Université Mohamed V
Ecole Nationale Supérieure d'Informatique et d'Analyse des Systèmes - RABAT

Rapport du projet de complition

Conception et réalisation d'un mini compilateur C pour un langagee de programmation VHDL

Filière : Ingénierie des Systèmes Embarqués et Mobiles

Réalisé par :

AIT JA Abdelilah

BENALI Hajar

BENHAMMOU Nouhayla

FTERI MGHARI Khadija

KOUKOUS Saad

Encadré par :

M.TABII Younes

Année universitaire 2020/2021

Remerciements

En premier lieu, nous tenons à remercier notre encadrant Mr Tabii Younes pour son encadrement, pour sa disponibilité et pour la confiance qu'il nous a accordé pour élaborer ce projet librement.

Nous voudrions ensuite exprimer notre reconnaissance envers notre chef de filière Mr.El Bouannani Fayçal pour son soutien continu et ses encouragements acharnés.

Nous souhaitonsenfin adresser nos remerciements à toute personne ayant contribué de près ou de loin a l'acheminement de ce travail par toute intervention et toute remarque.

Résumé

Comme tout langage naturel écrit, un langage de programmation se compose de lettres, de nombres et de caractères spéciaux obéissant à des règles de syntaxe et de sémantique. Un langage est destiné à décrire l'ensemble des actions consécutives qu'un ordinateur doit exécuter.Pour ce faire, un ordinateur à besoin d'un compilteur pour traduire du langage de programmation utilisé par l'utilisateur au langage machine.

Notre projet consiste à créer un mini compilateur d'un language de programmation hardware COMHDL. Notre langage est un langage innovant, créatif et facile à manipuler.Il permet de coder le bas niveau en respectant la meme sémantique du VHDL mais avec une nouvelle syntaxe moins complexe. Ce document a pour but de décrire le déroulement de notre projet qui vise à créer un compilateur pour ce langage. Nous vous présenterons donc tout au long du rapport, les étapes de la réalisation de ce projet, ainsi que les outils que nous avons utilisé pour réaliser notre projet. Notre compilateur est basé sur le langage C et a pour objectif de faire une analyse lexicale,syntaxique et sémantique du code VHDL. Au premier lieu, nous allons présenter le contexte général du projet , ensuite nous attaquerons la conception , et vers la fin de ce document nous aborderons la réalisation du projet.

Mot clés : **Compilation,Langage de programmation, Grammaire, Analyseur lexical,Analyseur syntaxique,Analyseur sémantique .**

Abstract

Like any written natural language, a programming language consists of letters, numbers and special characters obeying rules of syntax and semantics. One language is intended to describe the set of consecutive actions that a computer must perform. To do this, a computer needs a compiler to translate from the programming language used by the user to the machine language.

Our project consists in creating a mini compiler of a hardware programming language VHDL. The purpose of this document is to describe the progress of our compilation project. We will therefore present to you throughout the report, the stages of the realization of this project, as well as the tools we used to carry out our project. Our compiler is based on the C language and aims to perform a lexical, syntactic and semantic analysis of VHDL code. First, we will present the general context of the project, then we will attack the design, and towards the end of this document we will approach the realization of the project.

Keywords : **Compilation, Programming language, Grammar, Lexical analyzer, Syntactic analyzer, Semantic analyzer.**

Table des matières

Table des figures

Introduction générale

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.

Ce rapport est donc élaboré pour montrer les importantes étapes surpassées pour réaliser ce projet, il est donc structuré comme suit :

• Le chapitre 1 intitulé **" Contexte générale du projet "** donnera une vue général sur le langage crée ainsi que son compilateur.Lors de cette partie nous allons voir les objectifs de notre projet ainsi que les problématiques qu'il traite.

• Le chapitre 2 intitulé **" Analyse et conception "** est consacré à l'étude effectuée et la démarche adoptée dans les principales tâhes.Cette partie donne une vue approfondie du plan de travail et les différentes fonctionnalitées.

• Le chapitre 3 nommé **"Réalisation"** expose dans une première section les outils utilisés dans l'élaboration de ce projet tandis que la deuxième section présentera les differentesétapes pour réaliser ce mini compilateur.

• Dans la **conclusion** on va mettre le point sur l'apport de notre compilateur, ses limites et les perspectives qu'il peut engendrer.

