L'École Nationale Supérieure
d'Informatique et d'Analyse
des Systèmes

Projet compilation

CONCEPTION D'UN LANGAGE ET REALISATION D'UN COMPILATEUR AVEC LE LANGAGE C

Filière : ISEM 2ème Année

Réalisé par :

Chaimae HIMANE
MOHAMED EL HLAFI
Abdelkarim BEN BRAHIM
Amine CHRIFI ALAOUI

Encadré par :

Pr. Younes TABII

TABLES DES MATIÈRES

INTRODUCTION

Le nombre de langage d'ordinateur existant aujourd'hui est assez énorme et continue sans cesse de croître. Les langages machines sont utilisés dans plusieurs domaines pour des buts différents, Ils se diversifient selon la finalité d'usage ou les exigences métiers. Ainsi, Ils vont des langages de programmation traditionnels comme C, C++ et Java en allant aux langages de markup (soit balisage en français) comme HTML et XML ou encore aux langages de modélisation comme UML. L'intégration d'un nouveau langage de programmation ,offrant un plus, nécessite une étude et une analyse approfondie qui fait l'objet de ce rapport.

Afin d'appliquer les méthodologies et les notions enseignées durant le cours compilation, nous sommes invités à réaliser un projet qui va nous permettre d'appliquer nos connaissances théoriques sur le champ pratique. Ce projet de compilation a pour objet l'élaboration d'un langage de programmation ainsi qu'un compilateur pour l'analyse lexicale et syntaxique. Pour ce faire, nous nous sommes basés, d'une part, sur la structure du langage VHDL qui est un langage de description matérielle, utilisé pour décrire des systèmes logiques synchrones ou asynchrones et d'autre part sur la logique des langages de programmation enseignés durant les deux années écoulées afin de donner naissance à un nouveau langage combinant entre les deux structures.

Le présent travail est destiné à la présentation du langage réalisé et sera structuré comme suit :

La première partie est consacrée à la présentations des éléments principaux de notre langage engendré par une grammaire LL1. La deuxième partie se focalise sur l'analyseur lexical et ses différentes parties, à savoir la liste des tokens, et les fonctions utilisées. Finalement, une troisième partie est destinée à la présentation de l'analyseur syntaxique, en particulier l'arbre syntaxique et les fonctions de ce PARSER.

I- CONSTITUTION DES ELEMENTS DU LANGAGE

1. LES OPÉRATEURS

Les opérateurs sont des symboles qui permettent de manipuler des variables, c'est-à-dire effectuer des opérations, les évaluer, etc.

On distingue plusieurs types d'opérateurs :

- les opérateurs de calcul
- les opérateurs de comparaison
- les opérateurs logiques

• Les opérateurs de calcul :

Les opérateurs de calcul permettent de modifier mathématiquement la valeur d'une variable. Alors on résume les opérateurs de calcul de notre langage dans le tableau suivant :

Opérateur	Dénomination	Effet	Exemple	Résultat (avec x valant 7)
+	opérateur d'addition	Ajoute deux valeurs	x+3	10
-	opérateur de soustraction	Soustrait deux valeurs	x-3	4
*	opérateur de multiplication	Multiplie deux valeurs	x*3	21
/	opérateur de division	Divise deux valeurs	x/3	2.3333333
<-	opérateur d'affectation	Affecte une valeur à une variable	x=3	Met la valeur 3 dans la variable x

Tableau 1 : OPERATEUR DE CALCUL

Les opérateurs de comparaison :

les opérateurs de comparaison permettent de comparer deux entités de même types. Alors la table suivante présente les opérateurs de comparaison de notre langage.

Opérateur	Dénomination	Effet	Exemple	Résultat (avec x valant 7)
=	opérateur d'égalité	Compare deux valeurs et vérifie leur égalité	x=3	Retourne 1 si x est égal à 3, sinon 0
<	opérateur d'infériorité stricte	Vérifie qu'une variable est strictement inférieure à une valeur	x<3	Retourne 1 si x est inférieur à 3, sinon 0
<=	opérateur d'infériorité	Vérifie qu'une variable est inférieure ou égale à une valeur	x<=3	Retourne 1 si x est inférieur ou égal à 3, sinon 0
>	opérateur de supériorité stricte	Vérifie qu'une variable est strictement supérieure à une valeur	x>3	Retourne 1 si x est supérieur à 3, sinon 0
>=	opérateur de supériorité	Vérifie qu'une variable est supérieure ou égale à une valeur	x>=3	Retourne 1 si x est supérieur ou égal à 3, sinon 0
<>	opérateur de différence	Vérifie qu'une variable est différente d'une valeur	X<>3	Retourne 1 si x est différent de 3, sinon 0

Tableau 2: OPERATEUR DE COMPARAISON

Les opérateurs logiques (booléens) :

Ce type d'opérateur permet de vérifier si plusieurs conditions sont vraies ou non, alors on les indiques dans une table dans l'ordre de classification est le suivant :

Opérateur	Dénomination	Effet	Syntaxe
and	OU logique	Vérifie qu'une des conditions est réalisée	and(a,b)
or	ET logique	Vérifie que toutes les conditions sont réalisées	or(a,b)
not	NON logique	Inverse l'état d'une variable booléenne (retourne la valeur 1 si la variable vaut 0, 0 si elle vaut 1)	not(a)

Tableau 3: OPERATEUR LOGIQUE

2. LES TYPES

Une variable nous fournit un stockage nommé que nos programmes peuvent manipuler. Chaque variable de notre langage a un type spécifique, qui détermine la taille et la disposition de la mémoire de la variable, la gamme des valeurs qui peuvent être stockées dans cette mémoire et l'ensemble des opérations qui peuvent être appliquées à la variable.

