = <declarateur-sp>< declaration-sp’>
9. <declaration-sp’> : : = ,< declarateur-sp><declaration-sp’>**|epsilon**
10. <declarateur-sp> : : = **entier**<variable>
11. <declarateur-cp> : : = **entier** <declarateur-cp’> | **struct** <variablestruct>**{**<liste-declarations-sp>}
12. <declarateur-cp’> : : = <variabletab>[<nombre>] | < variable>
13. < liste-fonctions> : : = <liste-fonctions’> **main()**<fonction’’>
14. < liste-fonctions’> : : = < fonction> <liste-fonctions’> | **epsilon**
15. <fonction> : : = <identificateur>(<liste-parms>)<fonction’>
16. <fonction’> : : = < liste-declarations-sp><bloc> | <bloc>
17. <fonction’’> : : = < liste-declarations-sp><blocmain> | <blocmain>
18. <liste-parms> : : = <parm><liste-parms’>
19. <liste-parms’> : : = ,<parm><liste-parms’> | **epsilon**
20. <parm> : : = **entier** <variable>
21. <bloc> : : = {<liste-instructions>}
22. <blocmain> : : = {<liste-instructions>}
23. <blocfaire> : : = {<bloc’>
24. <blocalors> : : = {<bloc’>

<bloc’> : : = <liste-instructions>} | }

1. <liste-instructions> : : = <instruction ><liste-instructions’>
2. <liste-instructions’>: : = <instruction ><liste-instructions’> | **epsilon**
3. <instruction> : : = < iteration>|<selection>| <saut> ; | <affectation> ;|<appel> ;
4. <iteration> : : = **tantque** <condition> **faire** <blocfaire>
5. <selection> : : = **si** <condition> **alors** <blocalors> <selection’>
6. <selection’> : : = **sinon** <blocfaire> | **epsilon**
7. <saut> : : = **retour** <expression>
8. <affectation> : : = <variable>=<affectation’> | <variablestruct>.<variable>=<affectation’>|<variabletab>[<nombre>]=< affectation’>

<affectation ’> : : = <expression> | **lire** ( )|<appel>

<affectation ’’> : : = <expression> | **lire** ( )|<appel>

1. <affectation ’> : : = <expression> | **lire** ( )|<appel>
2. <appel> : : = <identificateur> ( <liste-expressions>) | **ecrire** (<expression>)
3. <liste-expressions> : : = <expression><liste-expressions’>
4. <liste-expressions’>: : = ,< expression><liste-expressions’> | **epsilon**
5. <condition> : : = !(<expression>) | <expression><condition’>
6. <condition’> : : = <binary-rel><expression> | <binary-comp><expression>
7. <expression> : : = <terme><expression’>

<expression> : : = <expression>+<terme>|<expression>-<terme>­|<terme>

<terme>  : : = <terme> \*<facteur>|<terme> /<facteur>

|<facteur>

<facteur> : : = <variable> | <nombre> |<variablestruct>.<variable>|<variabletab>[<nombre>]­|(<expression>

1. <expression’> : : = <binary-op><expression> | **epsilon**
2. <terme>  : : = <variable> | <nombre> |<variablestruct>.<variable>|<variabletab>[<nombre>]
3. <variable>  : : = < lettre><variable’>
4. <variablestruct>  : : = < lettre><variable’>
5. <variabletab>  : : = < lettre><variable’>
6. <variable’>  : : = < variable’’><variable’> | **epsilon**
7. <variable’’>  : : = < lettre>| <nombre>
8. <nombre>  : : = <constante><nombre’>
9. <nombre’>  : : = <constante><nombre’> | **epsilon**
10. <identificateur>  : : = < lettre>< identificateur’>
11. < identificateur’>  : : = < variable’’>< identificateur’> | **epsilon**
12. <lettre>  : : = A|B|C|...|Z|a|b|c|...­|z
13. <constante>  : : = 0|1|…­|9
14. <binary-op> : : = +|-|\*|/
15. <binary-rel> : : = **&** | |
16. <binary-comp> : : = <|>|>=|<=|==|!=

