



المدرسة الحسنية للأشغال العمومية
ECOLE HASSANIA DES TRAVAUX PUBLICS

RAPPORT DU PROJET DE COMPILATION

08-JAN-2019

Développement d'un compilateur en C++

Réalisé par:

BOUCHANE Amine

CHAMI Rachid

EL AOUAD Ilyas

EL MOHAMMADI Sohaib

Encadré par:

Monsieur SOUHAIL EL

GHAZI

Contents

0.1	Introduction	7
1	Partie lexicale	9
1.1	Grammaire initiale	9
1.2	Grammaire améliorée	10
1.3	L'alphabet du langage	11
1.4	Exemple de programme	12
1.5	Tables lexicale	13
1.6	Automate du langage	15
1.6.1	Erreurs	20
1.7	Structures de données utilisées	21
1.7.1	Enumération des mots du langage	21
1.7.2	Structures de données	21
1.7.3	Fonction de Hashage	23
1.8	Exemple d'exécution du lexicale	24
1.8.1	Affichage erreurs	25
2	Partie Syntaxique	27
2.1	Grammaire améliorée	27
2.2	Enlever la récursivité à gauche et factoriser la grammaire	29
2.3	Table des Premiers et Suivants pour cette grammaire	33
2.4	Résolution des ambiguïtés	34
2.5	Améliorations	35
2.5.1	Les operations d'affectation	35
2.5.2	Les parametres d'une fonction	36
2.5.3	Le problème de Programme et Liste instructions	36
2.5.4	Problème déclaration et instruction	37
2.5.5	Manipulation des chaines	37
2.6	La grammaire finale	39
2.7	Exemples d'erreurs levées par le syntaxique:	42

3	Partie Sémantique	45
3.1	Introduction	45
3.2	La grammaire du langage avec des actions sémantiques	46
3.3	Définition des actions sémantiques	47
3.3.1	Vérification des redéfinitions	47
3.3.2	Vérification des définitions	47
3.3.3	Vérification du caractère simple ou composé d'une variable . .	48
3.3.4	Vérification des types	48
3.3.5	Vérifications des index nulles	48
3.3.6	Vérification des manipulations de chaînes	49
3.4	Structures utilisées	50
3.4.1	Table des symboles	50
3.4.2	La structure variable	50
3.5	Exemples d'erreurs levées par le sémantique.	51

List of Figures

1.1	Alphabet du langage	11
1.2	Table des unités lexicales	13
1.3	Table des unités lexicales	14
1.4	Automate du langage	15
1.5	Automate du langage	16
1.6	Automate du langage	17
1.7	Automate du langage	18
1.8	Automate du langage	19
1.9	Automate du langage	20
1.10	input du lexical	24
1.11	Output du lexical	25
1.12	Erreur : Introduction d'un mot qui n'appartient pas au langage . . .	26
1.13	Erreur : Chaine de caractères qui ne finit pas par "	26
2.1	Enlever la récursivité à gauche et factorisation	29
2.2	Enlever la récursivité à gauche et factorisation	30
2.3	Enlever la récursivité à gauche et factorisation	31
2.4	Enlever la récursivité à gauche et factorisation	32
2.5	Table des Premiers et Suivants pour cette grammaire	33
2.6	Table des Premiers et Suivants pour cette grammaire	34
2.7	Texte source.	42
2.8	Erreur : oublie du point virgule.	42
2.9	Texte source.	42
2.10	Erreur : oublie de fermer un crochet ouvrant.	42
2.11	Texte source.	43
2.12	Erreur : opération non permise.	43
2.13	Texte source.	43
2.14	Erreur : oublie de donner un type lors de la déclaration d'une fonction. .	43
2.15	Texte source.	43
2.16	Erreur : oublie de l'identifiant.	43

3.1	Grammaire avec actions sémantiques.	46
3.2	Grammaire avec actions sémantiques	46
3.3	Erreur index négative.	48
3.4	Erreur index négative.	49
3.5	Erreur de redéfinition.	51
3.6	Erreur de redéfinition.	51
3.7	Erreur des types lors d'une affectation.	51
3.8	Erreur des types lors d'une affectation.	52
3.9	Erreur des types lors d'une affectation.	52
3.10	Erreur des types lors d'une affectation.	52
3.11	Erreur des types lors d'une affectation.	52
3.12	Erreur des types lors d'une affectation.	52
3.13	Erreur d'indice d'un tableau.	52
3.14	Erreur d'indice d'un tableau.	52
3.15	Erreur de définition.	52
3.16	Erreur de définition.	52

0.1 Introduction

Ce document a pour but décrire le déroulement du projet proposé par M. EL GHAZI au cours du module Théorie de la Compilation. C'est le résultat du travail qui nous a permis d'implémenter un compilateur depuis le langage C- et lui proposer un analyseur lexical, syntaxique et finalement sémantique.

Chapter 1

Partie l  xicale

1.1 Grammaire initiale

```
<programme> : <liste-declarations> <liste-fonctions>
<liste-declarations> : <liste-declarations> <declarations> | epsilon
<declarations> : int <liste-declarateurs>
<liste-declarateurs> : <list-declarateurs> , <declarateur>
                    | <declarateur>
<Declarateur> : identificateur | identificateur [ constante ]
<liste-fonctions> : <liste-fonctions> <fonction> | epsilon
<fonction> : <type> identificateur ( <liste-parms> )
            { <liste-declarations> <liste-instructions> }
            | extern <type> identificateur ( <liste-parms> );
<type> : void | int
<liste-parms> : <liste-parms> , <parm> | epsilon
<parm> : int identificateur
<liste-instructions> : <liste-instructions> <instruction> | epsilon
<instruction> : <iteration> | <selection> | <saut>
              | <affectation>; | <bloc> <appel>
<iteration> : for (<affectation> ; <condition>
                ; <affectation>) <instruction>
              | while (<condition>) <instruction>
              | if ( <condition> ) <instruction>
              | if ( <condition> ) <instruction> else <instruction>
<saut> : return ; | return <expression> ;
<affectation> : <variable> = <expression>
<bloc> : { <liste-instructions> }
```

```

<appel> : identificateur (<liste-expressions>);
<variable> : identificateur | identificateur [ <expression> ]
<expression> : ( <expression> )
               | <expression> <binary-op> <expression>
               | - <expression>
               | <variable>
               | identificateur ( <liste-expressions> )
<liste-expressions> : <liste-expressions> , <expression> | epsilon
<condition> : ! ( <condition> )
               | <condition> <binary-rel> <condition>
               | ( <condition> )
               | <expression> <binary-comp> <expression>
<binary-op> : + | - | * | / | << | >> | & | ||
<binary-rel> : && | ||
<binary-comp> : < | > | >= | <= | == | !=

```

1.2 Grammaire améliorée

```

<programme> : <liste-declarations> <liste-fonctions>
<liste-declarations> : <liste-declarations> <declarations> | epsilon
<declarations> : int <liste-declarateurs>
                 | chaine <liste-declarateurs>
<liste-declarateurs> : <list-declarateurs> , <declarateur>
                     | <declarateur>
<Declarateur> : identificateur | identificateur [ constante ]
<liste-fonctions> : <liste-fonctions> <fonction> | epsilon
<fonction> : <type> identificateur ( <liste-parms> )
             { <liste-declarations> <liste-instructions> }
             | extern <type> identificateur ( <liste-parms> );
<type> : void | int | chaine
<liste-parms> : <liste-parms> , <parm> | epsilon
<parm> : int identificateur
<liste-instructions> : <liste-instructions> <instruction> | epsilon
<instruction> : <iteration> | <selection>
               | <saut> | <affectation>; | <bloc> <appel>
<iteration> : for (<affectation> ; <condition>
                 ; <affectation>) <instruction>
               | while (<condition>) <instruction>
               | if ( <condition> ) <instruction>

```

```

        | if ( <condition> ) <instruction> else <instruction>
<saut> : return ; | return <expression> ;
<affectation> : <variable> = <expression>
<bloc> : { <liste-instructions> }
<appel> : identificateur (<liste-expressions>);
<variable> : identificateur | identificateur [ <expression> ]
<expression> : ( <expression> )
               | <expression> <binary-op> <expression>
               | - <expression>
               | <variable>
               | identificateur ( <liste-expressions> )
<liste-expressions> : <liste-expressions> , <expression> | epsilon
<condition> : ! ( <condition> )
               | <condition> <binary-rel> <condition>
               | ( <condition> )
               | <expression> <binary-comp> <expression>
<binary-op> : + | - | * | / | << | >> | & | ||
<binary-rel> : && | ||
<binary-comp> : < | > | >= | <= | == | !=
<lire> : lire (<liste-declarateur>);
<ecrire> : écrire (<liste-declarateur>);
<chaine> : "<DefChaine>"
<DefChaine> : <lettre> <DefChaine>
               | <chiffre> <DefChaine> | epsilon
<entier> : <chiffre> <entier> | epsilon
<mot> : <letter> <mot> | epsilon
<chiffre> : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<lettre> : a | ... | z | A | ... | Z