On doit déclarer toutes les variables avant qu'elles ne puissent être utilisées dans le bloc declare_var().

Voici la forme de base d'une déclaration des variables :

declar_var(TYPE variable [= value][, variable [= value] ...];);

Dans notre langage on a défini un typage statique en utilisant 6 types de base :

int , float ,char , string ,bit ,void .

3. LES STRUCTURES CONDITIONNELLES

Souvent les problèmes nécessitent l'étude de plusieurs situations qui ne peuvent pas être traitées par les séquences d'actions simples. Puisqu'on a plusieurs situations, et qu'avant l'exécution, on ne sait pas à quel cas de figure on aura à exécuter, on doit prévoir tous les cas possibles. Ce sont les structures conditionnelles qui le permettent, en se basant sur ce qu'on appelle prédicat ou condition.

Alors nous allons définir ces structures sous la forme suivante :

if condition **then**
  instructions ;
elsif condition **then**
  instructions ;
else instructions ;
end if

Lorsque les cas à gérer sont nombreux donc pour remplacer une série (souvent peu élégante) de if… else nous allons définir dans notre langage la structure switch en se basant sur la syntaxe suivante :

```
switch (id)
        case expression 1:instructions ; break ;
        case expression 2:instructions ; break ;
        case expression 3:instructions ; break ;
                .
                .
        case expression n:instructions ; break ;
        default :instructions ; break ;
end switch
```

4. LES BOUCLES

Il peut arriver que nous devons exécuter un bloc de code plusieurs fois. En général, les instructions sont exécutées de manière séquentielle : La première instruction d'une fonction est exécutée en premier, suivie de la seconde, et ainsi de suite.

Les langages de programmation fournissent diverses structures de contrôle qui permettent des chemins d'exécution plus compliqués.

Une instruction de boucle nous permet d'exécuter une instruction ou un groupe d'instructions plusieurs fois et la forme générale d'une instruction de boucle dans notre langage est la suivante :

4.1. Boucle FOR

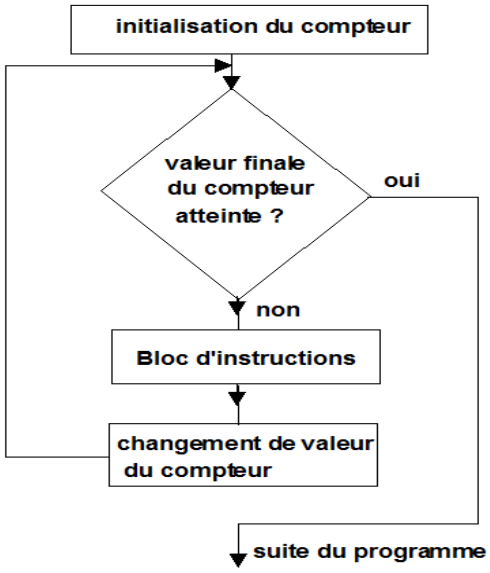

Figure 1 : BOUCLE FOR

La syntaxe de cette boucle dans notre langage est la suivante :

for(id<- ; condition d'arrêt ; le pas)

instruction 1 ;

instruction 2 ;

.

instruction n ;

end for

4.2. Boucle TANT QUE

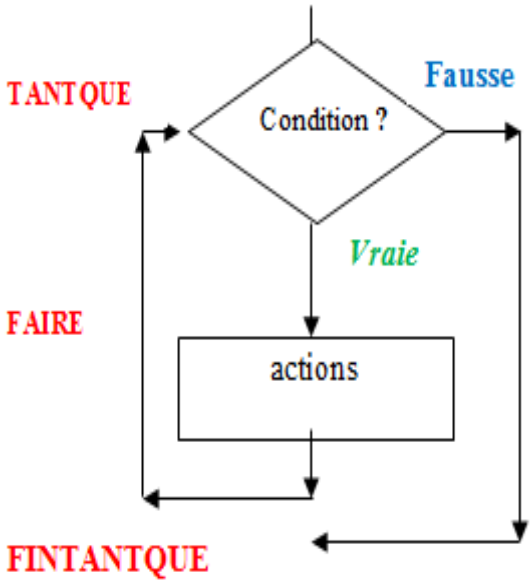

Figure 2 : BOUCLE TANT QUE

Pour cette boucle la syntaxe proposée est la suivante :

While condition **then**

instruction 1 ;

instruction 2 ;

.

.

instruction n ;

end while

5. LES INSTRUCTIONS DE SAISI

Elles permettent de récupérer une valeur venant de l'extérieur ou de transmettre une valeur à l'extérieur.

