ENSIAS

جامعة محمد الخامس بالرباط
Université Mohammed V de Rabat

Rapport de Projet Compila

conception d'un langage et réalisation d'un compilateur avec le langage c

Préparé par :

- ***MOSLEH ismail***
- ***LYAZRHI Mohammed Amine***
- ***MOFLIH Manal***
- ***MESTADI Ibrahim***
- ***MANSOURI Ayoub***

Encadre par :
Pr.TABII Younes

Membres de Jury :

-
-

Filière Génie Logiciel
Année universitaire : 2020/2021

Remerciements

Avant d'aborder la description des parties importantes du projet, nous aimerons tout d'abord exprimer notre gratitude intense à toute personne qui a contribué énormément dans l'élaboration et la réalisation de ce travail. on commence ainsi par offrir nos remerciements à l'intégralité des personnes travaillant au sein de l'école Nationale supérieure d'informatique et d'analyse des systèmes. Toute nos gratitude et profonde reconnaissance s'adressent à notre encadrant de projet ***TABII*** *Younes, et à toute personne qui a contribué dans la réalisation de ce projet. on vous remercie énormément pour l'encadrement de qualité garanti pour bien mener et assurer la réalisation de ce travail dans les meilleures des conditions.*

Résumé

Ce rapport présente le projet que nous avons réalisé en groupe pour le module de compilation de la deuxième année à l'ENSIAS. Le but de ce projet est de faire la conception d'un langage de programmation, qu'on a nommé ALGO, et la réalisation d'un simple compilateur (analyse lexicale et syntaxique) pour ce langage. Dans ce rapport, nous allons présenter dans un premier temps le langage de programmation et les cas d'utilisation potentiels. Ensuite, nous allons nous intéresser à la présentation de notre grammaire LL (1). et après on va présenter le fonctionnement de l'analyseur lexicale et l'analyseur syntaxique.

Mots clés : ALGO , syntaxique, lexicale, compilateur

Abstract

This report presents the project that we carried out as a group for the compilation module of the second year at ENSIAS. The goal of this project is to design a programming language, which we named ALGO, and the realization of a simple compiler (lexical and syntactic analysis) for this language. In this report, we will first present the programming language and potential use cases. Next, we will focus on the presentation of our LL(1) grammar, and then we will present the functiones of the lexical analyzer and the parser analyzer.

Keywords : ALGO , parser, lexical, compiler

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

La première partie est consacrée à la présentation 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.
La troisième partie est destinée à la présentation de l'analyseur syntaxique, en particulier l'arbre syntaxique et les fonctions de ce PARSER.

Chapitre 1

Constitution des éléments du langage

1.1 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.**

1.1.1 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 10)
+	opérateur d'addition	Ajoute deux valeurs	x+3	13
-	opérateur de soustraction	Soustrait deux valeurs	x-3	7
*	opérateur de multiplication	Multiplie deux valeurs	x*3	30
/	opérateur de division	Divise deux valeurs	x/3	3.3333333
<-	opérateur d'affectation	Affecte une valeur à une variable	x<-3	x prend la valeur 3

1.1.2 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 ou égal à 3, sinon 0
>	opérateur de supériorité stricte	Vérifie qu'une variable est strictement superieure à une valeur	x>3	Retourne 1 si x est superieur 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 supériorité	Vérifie qu'une variable est différente d'une valeur	X<>3	Retourne 1 si x est différent de 3, sinon 0

1.1.3 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
|	OU logique	Vérifie qu'une des conditions est réalisée	a|b
&	ET logique	Vérifie que toutes les conditions sont réalisées	A&b
!	NON logique	Inverse l'état d'une variable booléenne	!a

1.2 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 1.
Voici la forme de base d'une déclaration des variables :

TYPE variable [<-value], variable [<-value] ... ;

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

entier, reel , caract, chaine,bit.

1.3 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 :

si condition **alors :**

instructions ;

si$_{autrecondition}$**alors :**

instructions ;

.

.

sinon

instructions ;

fin si

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

option (id)

cas expression 1 :instructions ; stop ;

cas expression 2 :instructions ; stop ;

cas expression 3 :instructions; stop;

.