```

1.3 L'alphabet du langage

II. L'alphabet du langage:

$\Sigma = \{ 0 \dots 9, ;, ,, , \{, \}, =, !, <, >, \&, +, -, *, /, (,), [,], a \dots z, A \dots Z, |, \& \}$

Figure 1.1: Alphabet du langage

1.4 Exemple de programme

```
chaine maChaine1;    /* Variable globale de type chaine */
chaine maChaine2;

Int MaPremiereFonction( int a , int b ) {
    int resultat;
    resultat = a + b;
    return resultat;
}

void MaDeuxiemeFonction( int nombre ){
    int i;
    for( i=0; i<=nombre; i=i+1 )
        ecrire(i);
}
int MaTroisiemeFonction(int a, int b){
    if(a>b)
        return a;
    else
        return b;
}
int main() {
    int a, b;
    ecrire("donnez le premier nombre");
    lire(a);
    ecrire("donnez le deuxieme nombre");
    lire(b);
    if(MaTroisiemeFonction(a,b) == a){
        ecrire(maChaine1);
    }
    else
        ecrire(maChaine2);
    return 0;
}
```

1.5 Tables lexicales

Unité lexicale	Lexème	Attribut	Model
PLUS	+	--	+
MOINS	-	--	-
MULT	*	--	*
DIV	/	--	/
INF	<	--	<
SUP	>	--	>
	&	--	&
BARRE		--	
NON	!	--	!
AFF	=	--	=
SOMICOLON	;	--	;
INFEG	<=	--	<=
SUOEG	>=	--	>=
EGAL	==	--	==
DIFF	!=	--	!=
ET	&&	--	&&
OU		--	
DINF	<<	--	<<
DSUF	>>	--	>>
PAROUV	(--	(
PARFER)	--)
ACCOUV	{	--	{
ACCFER	}	--	}
CROCHOUV	[--	[
CROCHFER]	--]

Figure 1.2: Table des unités lexicales

CROCHFER]	--]
PAROUV	(--	(
PARFER)	--)
INT	1,27...	Valeur de l'entier	Chiffre+
TAB	Tab[9]	l'indice dans la table des symboles	IDENT[Chiffre+]
CHAINE	"Bonjour ", "bon7jo ur "..	Position dans la table des chaines de caractères	" (lettre chiffre)* "
IDENT	IDENT A1...	Positon dans la table des identificateurs	Lettre(lettre chiffre)*
MOTCLE	extern,int,void,chai ne...	Position dans la table des mots clés	lettre+