Nous allons définir dans notre langage la fonction print_out pour transmettre une valeur à l'extérieur.et pour transmettre une valeur venant de l'extérieur nous aurons la fonction put_in.

Exemple :

```
print_out("hello ",a);
put_in(a);
```

6. LA GRAMMAIRE

La grammaire proposée pour engendrer le nouveau langage est une grammaire hors contexte définie sous la forme G =<S,T,NT,RP>

6.2. LES NON TERMINAUX (NT)

Les ensembles des symboles non terminaux NT qui n'apparaissent pas dans les mots générés, mais qui sont utilisés au cours de la génération. Alors la liste des non-terminaux de notre grammaire est comme suit :

NT= {**STARTP; VAR; TYPE; START; STARTF; STARTMF; INSTS; INST; ENDP; FUNCTION; AFFEC; SI; TANTQUE; FOR; ECRIRE; LIRE; SWITCH; EXP; COND; CHOICE; RELOP; ADDOP; TERM; MULOP; FACT; PREDEF; ID; LETTRE; CHIFFRE**}

6.3. LES TERMINAUX (T)

Les symboles terminaux T est le vocabulaire terminal, c'est-à-dire l'alphabet sur lequel est défini le langage

T ={**declare_var, int, float, void, char, bit,<-, =, <>,>=,<=, ;,+, -, /, not, and, floor, mod, print_out, put_in, (,), start, function, Main_function, return, end, end_prog, if, then, while, for, switch, default, break, case, elsif, else, ", a…z, A…Z, 0…9**}

6.4. L'AXIOME (S)

S ∈ N est le symbole de départ ou axiome. C'est à partir de ce symbole non terminal que l'on commencera la génération de mots au moyen des règles de la grammaire : **S -> STARTP**

6.5. LES REGLES DE PRODUCTION (RP)

STARTP::=	**start_prog VAR START ENDP**
VAR::=	Declare_Var (A); |epsilon
A::=	TYPE ID S' |epsilon
TYPE::=	int | float | char | bit | void
S'::=	<- NUM ; A |; A | , ID S'
START::=	start TYPE F
F::=	STARTF start TYPE STARTMF | STARTMF
STARTF::=	function ID (ARG) INSTS B
ARG::=	TYPE ID ARG | ,TYPE ID ARG | epsilon
STARTMF::=	Main_function ( ) INSTS B
INSTS::=	INST; return ID;
INST::=	FUNCTION|AFFEC| SI | TANTQUE | FOR | ECRIRE | LIRE | SWITCH | epsilon
B::=	end V
V::=	ID | Main_function
ENDP::=	end_prog
FUNCTION::=	(R)
R::=	EXP R |, EXP R | epsilon
AFFEC::=	ID <- EXP
SI::=	if COND then INST CHOICE end if
TANTQUE::=	while COND then INST end while
FOR::=	for (ID <- FACT; COND; ID <-EXP) then INST end for
ECRIRE ::=	print_out (T)
LIRE ::=	put_in (ID)
SWITCH ::=	switch (ID) G default: INST; break; end switch
EXP::=	TERM ADDOP TERM |TERM
COND::=	EXP RELOP EXP
CHOICE::=	elsif COND then INST | else INST |epsilon
G::=	case FACT: I NST ; break; G| epsilon
T::=	EXP K | K EXP
K::=	'' CHAINE '' K | , T
CHAINE::=	ID CHAINE | epsilon
RELOP::=	= | <> | < | > | <= | >=
ADDOP::=	+| -
TERM::=	FACT C
C::=	MULOP FACT |epsilon
MULOP::=	* | /
FACT::=	ID | NUM | (EXP)
PREDEF::=	not | floor |and |or | mod
ID::=	LETTRE Q | PREDEF
Q::=	LETTRE Q | CHIFFRE Q |epsilon
LETTRE::=	a |…|z |A|...|Z
NUM ::=	CHIFFRE NUM | epsilon
CHIFFRE::=	0|1|...|9

Tableau 4 : REGLE DE PRODUCTION

7. VERIFICATION DE LA GRAMMAIRE (LL1)

TABLE D'ANALYSE :

Colonne1	start_prog	Declare_Var	(	)	;	int	float	char	bit	void	<-
S	S ::= STARTP $										
STARTP	STARTP ::= start_prog VAR START ENDP										
VAR		VAR ::= Declare_Var (A) ;									
A				A ::= ε		A ::= TYPE ID S'	A ::= TYPE ID S'	A ::= TYPE ID S'	A ::= TYPE ID S'	A ::= TYPE ID S'	
TYPE						TYPE ::= int	TYPE ::= float	TYPE ::= char	TYPE ::= bit	TYPE ::= void	
S'					S' ::= ; A						S' ::= <- NUM ; A
START											
F											
STARTF											
ARG				ARG ::= ε		ARG ::= TYPE ID ARG , TYPE ID ARG	ARG ::= TYPE ID ARG , TYPE ID ARG	ARG ::= TYPE ID ARG , TYPE ID ARG	ARG ::= TYPE ID ARG , TYPE ID ARG	ARG ::= TYPE ID ARG , TYPE ID ARG	
STARTMF											
INSTS			INSTS ::= INST ; return ID ;		INSTS ::= INST ; return ID ;						
INST			INST ::= FUNCTION		INST ::= ε						
B											
V											
ENDP											
FUNCTION			FUNCTION ::= (R)								
R			R ::= EXP R	R ::= ε							
AFFEC											
SI											
TANTQUE											
FOR											
ECRIRE											
LIRE											
SWITCH											
EXP			EXP ::= TERM Y								
Y			Y ::= ε	Y ::= ε	Y ::= ε						
COND			COND ::= EXP RELOP EXP								
CHOICE											
G											
T			T ::= EXP K								
K											
CHAINE											
RELOP											
ADDOP											
TERM			TERM ::= FACT C								
C			C ::= ε	C ::= ε	C ::= ε						
MULOP											
FACT			FACT ::= (EXP)								
PREDEF											
ID											
Q											
LETTRE											
NUM											
N											
CHIFFRE											