Chapitre 1

Contexte général du projet

1.1 Introduction

Dans ce chapitre nous allons définir le contexte général du projet et ses objectifs.

1.2 Amenant

La compilation, est l'une des compétences fondamentales d'un ingénieur en informatique, puisque il permet de comprendre comment les langages de programmation fonctionnent, et toute la théorie derrière. En outre, apprendre à créer un compilateur d'un langage élevé va nous aider un jour à créer notre propre langage de programmation.

1.3 Problématique

Le but d'un langage de description matériel est de faciliter le développement d'un circuit numérique en fournissant une méthode rigoureuse de description du fonctionnement et de l'architecture du circuit désiré. Le défit est de développer un langage de programmation combinant entre la structure du langage VHDL et la logique des langages de programmation enseignés durant les deux années, ainsi qu'un compilateur pour l'analyse lexicale et syntaxique.

1.4 Objectifs du projet

Le projet consiste à réaliser un compilateur du langage VHDL , en respectant les différentes consignes présentées dans le cahier de charges :

— Présenter le lexique et la garmmaire.
— Rendre la grammaire LL(1).
— Programmer un analyseur lexial.
— Programmer un analyseur syntaxique.
— Introduire les règles sémantiques.
— Programmer un analyseur sémantique.
— Générer un code intermédiaire.

1.5 Le langage ComHDL

Le butd'un langage de description matériel tel que le VHDL est de faciliter le développement d'un circuit numérique en fournissant une méthode rigoureuse de description du fonctionnement et de l'architecture du circuit désirée. L'idée est de ne pas avoir à réaliser un composant réel, en utilisant à la place des outils de développement permettant de vérifier le fonctionnement attendu.

Un langage de programmation combinant à la fois la structure du VHDL et la logique des langages de programmation comme C, C++,... peut faciliter la tâche aux débutants et aux personnes habutués à programmer avec ces langages.

1.6 Planification du projet

Afin de réaliser le projet dans les délais établis, nous avons utilisé le diagramme de GANTT puisqu'il est considéré comme l'un des outils les plus efficaces pour représenter l'état d'avancement des différentes tâches . Comme le montre la figure ci-dessous, nous avons adopté le modèle en cascade pour réaliser notre projet. Ce dernier représente une organisation des activités sous forme de phases linéaires et séquentielles, où chaque phase correspond à une spécialisation des tâches et dépend des résultats de la phase précédente.

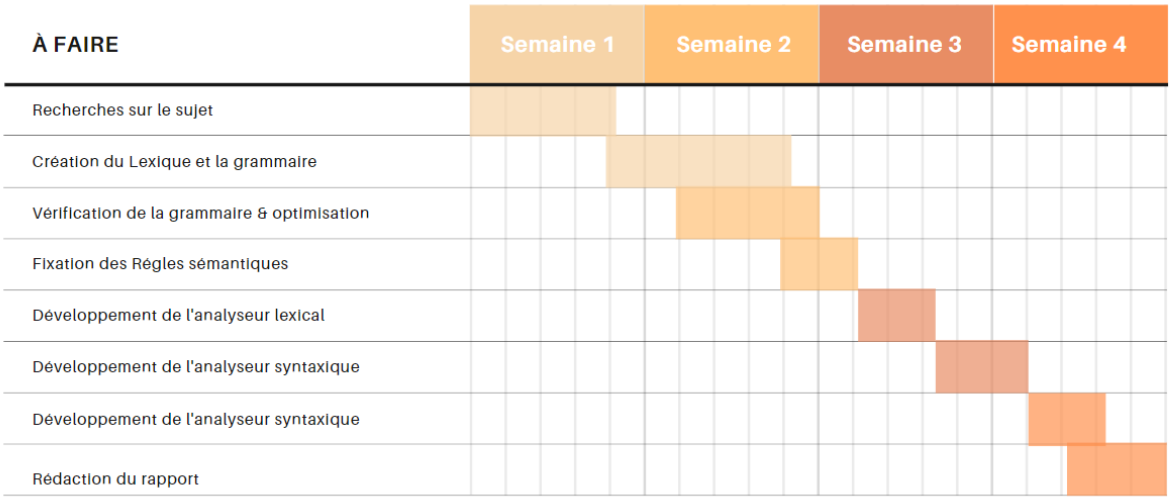

FIGURE 1.1 – Diagramme de GANTT

1.7 Conclusion

Dans ce chapitre nous avons abordé différentes parties. Premièrement, nous avons expliqué la problématique et préciser l'objectif du projet, ensuite nous avons présenté notre langage à develloper, ainsi que la planification suivi. Le deuxième chapitre sera dédié à l'analyse et conception.