Figure 1.3: Table des unités lexicales

1.6 Automate du langage

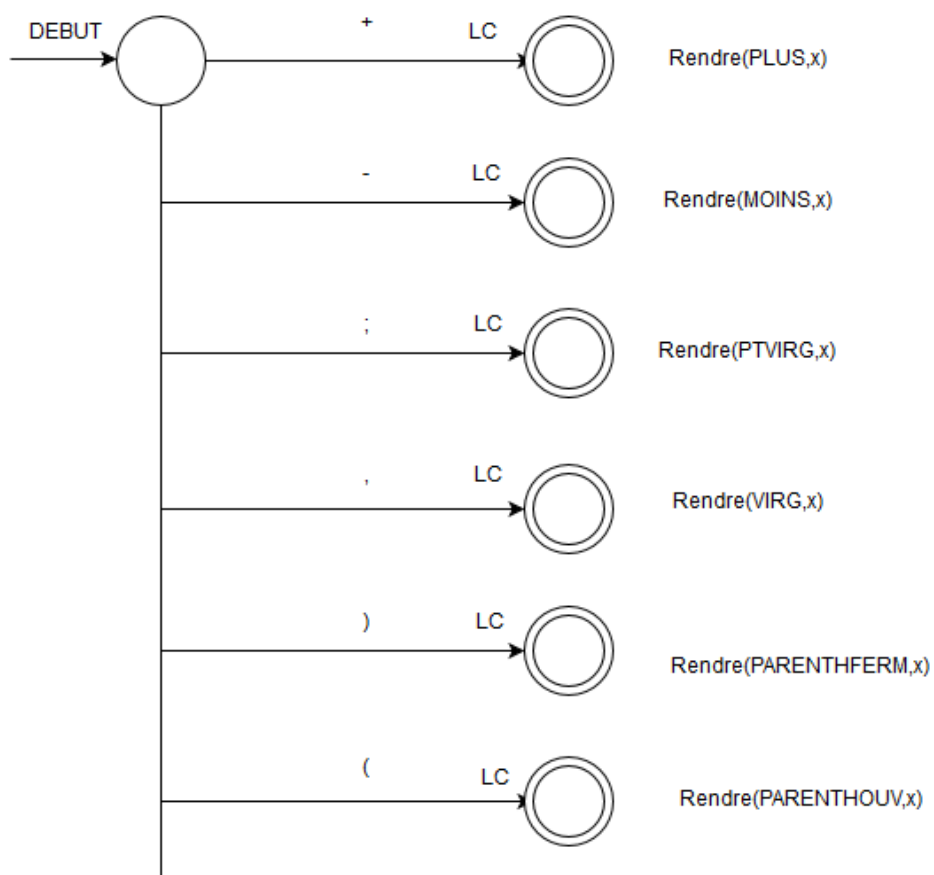


Figure 1.4: Automate du langage

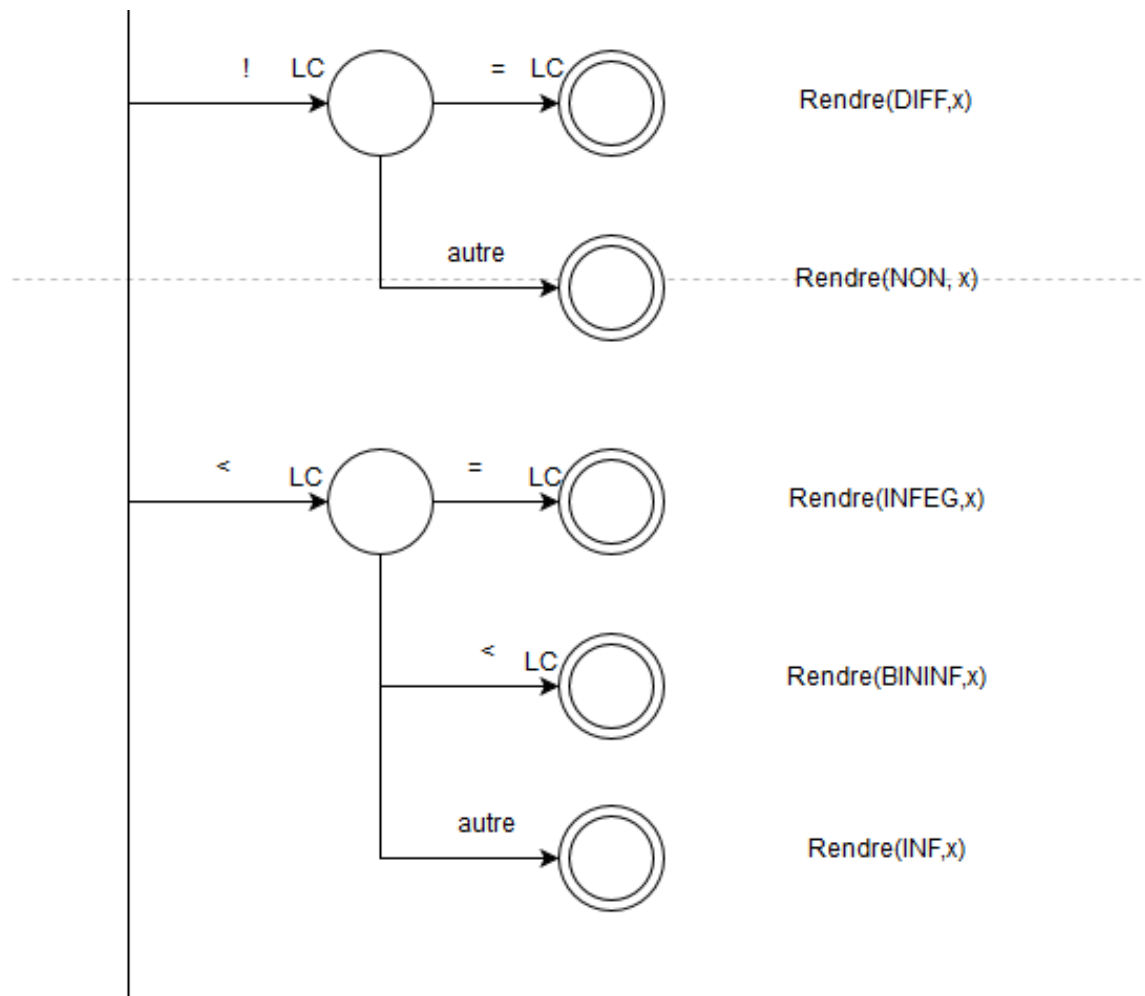


Figure 1.5: Automate du langage

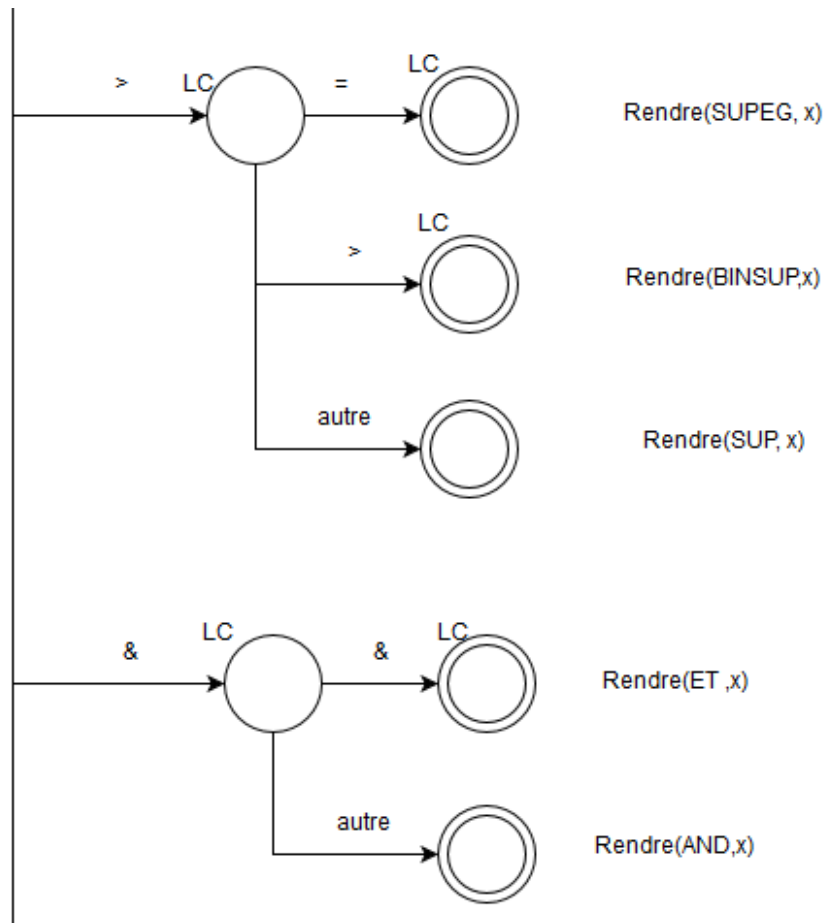


Figure 1.6: Automate du langage

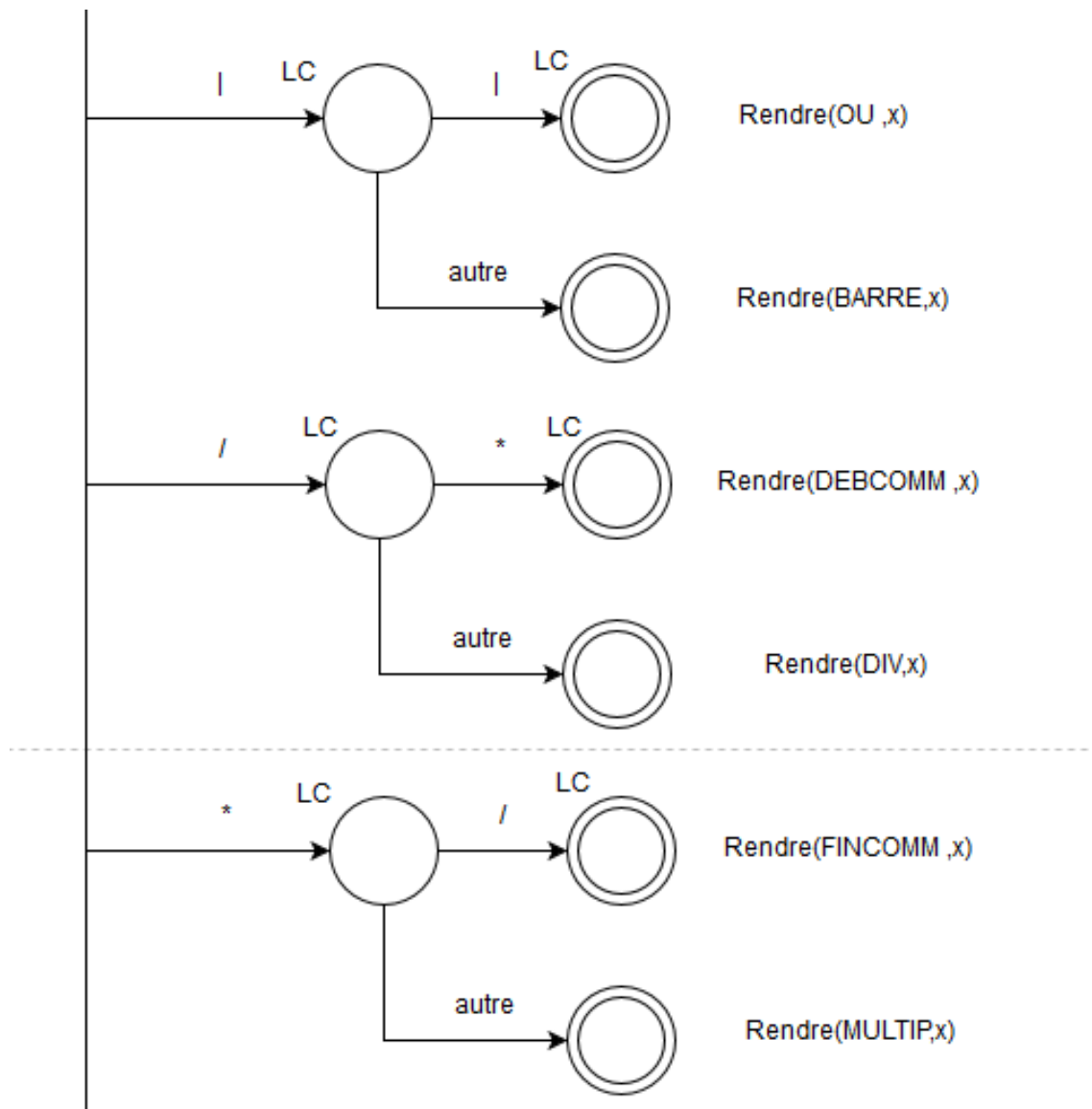


Figure 1.7: Automate du langage

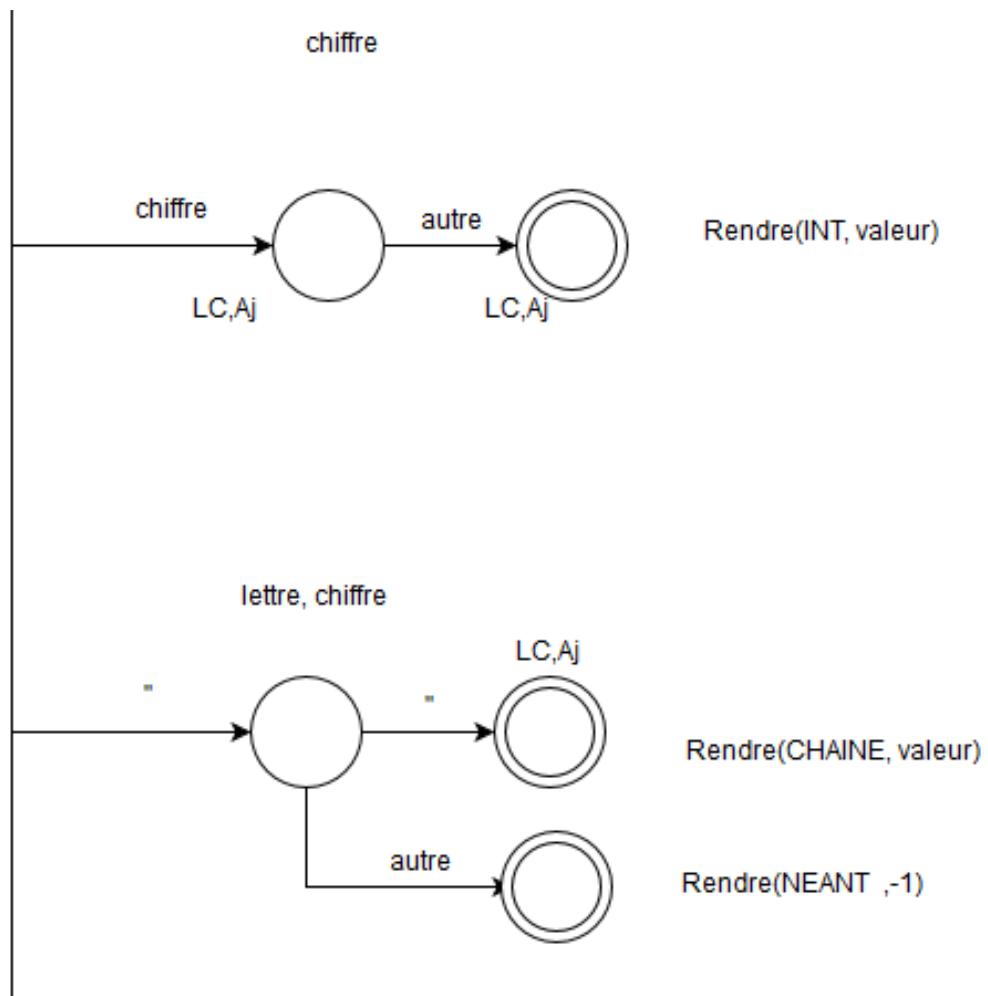


Figure 1.8: Automate du langage

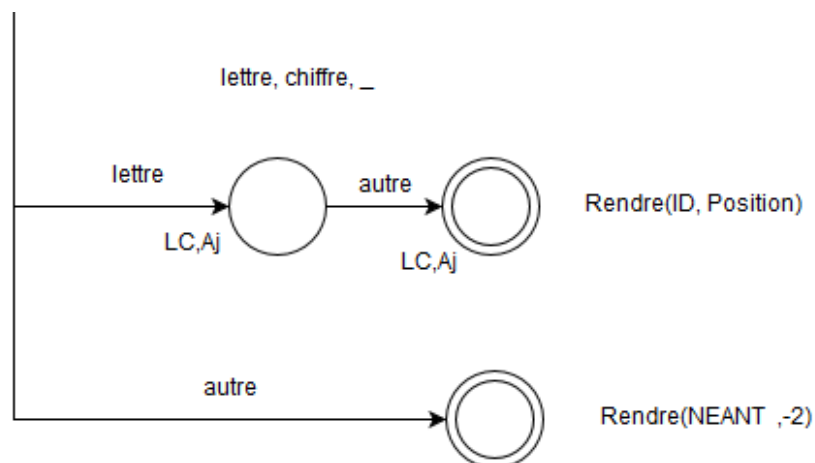


Figure 1.9: Automate du langage

1.6.1 Erreurs

Le lexical est conçu pour lever deux erreurs :

- Lexeme non appartenant au langage;
- Chaîne non terminée.

Les autres erreurs seront traitées dans la partie syntaxique et sémantique.

1.7 Structures de données utilisées

1.7.1 Enumération des mots du langage

```
enum Mots{
    ZERO, INT, CHAINE, ID, VOID, EXTERN, RETURN, FOR, WHILE,
    IF, ELSE, CONSTANTE, ACCOLADELOT, ACCOLADEFM,
    CROCHLOT, CROCHFM, INFEG, DIFF, INF, EGAL, PLUS
    , MOINS, MULTIP, DIV, OU, BARRE, ET, AND, NON, SUPEG, SUP, VIRG,
    PTVIRG, PARENTHOUV, PARENTHFERM, DEBCOMM, FINCOMM, BININF,
    BINSUP, AFFECT, CH, COMMENT, LIRE, ECRIRE, ERR, ERRCH
};
```

C'est l'énumération des unités lexicales utilisée pour distinguer entre les mots du langage et les identifiants etc.

1.7.2 Structures de données

Unité

```
struct unite {
    Mots ul;
    std::pair<int, int> attribut;

    std::string toString(AnalyseLexical* an) {
        // Definiton de la fonction
    }
};
```

On utilise cette structure pour stocker les lexèmes rencontrés lors de la lecture; et pouvoir les récupérer au moment convenable. On utilise un élément ul de Mots pour montrer sur quel lexème on travail et un pair (entier; entier) nommé attribut pour stocker la position:

- d'une chaîne de caractères: le premier entier c'est son hash (position dans la table des chaînes) et le deuxième c'est son index dans la liste chaînée en cas de collision.
- un identifiant ou le premier entier c'est son hash (position dans la table des identifiants) et le deuxième c'est son index dans la liste chaînée en cas de collision.

- un chiffre : ou le premier entier c'est sa valeur; le deuxième c'est, par convention, -1
- un mot clé : premier entier et le deuxième, par convention, égale à -1

Chaine

```
struct chaine {  
    std::string valeur;  
    chaine* suivant;  
};
```

On utilise cette structure pour stocker les chaines de caractères sous forme de liste chaînée (utilisable dans le cas des collisions). Elle contient, un string qui contient sa valeur; et le suivant qui pointe sur l'élément suivant de la liste chaînée.

Identifiant

```
struct identif {  
    std::string identifiant;  
    identif* suivant;  
};
```

C'est la struct des identifiants, Similaire à celle des chaines.

Tableaux de données

```
std::vector<identif*> tableauIdentif = std::vector<identif*> (100, nullptr);  