Tableau 5 : TABLE D'ANALYSE

TABLE D'ANALYSE (Suite) :

Colonne1	,	start	function	Main_function	return	end	end_prog	if	then	while	for
S											
STARTP											
VAR		VAR ::= ε									
A											
TYPE											
S'	S' ::= , ID S'										
START		START ::= start TYPE F									
F			F ::= STARTF start TYPE STARTMF	F ::= STARTMF							
STARTF			STARTF ::= function ID (ARG) INSTS B								
ARG	ARG ::= ε										
STARTMF				STARTMF ::= Main_function ( ) INSTS B							
INSTS								INSTS ::= INST ; return ID ;		INSTS ::= INST ; return ID ;	INSTS ::= INST ; return ID ;
INST						INST ::= ε		INST ::= SI		INST ::= TANTQUE	INST ::= FOR
B						B ::= end V					
V				V ::= Main_function							
ENDP							ENDP ::= end_prog				
FUNCTION											
R	R ::= , EXP R										
AFFEC											
SI								SI ::= if COND then INST CHOICE end if			
TANTQUE										TANTQUE ::= while COND then INST end while	
FOR											FOR ::= for (ID <- FACT ; COND ; ID <- EXP) then INST end for
ECRIRE											
LIRE											
SWITCH											
EXP											
Y	Y ::= ε					Y ::= ε			Y ::= ε		
COND											
CHOICE						CHOICE ::= ε					
G											
T	T ::= K EXP										
K	K ::= , T										
CHAINE											
RELOP											
ADDOP											
TERM											
C	C ::= ε					C ::= ε			C ::= ε		
MULOP											
FACT											
PREDEF											
ID											
Q											
LETTRE											
NUM											
N											
CHIFFRE											

Tableau 6:TABLE D'ANALYSE (suite)

TABLE D'ANALYSE (Suite) :

Colonne1	print_out	put_in	switch	default	:	break	elsif	else	case	"	=	<>	<	>	<=	>=
S																
STARTP																
VAR																
A																
TYPE																
S'																
START																
F																
STARTF																
ARG																
STARTMF																
INSTS	INSTS ::= INST ; return ID ;	INSTS ::= INST ; return ID ;	INSTS ::= INST ; return ID ;													
INST	INST ::= ECRIRE	INST ::= LIRE	INST ::= SWITCH				INST ::= ε	INST ::= ε								
B																
V																
ENDP																
FUNCTION																
R																
AFFEC																
SI																
TANTQUE																
FOR																
ECRIRE	ECRIRE ::= print_out (T)															
LIRE		LIRE ::= put_in (ID)														
SWITCH			SWITCH ::= switch (ID) G default : INST ; break ; end switch													
EXP																
Y							Y ::= ε	Y ::= ε		Y ::= ε	Y ::= ε	Y ::= ε	Y ::= ε	Y ::= ε	Y ::= ε	Y ::= ε
COND																
CHOICE							CHOICE ::= elsif COND then INST	CHOICE ::= else INST								
G				G ::= ε					G ::= case FACT : INST ; break ; G							
T										T ::= K EXP						
K										K ::= " CHAINE " K						
CHAINE										CHAINE ::= ε						
RELOP											RELOP ::= =	RELOP ::= <>	RELOP ::= <	RELOP ::= >	RELOP ::= <=	RELOP ::= >=
ADDOP																
TERM																
C							C ::= ε	C ::= ε		C ::= ε	C ::= ε	C ::= ε	C ::= ε	C ::= ε	C ::= ε	C ::= ε
MULOP																
FACT																
PREDEF																
ID																
Q																
LETTRE																
NUM																
N																
CHIFFRE																

Tableau 7: TABLE D'ANALYSE (suite)

TABLE D'ANALYSE (Suite) :