Chapitre 2

Conception Modélisation

2.1 Introdution

Ce chapitre présente l'étude de l'existant avec une analyse et description des besoins, ainsi que les étapes établies pour réaliser le projet.

2.2 Lexique

2.2.1 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

2.2.1.1 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

FIGURE 2.1 – Tableau des opérateurs de calcul

2.2.1.2 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 ouégal 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 ouégal 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

FIGURE 2.2 – Tableau des opérateurs de comparaison

2.2.1.3 Les opérateurs logiques :

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 (retournela valeur 1 si la variable vaut 0, 0 si elle vaut 1)	not(a)

FIGURE 2.3 – Tableau des opérations logiques

2.2.2 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**

2.2.3 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 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

2.2.4 Les boucles

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 :

2.2.4.1 Boucle FOR :

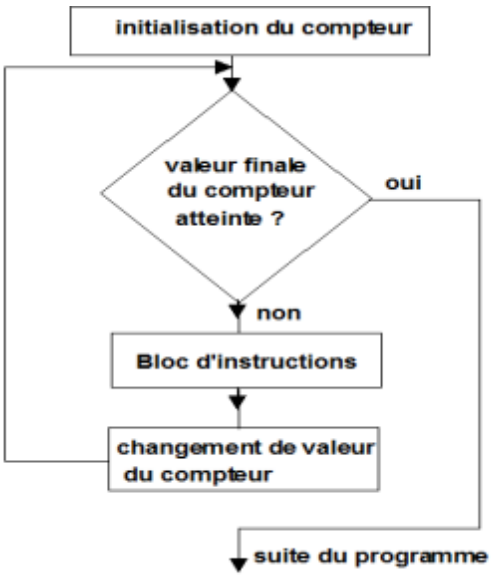

FIGURE 2.4 – La boucle FOR

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

2.2.4.2 La boucle while :

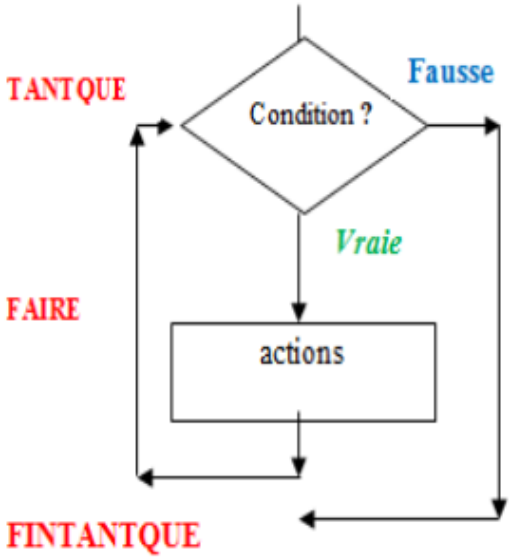

FIGURE 2.5 – La boucle while

Pour cette boucle la syntaxe proposée est la suivante :
While condition **then**
instruction 1 ;
instruction 2 ;
.
.
instruction n ;
end while

2.2.5 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 transmettreune 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) ;

2.3 Grammaire

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

2.3.1 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 nonterminaux 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