std::vector<chaine*> Chaines = std::vector<chaine*> (100, nullptr);
```

Le choix des vecteurs dynamiques est fait dans le but de mieux gérer la mémoire.

Map des mots Clés

```
std::map<std::string, Mots> motCles; // Map des mots clés
```

décider si un lexeme est mot clé ou non. La map utilisée prend comme indice un string et contient l'unité lexicale correspondante au mot clé; Par exemple : motCles("if") va retourner la position de 'if' dans l'énumérations. Et motCles('goto') va retourner zéro, donc ce n'est pas un mot clé. Ps : L'énumération des mots du langage contient un ZERO au début pour différencier entre l'existence d'un mot clé ou non.

Unité suivante

```
unite unite_uniteSuivante();
```

Fonction qui retourne la structure de l'unité dans laquelle on se trouve.

1.7.3 Fonction de Hashage

La fonction de hashage utilisée suit le schéma suivant:

- transforme le mot en miniscule; pour ne pas avoir des différences entre un identifiant en majuscule et autre en miniscule;
- parcourt tous ces caractères et multiplie par un nonce (nombre qui s'incrémente à chaque itération);
- retourne un nombre ≤ 100 (taille du tableau des identifiants).

Le code de la fonction:

```
int AnalyseLexical::hash(string str) {
    int nonce = 1;
    int hash = 0;
    str = stringToLower(str);
    for(std::string::iterator it=str.begin(); it!=str.end(); ++it){
        hash += ((int)*it) * nonce;
        ++nonce;
    }
    return hash % 100;
} // Fin hash(string)
```

1.8 Exemple d'exécution du lexicale

Entrée

```
==> Text a traduire :
int a1;
int a2;
int a3[22];

extern int produit(int x,int y);
extern int add(int x,int y);

void somme(int g, int b, int m){

    int a;
    int b,a;
    int c[100];

    a = b + (c -b);
    a = (b + 3);
    a = 3 * 3;
    a = 3 / 3;
    a = 3 - 3;
    a = 3 << 3;
    a = 3 & 3;
    a = 3 | 3;
    a = 3 >> 3;

    if( a == "rachd") {
        if ( a == b) {

        }
        else{ return c;}
    }

}

extern int divide(int x,int y);
```

Figure 1.10: input du lexical

Sortie

```
==> Analyse lexicale :
(Lexeme : int      Attribut : -1      unite lexicale : 1)

(Lexeme : a1       Attribut : 95      unite lexicale : 3)

(Lexeme : ;        Attribut : -1      unite lexicale : 33)

(Lexeme : int      Attribut : -1      unite lexicale : 1)

(Lexeme : a2       Attribut : 97      unite lexicale : 3)

(Lexeme : ;        Attribut : -1      unite lexicale : 33)

(Lexeme : int      Attribut : -1      unite lexicale : 1)

(Lexeme : a3       Attribut : 99      unite lexicale : 3)

(Lexeme : [        Attribut : -1      unite lexicale : 15)

(Lexeme : 22       Attribut : -1      unite lexicale : 12)

(Lexeme : ]        Attribut : -1      unite lexicale : 16)

(Lexeme : ;        Attribut : -1      unite lexicale : 33)

(Lexeme : extern   Attribut : -1      unite lexicale : 5)

(Lexeme : int      Attribut : -1      unite lexicale : 1)

(Lexeme : produit  Attribut : 0       unite lexicale : 3)
```