Colonne1	+	-	*	/	not	floor	and	or	mod	a	0
S											
STARTP											
VAR											
A											
TYPE											
S'											
START											
F											
STARTF											
ARG											
STARTMF											
INSTS					INSTS ::= INST ; return ID ;	INSTS ::= INST ; return ID ;	INSTS ::= INST ; return ID ;	INSTS ::= INST ; return ID ;	INSTS ::= INST ; return ID ;	INSTS ::= INST ; return ID ;	
INST					INST ::= AFFEC	INST ::= AFFEC	INST ::= AFFEC	INST ::= AFFEC	INST ::= AFFEC	INST ::= AFFEC	
B											
V					V ::= ID	V ::= ID	V ::= ID	V ::= ID	V ::= ID	V ::= ID	
ENDP											
FUNCTION											
R					R ::= EXP R	R ::= EXP R	R ::= EXP R	R ::= EXP R	R ::= EXP R	R ::= EXP R	R ::= EXP R
AFFEC					AFFEC ::= ID <- EXP	AFFEC ::= ID <- EXP	AFFEC ::= ID <- EXP	AFFEC ::= ID <- EXP	AFFEC ::= ID <- EXP	AFFEC ::= ID <- EXP	
SI											
TANTQUE											
FOR											
ECRIRE											
LIRE											
SWITCH											
EXP					EXP ::= TERM Y	EXP ::= TERM Y	EXP ::= TERM Y	EXP ::= TERM Y	EXP ::= TERM Y	EXP ::= TERM Y	EXP ::= TERM Y
Y	Y ::= ADDOP TERM	Y ::= ADDOP TERM			Y ::= ε	Y ::= ε	Y ::= ε	Y ::= ε	Y ::= ε	Y ::= ε	Y ::= ε
COND					COND ::= EXP RELOP EXP	COND ::= EXP RELOP EXP	COND ::= EXP RELOP EXP	COND ::= EXP RELOP EXP	COND ::= EXP RELOP EXP	COND ::= EXP RELOP EXP	COND ::= EXP RELOP EXP
CHOICE											
G											
T					T ::= EXP K	T ::= EXP K	T ::= EXP K	T ::= EXP K	T ::= EXP K	T ::= EXP K	T ::= EXP K
K											
CHAINE					CHAINE ::= ID CHAINE	CHAINE ::= ID CHAINE	CHAINE ::= ID CHAINE	CHAINE ::= ID CHAINE	CHAINE ::= ID CHAINE	CHAINE ::= ID CHAINE	
RELOP											
ADDOP	ADDOP ::= +	ADDOP ::= -									
TERM					TERM ::= FACT C	TERM ::= FACT C	TERM ::= FACT C	TERM ::= FACT C	TERM ::= FACT C	TERM ::= FACT C	TERM ::= FACT C
C	C ::= ε	C ::= ε	C ::= MULOP FACT	C ::= MULOP FACT	C ::= ε	C ::= ε	C ::= ε	C ::= ε	C ::= ε	C ::= ε	C ::= ε
MULOP			MULOP ::= *	MULOP ::= /							
FACT					FACT ::= ID	FACT ::= ID	FACT ::= ID	FACT ::= ID	FACT ::= ID	FACT ::= ID	FACT ::= NUM
PREDEF					PREDEF ::= not	PREDEF ::= floor	PREDEF ::= and	PREDEF ::= or	PREDEF ::= mod		
ID					ID ::= PREDEF	ID ::= PREDEF	ID ::= PREDEF	ID ::= PREDEF	ID ::= PREDEF	ID ::= LETTRE Q	
Q										Q ::= LETTRE Q	Q ::= CHIFFRE Q
LETTRE										LETTRE ::= a	
NUM											NUM ::= CHIFFRE N
N											N ::= NUM
CHIFFRE											CHIFFRE ::= 0

Tableau 8 : TABLE D'ANALYSE (suite)

8. EXEMPLE DU LANGAGE

```
start_prog
        declar_var(int a<-1 ,b<-2;
                                        float c,d;
                        bit e<-0, f<-1;);

        start int Function moyenne(int a, int b )

                c <- a+b/2;
                return c;

        end Function moyenne

        start void Main_function()

                d <- Function moyenne(a,b);
                if d>=2 then
                        print_out(d);
                        elsif d<2 then
                        print_out("hi"d/2,z"hi",a);
                end if

                while c<10 then
                end while

                switch (a)
                  case 1 : print_out("chhh");break;
                  case2 : put_in(e,c);
                           print_out("chhh");break;
                end switch

                g<- Function and(e,f);
                print_out(g);
                print_out("fin main function");

        end Main_function


end_prog
```

Figure 3 : EXEMPLE DU LANGAGE

II- ANALYSEUR LEXICAL

Le rôle de l'analyseur lexical consiste à valider lexicalement le texte ou un programme en entrée, c'est-à-dire s'assurer que tous les mots de ce texte quels qu'ils soient (entièrement visibles ou non) correspondent à une des unités lexicales définies dans la liste des Tokens.

À partir de la liste des Tokens, on génère l'analyseur lexical en se servant de ces derniers. Lorsqu'il est invoqué, l'analyseur lexical lit le texte en entrée, caractère par caractère, et s'arrête dès qu'il reconnaît un mot satisfaisant le modèle d'une des unités lexicales définies (Token) dans la liste. Il retourne alors l'unité lexicale associée au modèle reconnu. Le même processus reprend jusqu'à ce que la fin du fichier soit atteinte en retournant les erreurs rencontrées , c'est-à-dire l'ensembles des mots qui ne correspondent à aucune des unités lexicales.