## Les méthodes de l’analyseur syntaxique

### Fichier analyseur\_syntaxique.h

**#ifndef ANALYSEUR\_SYNTAXIQUE\_H**

**#define ANALYSEUR\_SYNTAXIQUE\_H**

**#include ’’Lexical.h"**

**#include <string>**

**#include <vector>**

**#include <fstream>**

**using namespace std;**

**class analyseur\_syntaxique**

**{**

**public:**

**analyseur\_syntaxique(string s);**

**// R1**

**void programme();**

**// R2**

**void listeDeclarations();**

**void listeDeclarationsPrime();**

**void listeDeclarationsSimple();**

**void listeDeclarationsSimplePrime();**

**// R3**

**void declaration();**

**void declarationPrime();**

**void declarationSimple();**

**void declarationSimplePrime();**

**void declarateurSimple();**

**void declarateurCompose();**

**void declarateurComposePrime();**

**// R4**

**void listeFonctions();**

**void listeFonctionsPrime();**

**// R5**

**void fonction();**

**void fonctionPrime();**

**void fonctionSecond();**

**// R8**

**void listeParms();**

**void listeParmsPrime();**

**// R9**

**void parm();**

**//10**

**void bloc();**

**void blocmain();**

**void blocfaire();**

**void blocalors();**

**void blocPrime();**

**// R11**

**void listeInstructions();**

**void listeInstructionsPrime();**

**// R12**

**void instruction();**

**// R13**

**void iteration();**

**// R14**

**void selection();**

**void selectionPrime();**

**// R15**

**void saut();**

**// R16**

**void affectation();**

**void affectationPrime();**

**// R17**

**// R18**

**void appel();**

**// R19**

**void variable();**

**void variablePrime();**

**void variableSecond();**

**void variablestruct();**

**void variabletab();**

**// R20**

**void expression();**

**void expressionPrime();**

**// R21**

**void listeExpressions();**

**void listeExpressionsPrime();**

**void terme();**

**// R22;**

**void condition();**

**void conditionPrime();**

**// R23**

**void nombre();**

**void nombrePrime();**

**void identificateur();**

**void identificateurPrime();**

**void binaryOp();**

**// R24**

**void binaryRel();**

**// R25**

**void binaryComp();**

**void finsyntax();**

**void balise\_ouvrante(std::string regle);**

**void balise\_fermante(std::string regle);**

**void balise\_unique(std:: string regle);**

**void arbreDerivation();**

**protected:**

**private:**

**char c=' ';string Str;int test=0;**

**int motCourant;vector<int> tokens;**

**int parcourir;int nbErreur;**

**ofstream monFlux{"ProgrammeTest.txt"};**

**ofstream fluxXML{"ArbreDerivation.txt"};**

**vector<string> identifiants;**

**vector<string> identificateurs;**

**vector<string> Variablestruct;**

**vector<string> Variabletab;**

**vector<string> variables;**

**//table des mot-clés**

**vector<string> motsCles={"sinon","alors","si","tantque","faire","entier","retour","struct"};**

**//table des mot-résérvés**

**vector<string> motsReserves={"lire","ecrire","main"};**

**//table des operateurs binaires**

**vector<string> binOp={"+","-","\*","/"};**

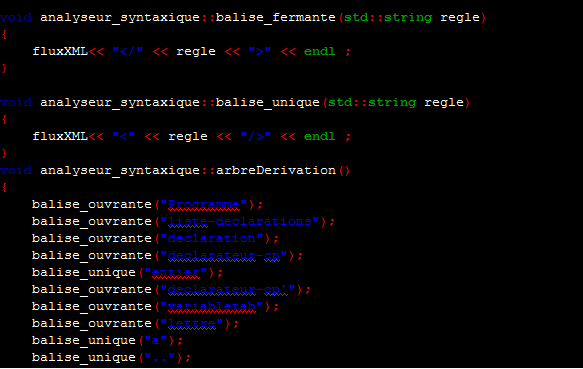
**//table des comparateurs binaires**

**vector<string> binComp={"<",">","<=",">=","==","!="};**

**};**

**#endif // ANALYSEUR\_SYNTAXIQUE\_H**

## L’arbre syntaxique





## Programme de test de syntaxique et son résultat

## 

## Programme de test :

## entier e[10,struct var{entier k,entier m;}

## somme(entier a,entier b)

## entier d[10];

## {

## d=a+b;retour d;

## }

## main(

## {

## e[0]= 153

## var.k=lire);

## ecrire(e[0]);

## Résultat :

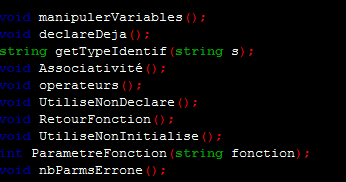
**CHAPITRE 4: ANALYSE SEMANTIQUE**

L’analyse sémantique est la phase intervenant après l’analyse syntaxique et avant la génération de code. Elle effectue les vérifications nécessaires à la sémantique du langage de programmation considéré, ajoute des informations à l’arbre syntaxique abstrait et construit la table des symboles.