.

cas expression n :instructions; stop;

default :instructions; stop;

fin Option

1.4 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 :

1.4.1 Boucle POUR

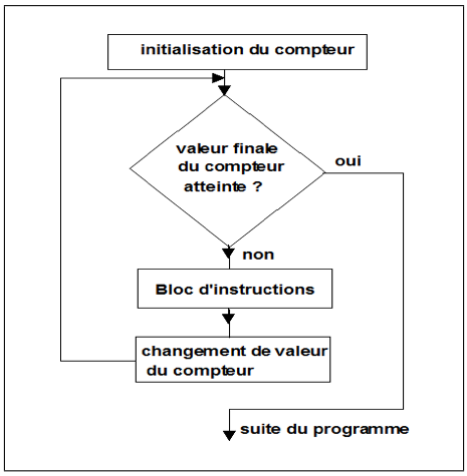

FIGURE 1.1 – Boucle POUR

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

pour i dans <a,b,pas> :

instruction 1;

instruction 2;

.

instruction n;

fin pour

1.4.2 Boucle TANT QUE

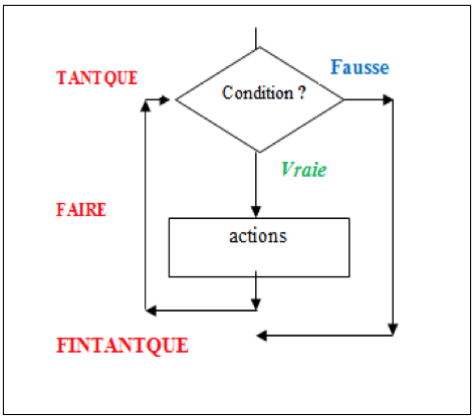

FIGURE 1.2 – Boucle TANT QUE

Pour cette boucle la syntaxe proposée est la suivante :

tantque condition Faire

instruction 1;

instruction 2;

.

.

instruction n;

fin tantque

1.5 Les instructions de saisie

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 Ecrire() pour transmettre une valeur à l'extérieur .

Et pour transmettre une valeur venant de l'extérieur nous aurons la fonction Lire().

Exemple :

ecrire("hello world",a);

lire(a);

1.6 La grammaire

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

1.6.1 Les non-terminaux

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= [DEBUT, FIN, FONCTION, MAIN, FAIRE, SI, SINON, SIAUTRE, ALORS, OPTION,CAS, STOP, DEFAULT, TANTQUE, POUR,DANS, ECRIRE, LIRE, DONNER,TYPE,BIT, REEL, ENTIER, CARACT, CHAINE,PV, PT, PLUS, MOINS, MULT, DIV, VIR, EG, AFF, INF, INFEG, SUP, SUPEG,DIFF, PO, PF, PP, ET, OR,NON,COMMENT, ESPACE, SAUT, ERREUR, ID, NUM, EOF_TOKEN,]

1.6.2 Les terminaux

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

T =[entier, reel , caract , chaine ,bit ,<-, =, <>,>=,<=, ;,+, -, /,*, !, ,| , ecrire, lire, (,), debut, fonction,main, donner, fin, si, alors, tantque, pour , faire , si_autre , option , default, stop, cas,sinon, ", a…z, A…Z, 0…9]

1.6.3 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 -> DEBUT

1.6.4 Les regles de production(RP)

DEBUT	<–	debut VAR FONCTIONS MAIN fin
VAR	<–	TYPE ID S' | epsilon
TYPE	<–	entier|caract|chaine|bit|reel
ID	<–	LETTRE ID'
ID'	<–	LETTRE|CHIFFRE|epsilon
LETTRE	<–	a...z|A...Z
NUM	<–	CHIFFRE|CHIFFRE NUM
CHIFFRE	<–	0...9
S'	<–	;VAR | ,ID S' | <-NUM S"|<-CHAINE S"|epsilon
S"	<–	;VAR| ,ID S'
FONCTIONS	<–	FONCTION FONCTIONS |epsilon
FONCTION	<–	debut TYPE ID (ARG) :INSTS fin ID
MAIN	<–	debut main() :INSTS fin main
ARG	<–	TYPE ID,ARG | ARG |epsilon
INSTS	<–	INST INSTS
INST	<–	AFFEC | SI | TANTQUE | ECRIRE | LIRE | epsilon
AFFEC	<–	ID <- EXPR
SI	<–	si COND alors :INSTS CHOIX fin si
CHOIX	<–	AUTRE CHOIX |sinon : INSTS | epsilon
AUTRES	<–	si_autre COND alors : INSTS
TANTQUE	<–	tantque COND faire : INSTS
ECRIRE	<–	ecrire(CHAINES)
CHAINES	<–	"ID"|"ID",CHAINES|""
EXPRS	<–	EXPR EXPRS
LIRE	<–	lire(IDS)
IDS	<–	ID ,IDS|ID
COND	<–	EXPR RELOP EXPR
RELOP	<–	= | <> | < | > | <= | >=
EXPR	<–	TERM ADDOP TERM
ADDOP	<–	+ | - | & | | |!
TERM	<–	FACT MULOP FACT
MULOP	<–	* | /
FACT	<–	ID | NUM | (EXPR)