2.3.2 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

2.3.3 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

2.4 Régeles sémantiques

2.4.1 LES REGLES DE PRODUCTION (RP)

Voici quelque règles que nous avons utilisées

STARTP::=	**start_prog VAR START ENDP**	**CHAINE::=**	ID CHAINE | epsilon
VAR::=	Declare_Var (A); |epsilon	**RELOP::=**	= | <> | < | > | <= | >=
A::=	TYPE ID S’ |epsilon	**ADDOP::=**	=+| -
TYPE::=	int | float | char | bit | void	**TERM::=**	FACT C
S’::=	<- NUM ; A |; A | , ID S’	**C::=**	MULOP FACT |epsilon
START::=	start TYPE F	**MULOP::=**	* | /
F::=	STARTF start TYPE STARTMF | STARTMF	**FACT::=**	ID | NUM | (EXP)
STARTF::=	function ID (ARG) INSTS B	**PREDEF::=**	not | floor |and |or | mod
ARG::=	TYPE ID ARG | ,TYPE ID ARG | epsilon	**ID::=**	LETTRE Q | PREDEF
STARTMF::=	Main_function ( ) INSTS B	**Q::=**	LETTRE Q | CHIFFRE Q |epsilon
INSTS::=	INST; return ID;	**LETTRE::=**	a |…|z |A|…|Z
INST::=	FUNCTION|AFFEC| SI | TANTQUE | FOR | ECRIRE | LIRE | SWITCH | epsilon	**NUM ::=**	CHIFFRE NUM | epsilon
B::=	end V	**CHIFFRE::=**	0|1|...|9
V::=	ID | Main_function	**COND::=**	EXP RELOP EXP
ENDP::=	end_prog	**CHOICE::=**	elsif COND then INST | else INST |epsilon
FUNCTION::=	(R)	**G::=**	case FACT: I NST ; break; G| epsilon
R::=	EXP R |, EXP R | epsilon	**T::=**	EXP K | K EXP
AFFEC::=	ID <- EXP	**K::=**	‘’ CHAINE ’’ K | , T
SI::=	if COND then INST CHOICE end if	**CHAINE::=**	ID CHAINE | epsilon
TANTQUE::=	while COND then INST end while	**RELOP::=**	B98= | <> | < | > | <= | >=
FOR::=	for (ID <- FACT; COND; ID <-EXP) then INST end for	**ADDOP::=**	$^^=+|-
ECRIRE ::=	print_out (T)	**TERM::=**	FACT C
LIRE ::=	put_in (ID)	**C::=**	MULOP FACT |epsilon
SWITCH ::=	switch (ID) G default: INST; break; end switch	**MULOP::=**	* | /
EXP::=	TERM ADDOP TERM |TERM	**FACT::=**	ID | NUM | (EXP)
COND::=	EXP RELOP EXP	**PREDEF::=**	not | floor |and |or | mod
CHOICE::=	elsif COND then INST | else INST |epsilon	**ID::=**	LETTRE Q | PREDEF
G::=	case FACT: I NST ; break; G| epsilon	**Q::=**	LETTRE Q | CHIFFRE Q |epsilon
T::=	EXP K | K EXP	**LETTRE::=**	a |…|z |A|…|Z
K::=	‘’ CHAINE ’’ K | , T	**NUM ::=**	CHIFFRE NUM | epsilon
		CHIFFRE::=	0|1|...|9

FIGURE 2.6 – Tableau des régles de production

2.4.2 VERIFICATION DE LA GRAMMAIRE (LL1)

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											

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											

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																

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

FIGURE 2.7 – Vérification de la grammaire par table d'analyse

2.4.3 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 2.8 – EXEMPLE DU LANGAGE

2.5 Conclusion

Cette partie a pour objectif de présenter l'ensemble des règles lexicales, syntaxiques et sémantiques utilisées dans le développement de notre compilateur.Il

s'agit de la conception de notre compilateur et son mode de fonctionnement interne.

Chapitre 3

Réalisation

3.1 Introduction

Dans ce chapitre nous présenterons les diffèrents outils utilisés dans la réalisation du projet. Puis nous allons expliquer les différents analyseurs implémentés.

3.2 Outils de développement

Langage C

L'une des raisons très fortes pour lesquelles le langage de programmation C est si populaire et utilisé si largement est la flexibilité de son utilisation pour la gestion de la mémoire. Les programmeurs ont la possibilité de contrôler comment, quand et où allouer et désallouer la mémoire. La mémoire est allouée de manière statique, automatique ou dynamique dans la programmation C à l'aide des fonctions malloc et calloc. Initialement, C a été conçu pour implémenter le système d'exploitation Unix. La plupart du noyau Unix, et tous ses outils et bibliothèques de support, ont été écrits en C. Plus tard, d'autres personnes l'ont trouvé utile pour leurs programmes sans aucune entrave, et ils ont commencé à l'utiliser. Une autre bonne raison d'utiliser le langage de programmation C est qu'il se trouve à proximité du système d'exploitation. Cette fonctionnalité en fait un langage efficace car les ressources au niveau du système, telles que la mémoire, sont facilement accessibles, ce qui nous a facilité de créer un compilateur.