Les vérifications sémantiques que nous avons réalisées sont les suivantes :

* Vérification si un identificateur utilisé est déjà déclaré ou non, si l’identificateur n’est pas déclaré, une erreur de compilation sera survenue.
* Vérification des types des identificateurs utilisés, lors de l’appel d’une fonction, d’un tableau ou l’utilisation d’une variable simple (exemples : affectation, opérations…). Si le type d’identificateur utilisé n’est pas compatible, une erreur de compilation sera survenue.
* Pour une fonction appelée, on vérifie si les arguments passés en paramètres sont compatibles, en nombre lors de sa déclaration. Une erreur de compilation sera survenue si l’un de ces critères n’est pas vérifié.
* Vérification de l’unicité des identificateurs utilisés.
* Vérification de type de retour d’une fonction.

## Les méthodes de l’analyseur semantique



## Programme de test de sémantique et son résultat

## 

## Programme de test :

## entier e[10],entier j,struct var{entier k[2],entier m;};

## struct bar{entier j,entier m;};

## entier bar;

## somme(entier a[10],entier b)

## entier d[10];

## {

## d=a+somme;

## b=somme(1,2,3);

## retour d;

## }

## main()

## entier somme,struct var{entier i,entier p;};

## {

## e[0]= 153;

## var=lire();

## ecrire(c);

## }

## Résultat :

## 

**CHAPITRE 5 : BILAN DU PROJET**

1. **Les apports du projet**

Le principal apport de ce projet a été de nous faire mieux comprendre la logique suivant laquelle fonctionne un compilateur. Nous avons donc pu toucher du doigt les différentes étapes de la compilation, les modéliser et ensuite les programmer.

Aussi ce projet, nous a permis de concrétiser et mettre en pratique les notions

théoriques acquises en cours .

Suivant l’aspect du travail en groupe, ce projet fut un bel exercice d’intégration de parties de codes produites séparément. En effet chacun de nous a sa manière de coder et cela peut entraîner très souvent des problèmes à l’exécution du code.

## Les difficultés rencontrées

Comme difficultés, nous avons quelques fois eu du mal à implémenter les cours théoriques sous forme de code, surtout au niveau de la partie sémantique. Il fallait identifier les règles sémantiques les plus importantes et les implémenter de façon à ne pas générer des erreurs ou créer des conflits. Aussi, l’intégration des parties codées séparément fut aussi source de difficultés pour nous.

## Perspectives

▶ Optimiser le code afin qu’il soit plus performant .

▶ Augmenter les règles sémantiques afin que le test soit plus rigoureux .

▶ Intégrer les notions apprises dans le but de créer un langage répondant à un certain

nombre de requêtes données .

▶ Générer code objet .

## Conclusion

Ce projet a été dans toute sa généralité enrichissante. Nous avons eu à exécuter les premières étapes d’une compilation à savoir l’analyse lexicale qui nous a permis d’identifier l’alphabet du langage, ensuite former les différentes unités lexicales

qui pourraient être formés et enfin les reconnaître. Aussi l’analyse syntaxique nous a permis de reformuler la grammaire du langage et de la rendre LL( 1 ) après avoir levé l’ambigüité, la récursivité à gauche et la non factorisation de ces grammaires prés-établis. L’analyse sémantique nous a permis d’établir le sens donné aux différentes phrases du programme écrit. Chaque partie a nécessité une phase d’appréhension des concepts théoriques et de conception et enfin une

phase d’implémentation. En définitive nous percevons mieux l’importance du rôle du

compilateur dans l’exécution d’un programme écrit.