Figure 1.11: Output du léxical

1.8.1 Affichage erreurs

```
(Lexeme : [      Attribut : -1      unite lexicale : 15)
(Lexeme : 23      Attribut : -1      unite lexicale : 12)
(Lexeme : ]      Attribut : -1      unite lexicale : 16)
(Lexeme : ;      Attribut : -1      unite lexicale : 33)
(Lexeme : erreur d'alphabet      Attribut : -1      unite lexicale : 45)

==> Probleme dans la partie lexicale.
```

Figure 1.12: Erreur : Introduction d'un mot qui n'appartient pas au langage

```
(Lexeme : [      Attribut : -1      unite lexicale : 15)
(Lexeme : 3      Attribut : -1      unite lexicale : 12)
(Lexeme : ]      Attribut : -1      unite lexicale : 16)
(Lexeme : =      Attribut : -1      unite lexicale : 40)
(Lexeme : erreur chaine non complete      Attribut : -1      unite lexicale : 46)

Erreur Chaine non complete, " non trouve. Ligne : 28
```

Figure 1.13: Erreur : Chaine de caractères qui ne finit pas par ”

Chapter 2

Partie Syntaxique

2.1 Grammaire améliorée

```
<programme> : <liste-declarations> <liste-fonctions>
<liste-declarations> : <liste-declarations> <declarations> | epsilon
<declarations> : int <liste-declarateurs>
                | chaine <liste-declarateurs>
<liste-declarateurs> : <list-declarateurs> , <declarateur>
                | <declarateur>
<Declarateur> : identificateur | identificateur [ constante ]
<liste-fonctions> : <liste-fonctions> <fonction> | epsilon
<fonction> : <type> identificateur ( <liste-parms> )
            { <liste-declarations> <liste-instructions> }
            | extern <type> identificateur ( <liste-parms> );
<type> : void | int | chaine
<liste-parms> : <liste-parms> , <parm> | epsilon
<parm> : int identificateur
<liste-instructions> : <liste-instructions> <instruction> | epsilon
<instruction> : <iteration> | <selection>
              | <saut> | <affectation>; | <bloc> <appel>
<iteration> : for ( <affectation> ; <condition>
                ; <affectation> ) <instruction>
              | while ( <condition> ) <instruction>
              | if ( <condition> ) <instruction>
              | if ( <condition> ) <instruction> else <instruction>
<saut> : return ; | return <expression> ;
```

```

<affectation> : <variable> = <expression>
<bloc> : { <liste-instructions> }
<appel> : identificateur (<liste-expressions>);
<variable> : identificateur | identificateur [ <expression> ]
<expression> : ( <expression> )
               | <expression> <binary-op> <expression>
               | - <expression>
               | <variable>
               | identificateur ( <liste-expressions> )
<liste-expressions> : <liste-expressions> , <expression> | epsilon
<condition> : ! ( <condition> )
               | <condition> <binary-rel> <condition>
               | ( <condition> )
               | <expression> <binary-comp> <expression>
<binary-op> : + | - | * | / | << | >> | & | ||
<binary-rel> : && | ||
<binary-comp> : < | > | >= | <= | == | !=
<lire> : lire (<liste-declarateur>);
<ecrire> : écrire (<liste-declarateur>);
<chaine> : "<DefChaine>"
<DefChaine> : <lettre> <DefChaine>
               | <chiffre> <DefChaine> | epsilon
<entier> : <chiffre> <entier> | epsilon
<mot> : <letter> <mot> | epsilon
<chiffre> : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<lettre> : a | ... | z | A | ... | Z

```

2.2 Enlever la récursivité à gauche et factoriser la grammaire

```

1- <programme> : <liste-declarations> <liste-fonctions>
2- <liste-declarations> : <liste-declarations> <declarations> | epsilon
    Récursivité à Gauche :
        <liste-declarations> : <liste-declaration'>
        <liste-declaration'> : <delcaration> <liste-declaration'> | epsilon
    donc:
2- <liste-declarations> : <delcaration> <liste-declaration> | epsilon
3- <declarations> : int <liste-declarateurs>; | chaîne <liste-declarateurs>;
4- <liste-declarateurs> : <list-declarateurs> , <declarateur> | <declarateur>
    Récursivité à gauche :
        <liste-declarateurs> : <declarateur> <liste-declarateurs'>
        <liste-declarateurs'> : , <declarateur> <liste-declarateurs'> | epsilon
    donc:
4-1 <liste-declarateurs> : <declarateur> <liste-declarateurs'>
4-2 <liste-declarateurs'> : , <declarateur> <liste-declarateurs'> | epsilon
5- <Declarateur> : identificateur | identificateur { constante }
    Factorisation:
        <Declarateur> : identificateur <Declarateur'>
        <Declarateur'> : [ constante ] | epsilon
    Donc:
5-1 <Declarateur> : identificateur <Declarateur'>
5-2 <Declarateur'> : [ constante ] | epsilon
6- <liste-fonctions> : <liste-fonctions> <fonction> | epsilon
    Récursivité à gauche :
        <liste-fonctions> : <fonction> <liste-fonctions> | epsilon
    Donc:
6- <liste-fonctions> : <fonction> <liste-fonctions> | epsilon
7- <fonction> : <type> identificateur ( <parm> <liste-parms> ) { <liste-declarations> <liste-
instructions> }
    | extern <type> identificateur ( <parm> <liste-parms> );

```

Figure 2.1: Enlever la récursivité à gauche et factorisation

```

8- <type> : void | int | chaîne
9- <liste-params> : <liste-params> , <parm> | epsilon
    Recursivite a gauche:
        <liste-params> : , <parm> <liste-params> | epsilon
    Donc:
9-1 <liste-params> : <parm> <liste-params>' | epsilon
9-2 <liste-params'> : , <parm> <liste-params'> | epsilon
10- <parm> : int identificateur | chaîne identificateur
11- <liste-instructions> : <liste-instructions> <instruction> | epsilon
    Recursivite a gauche :
        <liste-instructions> : <instruction> <liste-instructions> | epsilon
    Donc
11- <liste-instructions> : <instruction> <liste-instructions> | epsilon
12- <instruction> : <iteration> | <selection> | <saut> | <affectation>; | <bloc> | <appel>
13- <iteration> : for (<affectation> ; <condition> ; <affectation>) <instruction>
    | while (<condition>) <instruction>
14- <selection> : if (<condition>) <instruction>
    | if (<condition>) <instruction> else <instruction>
    Factorisation:
        <selection> : if (condition) <instruction> <selection'>
        <selection'> : else <instruction> | epsilon
    Donc
14-1 <selection> : if (condition) <instruction> <selection'>
14-2 <selection'> : else <instruction> | epsilon
15- <saut> : return ; | return <expression>;
    Factorisation:
        <saut> : <return> <saut'>
        <saut'> : <expression> ; ;
    Donc:
15-1 <saut> : <return> <saut'>
15-2 <saut'> : <expression> ; ;

```

30
Figure 2.2: Enlever la récursivité à gauche et factorisation

```

16- <affected> : <variable> = <expression>
17- <bloc> : { <liste-instructions> }
18- <appel> : identificateur (<liste-expressions>);
19- <variable> : identificateur | identificateur [ <expression> ]
    Factorisation :
        <variable> : identificateur <variable'>
        <variable'> : [ <expression> ] | epsilon

    Donc:
19-1 <variable> : identificateur <variable'>
19-2 <variable'> : [ <expression> ] | epsilon
20- <expression> : ( <expression> )
    | <expression> <binary-op> <expression>
    | <expression>
    | <variable>
    | identificateur ( <liste-expressions> )

    Recursive a gauche:
        <expression> : ( <expression> ) <expression'>
        | - <expression> <expression'>
        | <variable> <expression'>
        | identificateur ( <liste-expressions> ) <expression'>
        <expression'> : <binary-op> <expression> <expression'> | epsilon

    Donc:
20-2 <expression'> : <binary-op> <expression> <expression'> | epsilon
20-1 <expression> : ( <expression> ) <expression'>
    | - <expression> <expression'>
    | <variable> <expression'>
    | identificateur ( <liste-expressions> ) <expression'>

    Factorisation:
        <expression> : ( <expression> ) <expression'>
        | - <expression> <expression'>
        | identificateur <expression'>
        <expression'> : ( <liste-expressions> ) <expression'> | <variable'> <expression'>

```

Figure 2.3: Enlever la récursivité à gauche et factorisation

20-1 $\langle \text{expression} \rangle : (\langle \text{expression} \rangle) \langle \text{expression}' \rangle$
 $\quad \quad \quad | - \langle \text{expression} \rangle \langle \text{expression}' \rangle$
 $\quad \quad \quad | \text{identificateur} \langle \text{expression}'' \rangle$