3.3 Développement du compilateur

3.3.1 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.

3.3.1.1 La 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 :

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

FIGURE 3.1 – La tables des mots clés

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

FIGURE 3.2 – La table des symboles spéciaux

Les symboles erronnées :

LES SYMBOLES ERRONEES	
LE RESTE	ERREUR_TOKEN

FIGURE 3.3 – La table des symboles erronnées

3.3.1.2 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 :

3.3.2 Analyseur syntaxique

L'analyseur syntaxique vérifie si une expression appartient à la grammaire, si une expression dans le code n'appartient pas à la grammaire qu'in défini, une erreur syntaxique est déclenchée . D'abord, le code pour l'analyseur syntaxique est l'ensemble des procédures récursives, tel que chaque procédure représente une règle syntaxique. Juste comme vu dans le TP la procédure associée à l'axiome constitue le programme principal. C'est elle qui est appelée la première fois et celle qui appelle les autres, c'est la fonction Debut() dans notre programme.

3.3.2.1 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

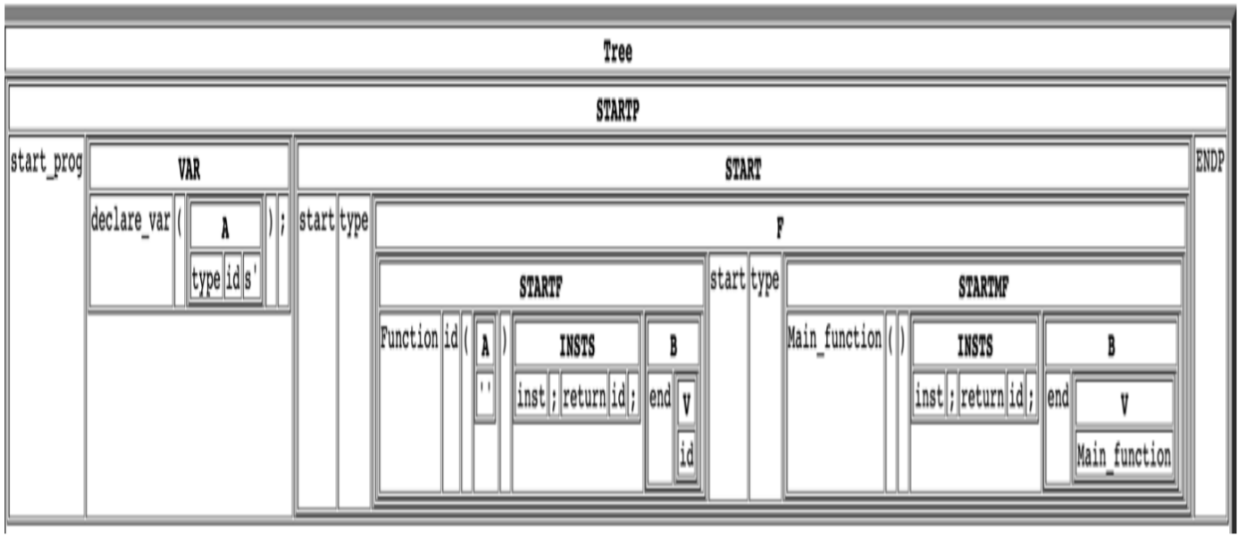

Figure 3.4 – ARBRE SYNTAXIQUE

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

3.3.2.2 LES FONCTIONS DU PARSER

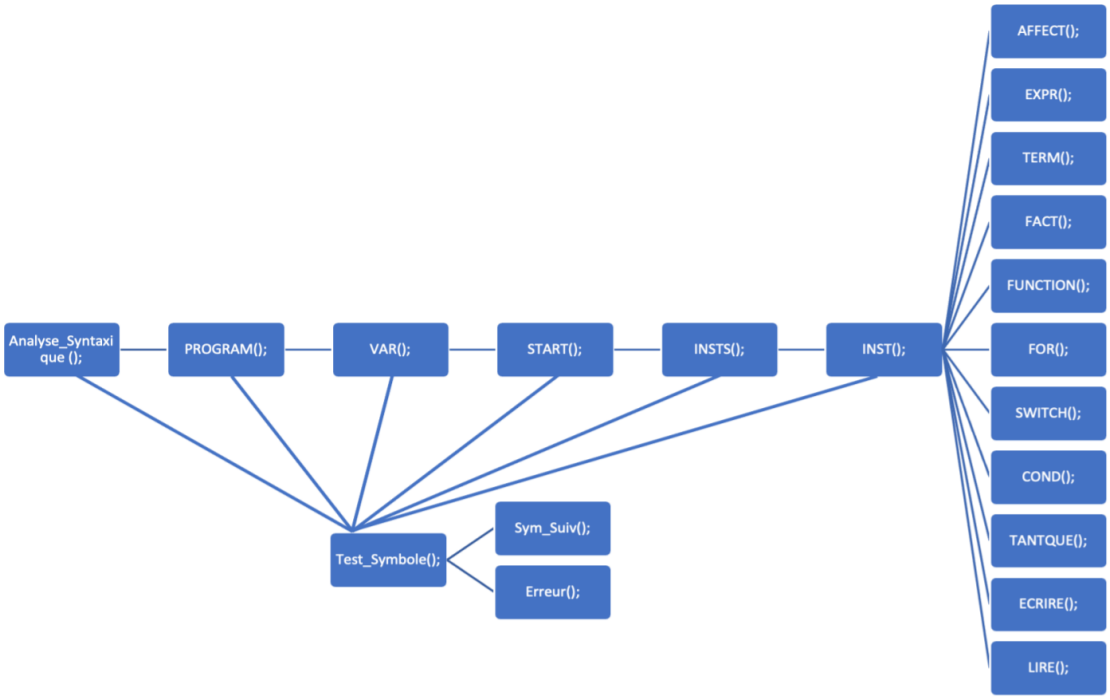

FIGURE 3.5 – Fonctions du Parser

- 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() ;

3.4 Conclusion

Ce chapitre met en exergue le développement de notre compilateur.Nous avons exploité le langage C pour ce faire. Nous avons aussi usé de notre lexique et des régles de production précedemment fixées.
Notre compilateur ComHDL a été rélisé suivant une analyse lexicale suivie d'une analyse syntaxique.Nous voudrions entamé la dernière étape qui est l'analyse sémantique, mais faute de temps nous ne pouvions pas le faire .