1.7 Exemple d'un programme

```
debut
    entier a<-112 ,b<-382;
    reel c<-112.44,d<-11.1;
    bit e<-0, f<-1,g;
    caract l<-'l';chaine s<-"abcd";
    #commentaire::yytzg#
    debut entier fonction add(entier a,entier b) :
        c <- a+b;
        donner c;
    fin fonction add
    debut main():
        d <- fonction add(a,b);
        si d<>0 alors :
            ecrire(d, " : est pair ");
        si_autre (a=b) alors :
            ecrire("a egal b");
        sinon :
            ecrire(d, " : est impair ");
        fin si
        c<-8;
        tantque c<10 faire :
            c<-c+1;
            ecrire(c,"\n");
        fin tantque

        option (b)
            cas 1 : ecrire(" Bienvenue,donner un entier");  stop;
            cas 2 : lire(a);
                ecrire( "vous avez donné ",a);
                stop;
            default : ecrire( "fin \n");
        fin option

        g<- (!e)&f;
        ecrire(g);
        ecrire("Fin  programme");
        #! : NON#
    fin main
fin
```

FIGURE 1.3 – Simple code

Chapitre 2

Analyseur Lexical

Le rôle de l'analyseur lexical consiste à valider le texte lexicalemnt 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.

2.1 La Structure Tokens

nous avons divisé les types des Tokens en trois parties :

- **les mots clés de notre langage.**
- **les mots spéciaux.**
- **les autres tokens.**

```
typedef enum
{
    //MOTS CLES
    DEBUT, FIN, FONCTION, MAIN, FAIRE, SI, SINON, SIAUTRE, ALORS, OPTION,
    CAS, STOP, DEFAULT, TANTQUE, POUR,DANS, ECRIRE, LIRE, DONNER,TYPE,
    BIT, REEL, ENTIER, CARACT, CHAINE,
    //MOTS SPECIAUX
    PV, PT, PLUS, MOINS, MULT, DIV, VIR, EG, AFF, INF, INFEG, SUP, SUPEG,
    DIFF, PO, PF, PP, ET, OR,NON,
    //AUTRES
    COMMENT, ESPACE, SAUT, ERREUR, ID, NUM, EOF_TOKEN,
} TOKENS;
```

FIGURE 2.1 – Structure des Tokens

2.2 La Liste des Tokens

pour enregistrer la liste des mots on a utilisé une liste chaînée sous la forme suivante :

```
typedef struct Structure_TOKEN
{
    TOKENS token;
    char* nom;
    int val;
    struct Structure_TOKEN* next;
} Structure_TOKEN;
```

FIGURE 2.2 – Liste des Tokens

2.3 La liste des fonctions

L'analyse lexicale commence par appeler la fonction analyse_lexicale() qui fait toutes les étapes pour vérifier le lexique du programme.