L'analyseur lexical permet d'avoir une liste chainée des tokens reconnus durant la vérification du texte ce qui permet de retenir l'essentiel du code autrement dit le code nécessaire pour effectuer l'analyse syntaxique.

1. LISTE DES TOKENS

Lors de la constitution de la liste des tokens qui serait la base de notre analyseur il a été convenu de décortiquer notre liste sous trois parties distinctes représentées dans les trois tableaux suivants :

1.2. Les mots clés

LES MOTS CLÉS	
start_prog	STARTP_TOKEN
declare_Var	VAR_TOKEN
start	START_TOKEN
Function	FUNCTION_TOKEN
MAIN_function	MAIN_TOKEN
end	END_TOKEN
end_prog	ENDPROG_TOKEN
if	IF_TOKEN
elsif	ELSIF_TOKEN
else	ELSE_TOKEN
then	THEN_TOKEN
switch	SWITCH_TOKEN
case	CASE_TOKEN
break	BREAK_TOKEN
default	DEFAULT_TOKEN
while	WHILE_TOKEN
for	FOR_TOKEN
print_out	PRINT_TOKEN
put_in	PUT_TOKEN
return	RETURN_TOKEN
int	INT_TOKEN

Tableau 9 : MOTS CLES -> TOKENS

1.3. Les symboles spéciaux

LES SYMBOLES SPÉCIAUX	
;	PV_TOKEN
+	PLUS_TOKEN
-	MOINS_TOKEN
*	MULT_TOKEN
/	DIV_TOKEN
,	VIR_TOKEN
<-	AFF_TOKEN
<	INF_TOKEN
<=	INFEG_TOKEN
=	EG_TOKEN
>	SUP_TOKEN
>=	SUPEG_TOKEN
<>	DIFF_TOKEN
(	PO_TOKEN
)	PF_TOKEN
ID	ID_TOKEN
NUM	NUM_TOKEN
EOF	EOF_TOKEN

Tableau 10 : SYMBOLES SPECIAUX -> TOKENS

1.4. Les symboles spéciaux

LES SYMBOLES ERRONEES	
LE RESTE	ERREUR_TOKEN

Tableau 11 : SYMBOLES ERRONEES -> TOKEN

2. LES FONCTIONS DE L'ANALYSEUR

Les fonctions utilisées pour cet analyseur lexical permettent d'ouvrir le fichier par la fonction « Openfile() ; », de lire le fichier texte à analyser caractère par caractère par la fonction « lire_car() ; » pour constituer le mot qui serait de nature d'un mot par la fonction « lire_mot() ; » ou de nature d'un nombre par « lire_nombre() ; » ou encore un symbole par « lire_symbole() ». une fois la constitution du mot est terminée on procède à la vérification pour décider si ce mot doit être rejeté ou ajouté à notre liste chainée par la fonction « ajouter() ; »

Nous avons ajouté une fonction afficher liste qui permet de voir la liste des tokens générée à partir du fichier à tester et une autre fonction de conversion réalisée à partir d'une liste d'énumération des tokens afin de pouvoir retourner le type du token et aussi pour un éventuel débogage.

Le schéma suivant mets en évidence la hiérarchie des fonctions adoptées pour réaliser cette partie d'analyse :