20-2 $\langle \text{expression}' \rangle : \langle \text{binary-op} \rangle \langle \text{expression} \rangle \langle \text{expression}' \rangle \mid \text{epsilon}$

20-3 $\langle \text{expression}'' \rangle : (\langle \text{liste-expressions} \rangle) \langle \text{expression}' \rangle \mid \langle \text{variable}' \rangle \langle \text{expression}' \rangle$

~~21- $\langle \text{liste-expressions} \rangle : \langle \text{liste-expressions} \rangle , \langle \text{expression} \rangle \mid \text{epsilon}$~~

Recursive à gauche :

~~$\langle \text{liste-expressions} \rangle : , \langle \text{expression} \rangle \langle \text{liste-expressions} \rangle \mid \text{epsilon}$~~

Donc:

21- $\langle \text{liste-expressions} \rangle : , \langle \text{expression} \rangle \langle \text{liste-expressions} \rangle \mid \text{epsilon}$

~~22- $\langle \text{condition} \rangle : ! (\langle \text{condition} \rangle)$~~

~~——— $| \langle \text{condition} \rangle \langle \text{binary-rel} \rangle \langle \text{condition} \rangle$~~

~~——— $| (\langle \text{condition} \rangle)$~~

~~——— $| \langle \text{expression} \rangle \langle \text{binary-comp} \rangle \langle \text{expression} \rangle$~~

Recursive à gauche :

~~$\langle \text{condition} \rangle : ! (\langle \text{condition} \rangle) \langle \text{condition}' \rangle$~~

~~$| (\langle \text{condition} \rangle) \langle \text{condition}' \rangle$~~

~~$| \langle \text{expression} \rangle \langle \text{binary-comp} \rangle \langle \text{expression} \rangle \langle \text{condition}' \rangle$~~

~~$\langle \text{condition}' \rangle : \langle \text{binary-rel} \rangle \langle \text{condition} \rangle \langle \text{condition}' \rangle \mid \text{epsilon}$~~

Donc:

22-1 $\langle \text{condition} \rangle : ! (\langle \text{condition} \rangle) \langle \text{condition}' \rangle$
 $\quad \quad \quad | (\langle \text{condition} \rangle) \langle \text{condition}' \rangle$
 $\quad \quad \quad | \langle \text{expression} \rangle \langle \text{binary-comp} \rangle \langle \text{expression} \rangle \langle \text{condition}' \rangle$

22-2 $\langle \text{condition}' \rangle : \langle \text{binary-rel} \rangle \langle \text{condition} \rangle \langle \text{condition}' \rangle \mid \text{epsilon}$

23- $\langle \text{binary-op} \rangle : + \mid - \mid * \mid / \mid \ll \mid \gg \mid \& \mid \parallel$

24- $\langle \text{binary-rel} \rangle : \&\& \mid \parallel$

Figure 2.4: Enlever la récursivité à gauche et factorisation

2.3 Table des Premiers et Suivants pour cette grammaire

Regle de Production	Premiers	Suivant	LL1
Programme	-	-	OUI
Liste-declarations	{epsilon, int, chaîne}	{int chaîne, for, while, if, ...}	NON
Declarations	{int}, {chaîne}	{	OUI
Liste-declareurs	-	-	OUI
Liste-declareurs'	{epsilon, ,}	{int, chaîne}	OUI
Declareur	-	-	OUI
Declareur'	{[}	{int, chaîne, ,}	OUI
Liste-fonctions	{epsilon, void, int, chaîne, extern}	{\$}	OUI
Fonction	{int, void, chaîne}, {extern}	-	OUI
Type	{void}, {int}, {chaîne}	-	OUI
Liste-parms	{epsilon, ,}	{)}	OUI
Parm	{int}, {chaîne}	-	OUI
Liste-instructions	{for, while, if, return {, identificateur}	{}	OUI
Instuction	{for}, {while}, {if}, {return}, {identificateur}, {{}	-	OUI
Iteration	{for}, {while}	-	OUI

Figure 2.5: Table des Premiers et Suivants pour cette grammaire

Iteration	{for}, {while}	-	OUI
Selection	-	-	OUI
Selection'	{epsilon, else}	{}}	OUI
Saut	-	-	OUI
Saut'	{;}, {(, -, <u>identificateur</u> }	-	OUI
Affectation	-	-	OUI
Bloc	-	-	OUI
Appel	-	-	OUI
Variable	-	-	OUI
Variable'	{epsilon, []}	{+ - * / << >> & , < > >= <= == !=, && }	OUI
Expression	{(), {-}, { <u>identificateur</u> }	-	OUI
Expression'	{epsilon, +, -, *, /, <<, >>, &, }	{[],), :, &&, , +, -, *, /, <<, >>, &, }	NON
Expression"	{(), {[]}	-	OUI
Liste-expressions	{epsilon, , }	{})	OUI
Condition	{!}, { (), { (-, <u>identificaeur</u> }	-	NON
Condition'	{&&, , epsilon}	{&&, }	NON

Figure 2.6: Table des Premiers et Suivants pour cette grammaire

2.4 Résolution des ambiguïtés

La première ambiguïté se trouve dans liste declarations où on constate que le premier et le suivant de `[liste-declarations]` sont d'intersection non vide. Pour remédier à cela, on suivra une méthode d'éclatement des non terminaux pour trouver la partie commune et la factoriser.

On se retrouve avec la grammaire suivante :

```

1-1 <programme> : int <programme'>
                  | chaine <programme'>
                  | void identificateur (<liste-parms> ) {
<liste-declarations> <liste-instructions> }
<programme>
    
```

```

        | extern <type> identificateur (<liste-params> );
<programme>
        | epsilon

1-2 <programme'> : identificateur <programme''>
1-3 <programme''> : <declarateur'>
        <liste-declarateurs'>; <programme>
        | (<liste-params> ) { <liste-declarations>
<liste-instructions> } <programme>
2- <liste-declarations> : <delcaration> <liste-declaration>
        | epsilon
3- <declarations> : int <liste-declarateurs> ;
        | chaine <liste-declarateurs> ;
4-1 <liste-declarateurs> : <declarateur> <liste-declarateurs'>
4-2 <liste-declarateurs'> : , <declarateur> <liste-declarateurs'>
        | epsilon
5-1 <Declarateur> : identificateur <Declarateur'>
5-2 <Declarateur'> : [ constante ] | epsilon
8- <type> : void | int | chaine

```

Maintenant, l'intersection qui était non vide est devenu vide, et "int, chaine" que faisaient problème sont réunies dans programme .

2.5 Améliorations

2.5.1 Les operations d'affectation

La grammaire tolere les operations de type "a = b +c", mais elle ne tolere pas "a = "chaine";" ou bien "a = 3" etc. On peut ajouter cette fonctionalite a travers les changements suivants :

```

<expression> : ( <expression>) <expression'>
               | - <expression> <expression'>
               | identificateur <expression''>
               | constante <expression''>
               | <chaine> <expression'>

```

Cela donnera à chaine les même manipulations d'un entrier. Si on veut restreindre le nombre d'opérations en un nombre précis, on peut créer une expression'' similaire

à expression' qui va contenir une liste d'opérations binaires bien choisis destinés à être utilisés avec les chaines. Pour notre projet, on choisit de donner toute liberté à manipuler les chaines et avoir la possibilité d'implémenter plusieurs fonctionnalités selon le besoin qui peuvent être utilisées avec les chaines sans avoir à implémenter une bibliothèque de gestion des chaines de caractères. Le but de ce projet et d'intégrer les chaines de caractères complètement dans le langage et ne pas restreindre ces manipulations seulement à la concaténation, etc...

2.5.2 Les parametres d'une fonction

Si on enleve la recursivite a gauche à liste parametres, on se trouve avec :

`<liste -parms> : , <parm> <liste -parms> | epsilon`

Mais cela pose probleme lorsqu'il s'agit de mettre plusieurs paramtres ou aucun. On la transforme vers ceci:

9-1 `<liste -parms> : <parm> <liste -parms'> | epsilon`

9-2 `<liste -parms'> : , <parm> <liste -parms'> | epsilon`

Même problème avec liste-expressions et on la remplace par ce qui suit :

21-1 `<liste -expressions> : <expression><liste -expressions'> | epsilon`

21-2 `<liste -expressions'> : ,<expression><liste -expressions'> | epsilon`

2.5.3 Le problème de Programme et Liste instructions

Sur le code de Programme, on retourne 'true' pour représenter le 'epsilon' sur la grammaire, ie, ne pas avoir d'autres définitions de fonctions ou déclarations. Mais cela pose probleme si on ajoute une instruction a l'exterieur de la fonction, même s'il detecte que cela ne doit pas etre tolere, le 'else return true' final le montre comme juste. Pour remedier a cela, on va ajouter une condition que verifie s'il s'agit d'une operation non tolerable, elle retourne false;

```
// Programme
...
    } else if (motCourant.ul == ERR) {
        //ERR c'est la fin du fichier ou erreur d'alphabet
        return true;
```

```

    } else {
        return false;
    }

```

Ce même problème on le trouve avec liste-instructions dans son code. Pour le résoudre, on ajoute un flag "insideFalse" qui se leve si on trouve une faute à l'intérieur d'une instruction.