Les étapes de notre analyse lexicale :

- **analyse_lexicale();**
    - **Open_file();**
    - **Lire_Car();**
    - **analyse();**
        - **Lire_mot();**
        - **Lire_nombre();**
        - **Lire_Symbole();**
        - **ajouter();**
    - **AfficherListe();**

```
/* fonction d'analayse lexicale */
int analyse_lexicale() {…
/* lecture des caractères à partir du fichier */
char Lire_Car() {…
/* verificaion d'un mot */
TOKENS lire_mot() {…
/* test d'un nombre */
TOKENS lire_nombre() {…
/* test d'un symbole */
TOKENS lire_Symbole() {…
/* test du type du mot à analyser*/
TOKENS analyse() {…
/* ouverture du fichier */
void openfile(char* filename) {…
/* ajout des resultats de test dans la liste chainée */
void ajouter() {…
/* affichage de la liste chainée et écriture sur le fichier résultat */
void AfficherListe() {…
/* conversion des valeurs du token créé en chaine de caractère */
char* conversion(TOKENS token) {…
```

FIGURE 2.3 – La liste des fonctions lexicales

2.4 Résultats de l'analyseur lexicale

```
================================================================================
============================LISTE DES TOKENS============================
================================================================================
TOKEN 0 :  debut ==> debut
TOKEN 1 :  retour ligne ==> SAUT
TOKEN 2 :  entier ==> type
TOKEN 3 :  a ==> ID
TOKEN 4 :  <- ==> AFF
TOKEN 5 :  112 ==> entier
TOKEN 6 :  , ==> VIR
TOKEN 7 :  b ==> ID
TOKEN 8 :  <- ==> AFF
TOKEN 9 :  382 ==> entier
TOKEN 10 :  ; ==> PV
TOKEN 11 :  retour ligne ==> SAUT
TOKEN 12 :  reel ==> type
TOKEN 13 :  c ==> ID
TOKEN 14 :  <- ==> AFF
TOKEN 15 :  112.44 ==> reel
TOKEN 16 :  , ==> VIR
TOKEN 17 :  d ==> ID
TOKEN 18 :  <- ==> AFF
TOKEN 19 :  11.1 ==> reel
TOKEN 20 :  ; ==> PV
TOKEN 21 :  retour ligne ==> SAUT
TOKEN 22 :  bit ==> type
TOKEN 23 :  e ==> ID
TOKEN 24 :  <- ==> AFF
TOKEN 25 :  0 ==> bit
TOKEN 26 :  , ==> VIR
TOKEN 27 :  f ==> ID
TOKEN 28 :  <- ==> AFF
TOKEN 29 :  1 ==> bit
TOKEN 30 :  , ==> VIR
TOKEN 31 :  g ==> ID
```

FIGURE 2.4 – Résultats de l'analyseur lexicale

Chapitre 3

Analyseur 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.

3.1 Arbre syntaxique

Afin de pouvoir illustrer un arbre syntaxique pour notre grammaire nous allons prendre en charge un exemple d'écrire "hello world" :

```
debut
    debut main() :
        ecrire("hello world");
    fin main
fin
```