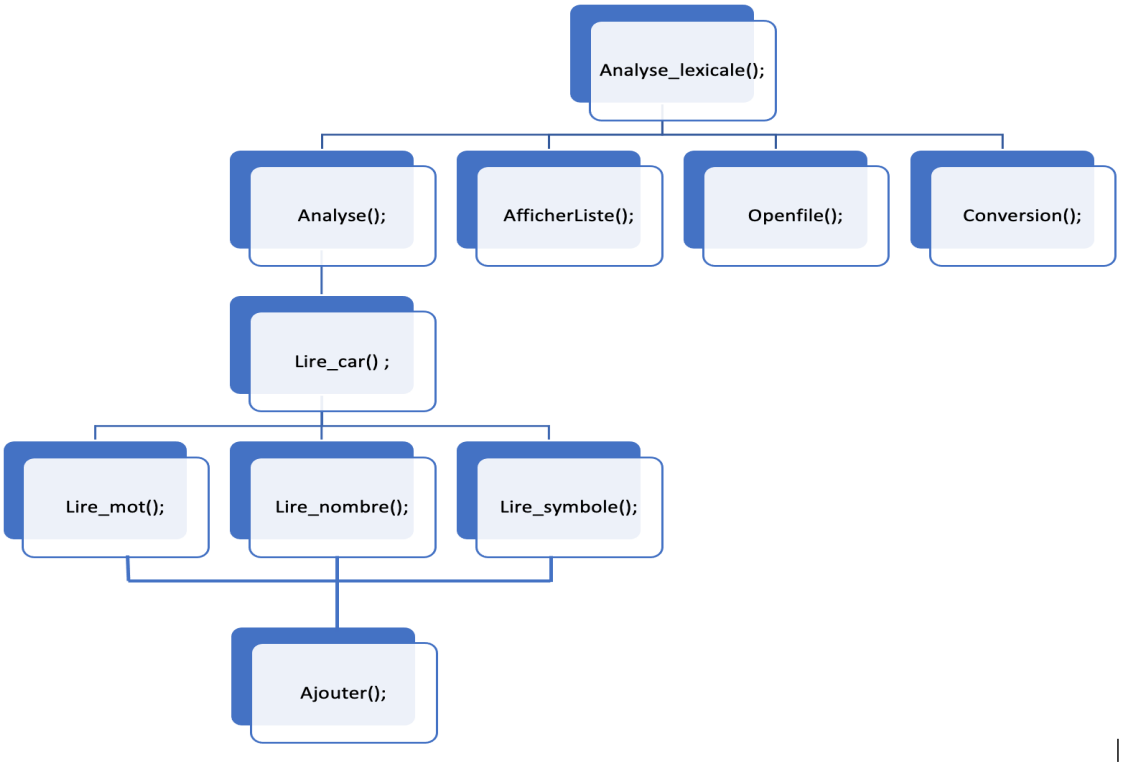

Figure 4 : FONCTIONS D'ANALYSEUR LEXICALES

III- ANALYSEUR SYNTAXIQUE

Soit une grammaire G et S son axiome définit par G=<T,NT,S,RP> susmentionné dans la partie précédente. Le langage engendré par la grammaire proposée peut donc être défini en utilisant des dérivations successives de l'axiome de la grammaire.

1. ARBRE SYNTAXIQUE

L'analyseur syntaxique généré fait la validation grammaticale du programme ou du texte d'entrée. Pour ce faire, il se base sur les règles grammaticales spécifiées dans la grammaire. Comme précédemment mentionné, une règle grammaticale correspond à une alternative d'une production.

L'analyseur syntaxique ne lit pas directement le fichier en entrée. Il reçoit un flot d'unités lexicales par l'intermédiaire de l'analyseur lexical et c'est à partir de ces unités lexicales que sont appliquées les règles de validation.

Afin de pouvoir illustrer un arbre syntaxique pour notre grammaire nous allons prendre en charge un exemple minimisé du notre langage comme suit :

start_prog
declare_var (type id s') ;
start type Function id ( ) inst ; return id ; end id
start type Main_function ( ) inst ; return id ; end Main_function
ENDP

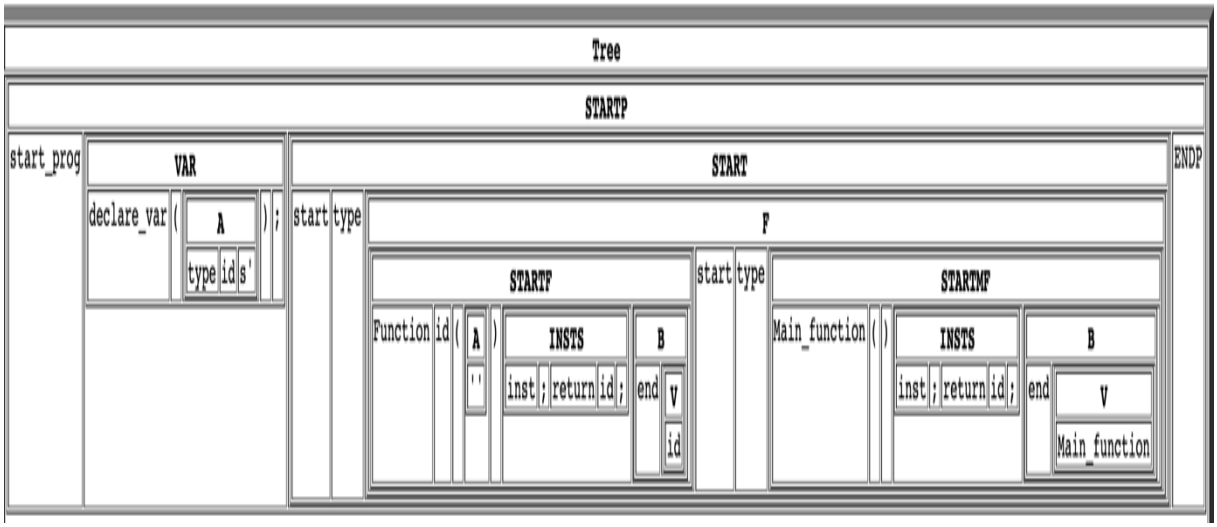

Figure 5 : ARBRE SYNTAXIQUE

Pour une phrase donnée, l'arbre syntaxique équivalent peut être associé à une dérivation particulière de l'axiome de la grammaire qui est constituée d'une ou de plusieurs séquences de remplacement. À partir de cette séquence de remplacement, on peut déduire la forme générale de l'arbre syntaxique ainsi que les différents nœuds et feuilles qui la composent. Il est assez facile de voir que l'arbre syntaxique de la figure 5 est construit en utilisant la dérivation suivante:

STARTP -> start_prog VAR START ENDP
VAR -> declare_var (A) ;
VAR -> ''
A -> type id s'
A -> ''
START -> start type F
F-> STARTF start type STARTMF
STARTF -> Function id (A) INSTS B
STARTF -> ''
STARTMF -> Main_function ( ) INSTS B
INSTS -> inst ; return id ;
B -> end V
V -> id
V -> Main_function