2.5.4 Problème déclaration et instruction

```

bool AnalyseSyntaxique::declaration() {
    if (motCourant.ul == INT) {
        ...
    } else {
        return true;
    }
}

```

même que cela retourne un false, liste-declarations retourne un true à cause du epsilon (else return true;). Pour remédier à ce problème on va éclater declaration dans liste declarations et l'éliminer.

```

<liste-declaration> : int <liste-declarateurs>;
                    <liste-declaration>
| chaîne <liste-declarateurs>;
                    <liste-declaration> | epsilon

```

Même chose avec jinstruction_i. On va éclater jinstruction_i dans jliste instructions_i et on va avoir:

```

12-1 <liste-instructions> : <iteration> <liste-instructions>
| <selection> <liste-instructions>
| <saut> <liste-instructions>
| identificateur <aff-app>; <liste-instructions>
| <bloc> <liste-instructions>
| epsilon

```

2.5.5 Manipulation des chaînes

Après avoir fait ces améliorations, on trouve que chaîne peut être manipulée de la même façon qu'une constante, etc. Sur ce, on a le choix. Soit d'ajouter des opérations définies dans le langage que traitent les chaînes, soit les abolir.

Donc, on choisira de donner sens à ces opérations, tels que:

- + : Concatenation de deux chaines ou d'une chaine et un entier;
- - : suppression à partir de la fin de la chaine la partie commune et lever une erreur si on est face à un caractere non commun;
- == : Vérifier si deux chaines ont les mêmes caractères;
- != : L'opposé de ==
- j ou ĵ ou ĭ ou ĵ= ou ĵ= : raisonnent sur la somme du code ASCII des caractères.

Donc, on aura un langage riche qui supporte complètement le type chaine et on fera ces changements sur la grammaire :

```

20-1 <expression> : ( <expression>) <expression'>
                  | - <expression> <expression'>
                  | identificateur <expression''>
                  | <chaine> <expression'''>
                  | constante <expression'>
20-2 <expression'> : <binary-op> <expression> <expression'>
                  | epsilon
20-3<expression'''> : <binary-op'> <expression> <expression'>
                  | epsilon
20-4 <binary-op'> : + | -
    
```

Cette transformation va interdire les operations telles que "a = "aa" * "bb";" et "a = "aa" * 3;" mais va tolérer " a = 3 * "aa";"; Cette partie va être traitée dans la partie sémantique.

2.6 La grammaire finale

```

1-1 <programme> : int <programme'>
                  | chaine <programme'>
                  | void identificateur (<liste-params> ) {
                    <liste-declarations> <liste-instructions> } <programme>
                  | extern <type> identificateur (<liste-params> );
                    <programme>
                  | epsilon

1-2 <programme'> : identificateur <programme''>
1-3 <programme''> : <declarateur'> <liste-declarateurs'>;
                    <programme>
                    | (<liste-params> ) { <liste-declarations>
                      <liste-instructions> } <programme>
2- <liste-declaration> : int <liste-declarateurs>;
                    <liste-declaration>
                    | chaine <liste-declarateurs>;
                    <liste-declaration> | epsilon
3- <declarations> : int <liste-declarateurs> ;
                    | chaine <liste-declarateurs> ;
4-1 <liste-declarateurs> : <declarateur> <liste-declarateurs'>
4-2 <liste-declarateurs'> : , <declarateur> <liste-declarateurs'>
                    | epsilon
5-1 <Declarateur> : identificateur <Declarateur'>
5-2 <Declarateur'> : [ constante ] | epsilon
8- <type> : void | int | chaine
9-1 <liste-params> : <parm> <liste-params'> | epsilon
9-2 <liste-params'> : , <parm> <liste-params'> | epsilon
10- <parm> : int identificateur | chaine identificateur
12-1 <liste-instructions> : <iteration> <liste-instructions>
                    | <selection> <liste-instructions>
                    | <saut> <liste-instructions>
                    | identificateur <aff-app>; <liste-instructions>
                    | <bloc> <liste-instructions>
                    | epsilon
12-2 <aff-app> : <variable'> = <expression>
                    | (<liste-expressions>)
13- <iteration> : for (<affectation> ; <condition> ;
                    <affectation>) <instruction>

```

```

        | while (<condition>) <instruction>
14-1 <selection> : if (condition) <instruction> <selection'>
14-2 <selection'> : else <instruction> | epsilon
15-1 <saut> : <return> <saut'>
15-2 <saut'> : <expression> ; | ;
16- <affectation> : <variable> = <expression>
17- <bloc> : { <liste-instructions> }
19-1 <variable> : identificateur <variable'>
19-2 <variable'> : [ <expression> ] | epsilon
20-1 <expression> : ( <expression> ) <expression'>
                | - <expression> <expression'>
                | identificateur <expression''>
                | <chaine> <expression'''>
                | constante <expression'>
20-2 <expression'> : <binary-op> <expression> <expression'>
                | epsilon
20-3 <expression''> : ( <liste-expressions> ) <expression'>
                | <variable'> <expression'>
20-4 <expression'''> : <binary-op'> <expression> <expression'>
                | epsilon
21-1 <liste-expressions> : <expression> <liste-expressions'>
                | epsilon
21-2 <liste-expressions'> : , <expression> <liste-expressions'>
                | epsilon
22-1 <condition> : ! ( <condition> ) <condition'>
                | ( <condition> ) <condition'>
                | <expression> <binary-comp> <expression>
                <condition'>
22-2 <conditon'> : <binary-rel> <condition> <condition'>
                | epsilon
23-1 <binary-op> : + | - | * | / | << | >> | & | ||
23-2 <binary-op'> : + | -
24- <binary-rel> : && | ||
25- <binary-comp> : < | > | >= | <= | == | !=
<lire> : lire (<liste-declarateur>);
<ecrire> : écrire (<liste-declarateur>);
<chaine>: "<DefChaine>"
<DefChaine>: <lettre> <DefChaine>
                | <chiffre> <DefChaine>
                | epsilon

```



```
<entier>:<chiffre><entier>|epsilon  
<mot>: <letter> <mot> | epsilon  
<chiffre>: 0|1|2|3|4|5|6|7|8|9
```

2.7 Exemples d'erreurs levées par le syntaxique:

```
1 int a1;  
2 int a2;  
3  
4 chaîne a22  
5 extern void produit(int x,int y);
```

Figure 2.7: Texte source.

```
Erreur : ';' attendu ! Ligne : 5  
  
==> Le code est syntaxiquement faux.
```

Figure 2.8: Erreur : oublie du point virgule.

```
1 int a1;  
2 int a2;  
3 int a3[23;  
4
```

Figure 2.9: Texte source.

```
Erreur : ']' attendu ! Ligne : 3  
  
==> Le code est syntaxiquement faux.
```

Figure 2.10: Erreur : oublie de fermer un crochet ouvrant.

```
1 int a3[23];
2
3 chaine a22;
4
5 a3[3] = 7;
```

Figure 2.11: Texte source.

```
Erreur : int, chaine, void, extern attendu ! Ligne : 5
==> Le code est syntaxiquement faux.
```

Figure 2.12: Erreur : opération non permise.

```
1 int a3[23];
2
3 chaine a22;
4 extern void produit(int x,int y);
5 extern add(int x,int y);
6
```

Figure 2.13: Texte source.

```
Erreur : int, chaine, void attendu ! Ligne : 6
==> Le code est syntaxiquement faux.
```

Figure 2.14: Erreur : oublie de donner un type lors de la déclaration d'une fonction.

```
1 int a3[23];
2
3 chaine a22;
4
5
6 extern void (int x,int y);
7 extern int add(int x,int y);
```

Figure 2.15: Texte source.

```
Erreur : ID attendu ! Ligne : 6
==> Le code est syntaxiquement faux.
```

Figure 2.16: Erreur : oublie de l'identifiant.