Conclusion générale

En définitive, 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.

Ce projet fait aussi appel à un véritable travail de réflexion sur la manière de le concevoir pour remplir les besoins inscrites dans le cahier de charges. Cela nous a permis de comprendre les difficultés d'un projet entre les choses que nous voulions réaliser et les contraintes que nous devrions surmonter.

Bibliographie

[1] data transition numerique. Panda python : Maîtrisez l'analyse des données avec python.
https://www.data-transitionnumerique.com/panda-python/, 2021.

[2] Futura-Sciences. Css : qu'est-ce que c'est ?
https://www.futura-sciences.com/tech/definitions/internet-css-4050/, 2001-2021.

[3] Pierre Giraud. Présentation de bootstrap.
https://www.pierre-giraud.com/bootstrap-apprendre-cours/introduction/, date de consultation : 03/08/2021.

[4] informatique news. Python, java, c : tiercé gagnant des langages de programmation.
https://www.informatiquenews.fr/python-java-c-tierce-gagnant-des\protect\@normalcr\relax-langages-de-programmation-63252, 2019.

[5] JDN. Html (hypertext markup langage) : définition, traduction.
https://www.journaldunet.fr/web-tech/dictionnaire-du-webmastering/1203255-html-hypertext-markup-langage-definition-traduction/, 2019.

[6] JDN. Javascript : définition simple et applications pratiques.
https://www.journaldunet.fr/web-tech/dictionnaire-du-webmastering/1203585-javascript/, 2020.

[7] python.doctor. Présentation de django.
https://python.doctor/page-django-introduction-python, date de consultation : 03/08/2021.

[8] stat4decision. Faire une régression logistique avec python.
https://www.stat4decision.com/fr/faire-une-regression-logistique\protect\@normalcr\relax-avec-python/, 2021.

[9] RIP Tutorial. Apprenez beautifulsoup. *RIP Tutorial*.