2. LES FONCTIONS DU PARSER

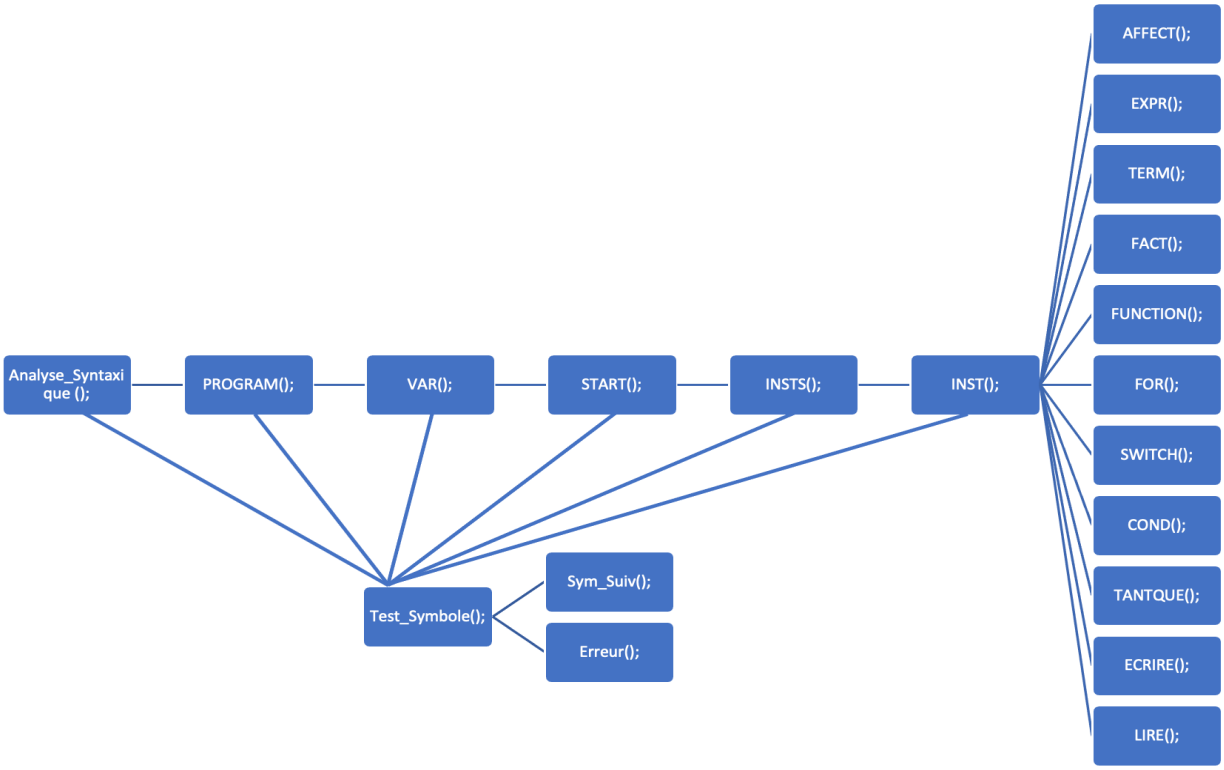

Figure 6 : FONCTIONS DU PARSER

- Analyse_syntaxique() ;
- PROGRAM () ;
- Test_Symbole() ;
- Sym_Suiv() ;
- Erreur() ;
- VARS() ;
- START() ;
- INSTS() ;
- INST() ;
- AFFEC() ;
- EXPR() ;
- TERM() ;
- FACT() ;
- FUNCTION() ;
- FOR() ;
- SWITCH() ;
- SI() ;
- CHOICE() ;
- COND() ;
- TANTQUE() ;
- ECRIRE() ;
- LIRE() ;

CONCLUSION

Dans le cadre du projet de compilation on a décidé de proposer un langage synthétisant la programmation impérative et celle de la description matériel et de réaliser un analyseur lexical et un parser pour l'analyse syntaxique.

Afin de mettre en œuvre ce projet, on a passé par plusieurs phases, de la collecte des informations à la définition des symboles passant par et la conception des règles de production et enfin la réalisation des deux analyseurs.

On a pu respecter le cahier des charges de ce projet en incluant toutes les fonctionnalités demandées. Globalement ce projet a donc été une très bonne occasion pour consolider nos connaissances en matière de programmation par le langage c et particulièrement nous a permet de comprendre parfaitement la chaine de compilation d'un code. Durant le travail sur ce projet, on a appris qu'une bonne répartition du temps et celle des tâches sont essentielles, ainsi qu'une analyse complète et détaillée est indispensable pour la réussite de ce genre de projet.

La réalisation du projet dans sa totalité présente plusieurs possibilité d'amélioration qui seront de l'ordre d'une amélioration des fonctions prédéfinies par insertion des bibliothèques dédiées pour cette fin, d'une élaboration des règles sémantiques pour offrir un compilateur complet pour le langage et enfin d'une génération de code.

LISTE DES TABLEAUX

LISTE DES FIGURES