Chapter 3

Partie Sémantique

3.1 Introduction

La partie sémantique est la partie du compilateur qui s'occupe des vérifications liés à l'exécution du programme. La phase d'analyse sémantique rassemble des informations qui serviront dans la production de code et contrôle un certain nombre de coherences sémantiques. Par exemple, c'est au cours de cette phase que sont déterminés les instructions, les expressions et les identificateurs. Sont également contrôlés les types dans les affectations et les passages de paramètres. Les types d'indices utilisés pour les tableaux peuvent aussi être vérifiés.

3.2 La grammaire du langage avec des actions sémantiques

```

1-1 <programme> : int <programme'>
                | chaine <programme'>
                | void identificateur {verifierRedefinition}(<liste-parms> ) {
<liste-declarations> <liste-instructions> } <programme>
                | extern <type> identificateur {verifierRedefinition} (<liste-parms> );
<programme>
                | epsilon

1-2 <programme'> : identificateur {verifierRedefinition} <programme">
5-1 <Declareur> : identificateur {verifierRedefinition} <Declareur'>
10- <parm> : int identificateur {verifierRedefinition} | chaine identificateur {verifierRedefinition}
12-1 <liste-instructions> : <iteration> <liste-instructions>
    | <selection> <liste-instructions>
    | <saut> <liste-instructions>
    | identificateur {estDefini} <aff-app>;<liste-instructions>
    | <bloc><liste-instructions>
    | epsilon

```

Figure 3.1: Grammaire avec actions sémantiques.

```

12-2 <aff-app>: {checkType} <variable'>=<expression>
    | (<liste-expressions>)
19-1 <variable> : identificateur {estDefini} <variable'>
19-2 <variable'> : [ {estTable} <expression> ] | epsilon
20-1 <expression> : ( <expression> ) <expression'>
    | - <expression> <expression'>
    | identificateur {estDefini} <expression">
    | <chaine> <expression'">
    | constante <expression'>
    | epsilon

```

Figure 3.2: Grammaire avec actions sémantiques

3.3 Définition des actions sémantiques

3.3.1 Vérification des redéfinitions

L'action **vérifierRedéfinition** vérifie si la déclaration d'une certaine variable a été déjà faite dans le même contexte ou dans un contexte supérieur. Son prototype :

```
bool verifierRedefinition(unite u, int ligne
                        , bool emplacement, int contexte, int fonction);
```

- *AnalyseLexical::unite u* : la structure utilisée dans la partie lexicales pour stocker les unités lexicales et les attributs correspondants. Son utilisation se fait dans la partie syntaxique où on passe le 'motCourant', qui est une unité, en paramètres à cette fonction pour l'analyse sémantique;
- *int ligne* : c'est la ligne où on se trouve.
- *bool emplacement* : si on vient d'une importation de fonction avec le mot clé externe; cette booléen prend la valeur "vraie" pour ne pas compter les variables introduites dans le prototype de la fonction comme variables du programme;
- *int contexte* : c'est le contexte où on se trouve. Soit on est au contexte global (=0); Soit on est à l'intérieur d'une fonction (=1);
- *int fonction*: contient le nombre 0 si on est dans un contexte global, sinon, cette variable prend comme valeur l'ordre de définition de la fonction.

L'appel de cette fonction avant chaque declaration nous permet de vérifier dans la table des symboles utilisées si on est passé par cette variable deux fois, la première c'est la déclaration de la variable et la deuxième c'est la redéclaration.

3.3.2 Vérification des définitions

L'action **estDéfini** permet de vérifier si une variable ou une fonction a été déclarée avec qu'elle soit utilisée. Son prototype:

```
bool AnalyseSemantique::estDefini(unite u, int contexte ,
                                int ligne , int fonction)
```

Son utilisation est similaire à celle de la vérification des redéfinitions. Son appel se fait après chaque identificateur qui introduit une manipulation de la variable et pas sa déclaration. Donc, elle vérifie dans la table des symboles si on a déjà défini cette

variable dans le même contexte ou dans un contexte supérieur et avant la ligne où elle est utilisée.

3.3.3 Vérification du caractère simple ou composé d'une variable

L'action **estTable** vérifie si la variable, sur laquelle on utilise les crochets, et un tableau ou non. Son prototype et ses paramètres sont similaires aux vérifications présentées et on l'utilise lorsqu'on utilise les crochets sur une variable dans une phase de manipulation différente de la déclaration.

3.3.4 Vérification des types

L'action **CheckType** vérifie si le type manipulé correspond aux opérations qu'on lui effectue. Par exemple, l'affectation d'une chaîne de caractères à un "int". Cette fonction vérifie aussi si la variable manipulée est un tableau, et oblige de spécifier l'index où on veut affecter la valeur manipulée. Son prototype:

```
bool AnalyseSemantique::checkType[0xFFFFD]  
    unite a, unite b, unite c, int nbCrochets  
    , int ligne, int contexte, int fonction)
```

- Les trois unités passées en paramètres contiennent les unités correspondantes à une opération "a = b + c" ou "a = b" avec "c" nulle.
- Le nombre de crochets correspond au nombre de crochets utilisés dans l'opération pour savoir si les variables qu'on manipule sont des tableaux et une spécification de l'index concerné est obligée.

3.3.5 Vérifications des index nuls

Cette fonctionnalité a été ajoutée lors du parcours du code à traduire par le sémantique. s'il tombe sur une taille de tableau négative, il la signale.



```
25  chaine testc[-3];
```

Figure 3.3: Erreur index négative.


A screenshot of a compiler error message. The text is white on a dark purple background. It reads: "Tableau de taille negative, Ligne : 25".

Tableau de taille negative, Ligne : 25

Figure 3.4: Erreur index négative.

3.3.6 Vérification des manipulations de chaines

Dans la partie syntaxique, on a donné au type chaine que deux opérations binaires valables "+" et "-". Si on essaie de faire d'autres opérations, le syntaxique lève une erreur de syntaxe. Mais cela n'est pas valable dans les deux sens: "*chaine*" * 3 va être signalée, mais 3 * "*chaine*" ne va pas l'être. Donc cela doit être implémenté dans cette partie et l'action de vérification des types le couvre.

3.4 Structures utilisées

3.4.1 Table des symboles

Pour implémenter ces fonctionnalités, on utilise une table de symboles qui se remplit avant l'exécution du syntaxique

```
vector<variable*> idents;
```

C'est un vecteur de pointeurs 'variable', une structure qui sera présentée par la suite.

3.4.2 La structure variable

C'est une structure qui contient les données prises par le parcours fait par la fonction *ajouterIdentifiants()* du sémantique qui fait un parcours du code et ajoute tous les identifiants, des variables et fonctions etc, dans cette table pour être exploitée par les actions sémantiques par la suite. Sa définition:

```
struct variable {
    string ident; // Va contenir l'identifiant de la
                  variable courante
    string type=""; // Va contenir le type des
                  identifiants
    int scope = 0 ; //Variables globales , locales .... par
                  niveau (0 = globale)
    int taille = 0; // s'il s'agit d'un tableau
    int times = 0; // nombre de fois on est passe par cet
                  identifiant
    int ou = 0; //Dans quelle ligne la definition de la
                  variable a ete rencontree
    int function = 0; // la fonction ou se trouve la
                  declaration de la variable.
};
```

3.5 Exemples d'erreurs levées par le sémantique.

```
1 int a1;  
2 int a2;  
3 int a3[23];  
4 int a1;
```

Figure 3.5: Erreur de redéfinition.

```
Redefinition de : a1 , ligne : 4  
==> Le code est semantiquement faux ...
```

Figure 3.6: Erreur de redéfinition.

```
24 int s;  
25 chaine testc;  
26  
27 s = testc;
```

Figure 3.7: Erreur des types lors d'une affectation.

```
Incoherence des types manipules... Ligne : 27
```

Figure 3.8: Erreur des types lors d'une affectation.

```
25 int s;|
26
27 s = "ch";
```

Figure 3.9: Erreur des types lors d'une affectation.

```
Pas de conversion connue entre le type entier et les chaines ... Ligne : 28
```

Figure 3.10: Erreur des types lors d'une affectation.

```
25 chaine testc;
26
27 testc = 33;
```

Figure 3.11: Erreur des types lors d'une affectation.

```
Pas de conversion connue entre le type entier et les chaines ... Ligne : 28
```

Figure 3.12: Erreur des types lors d'une affectation.

```
25 chaine testc[3];
26
27 testc = "ch";
```

Figure 3.13: Erreur d'indice d'un tableau.

```
/eillez specifier l'index ou vous voulez stocker... Ligne : 27
```

Figure 3.14: Erreur d'indice d'un tableau.

```
int var1;
var1 = var2 +3;
```

Figure 3.15: Erreur de définition.

```
var2 doit etre defini avant d'etre utilisee... Ligne 28
```

Figure 3.16: Erreur de définition.