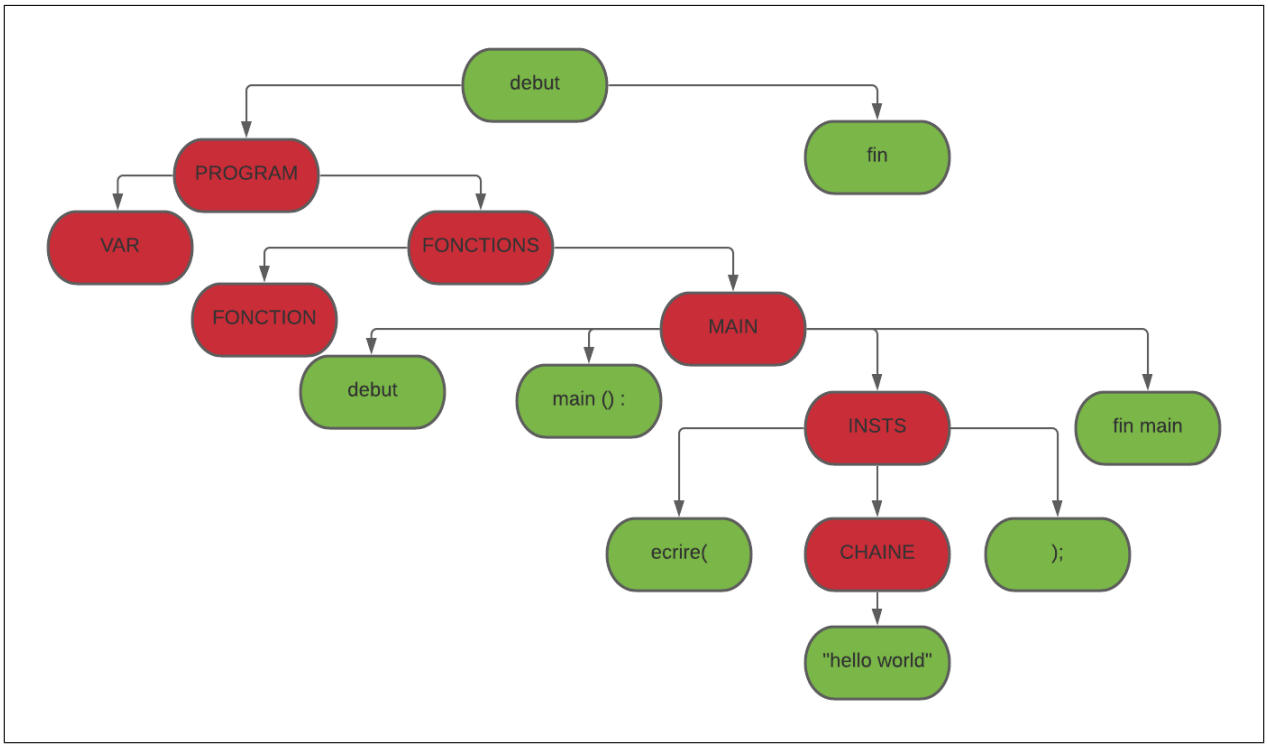

FIGURE 3.1 – Arbre syntaxique

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 est construit en utilisant la dérivation suivante :

DEBUT -> debut VAR FONCTIONS MAIN fin

VAR ->epsilon

FONCTIONS-> epsillon

MAIN->debut main() : INSTS fin

INSTS-> INST

INST -> ECRIRE

ECRIRE->ecrire(CHAINES)

CHAINES->CHAINE

CHAINE ->"hello world"

3.2 les fonctions de l'analyseur syntaxique

Notre programme parcours l'ensemble des nœuds de l'arbre syntaxique, de façon récursive. Une fonction générale analyse_syntaxique() est utilisée pour tester la syntaxe par tester chaque bloc du programme .

Chaque bloc associe a une fonction qui vérifie la syntaxe du bloc : si il y a une erreur il s'arrête l'analyse et il va spécifier le type et la ligne de l'erreur, sinon il va passer au bloc suivant.

Ces fonctions sont construits a partir des règles de production :

NT	<–	Régle de Production	Code C
DEBUT	<–	debut VAR FONCTIONS MAIN fin	PROGRAM()
VAR	<–	TYPE ID S' | epsilon	VARS();
TYPE	<–	entier|caract|chaine|bit|reel	switch(token) :
FONCTIONS	<–	FONCTION FONCTIONS |epsilon	FONCTIONS();
FONCTION	<–	debut TYPE ID (ARG) :INSTS fin ID	fonction();
MAIN	<–	debut main() :INSTS fin main	fonction(MAIN);
INSTS	<–	INST INSTS	INSTS();
INST	<–	AFFEC | SI | TANTQUE | ECRIRE | LIRE | epsilon	INST();
AFFEC	<–	ID <- EXPR	AFFEC();
SI	<–	si COND alors :INSTS CHOIX fin si	si();
CHOIX	<–	AUTRE CHOIX |sinon : INSTS | epsilon	CHOIX();
TANTQUE	<–	tantque COND faire : INSTS	tantque();
ECRIRE	<–	ecrire(CHAINES)	ECRIRE();
EXPRS	<–	EXPR EXPRS	EXPR();
LIRE	<–	lire(IDS)	LIRE();
COND	<–	EXPR RELOP EXPR	COND();
TERM	<–	FACT MULOP FACT	TERM();
FACT	<–	ID | NUM | (EXPR)	FACT5);

```
//fonction principale de analyse syntaxique
int analyse_syntaxique() {…
/* analyse de la liste des tokens */
void PROGRAM() {…
/* passage d'un symbole à l'autre */
void Sym_Suiv() {…
/* arreter le programme*/
void Erreur(TOKENS COD_ERR, int word) {…
/* test de symobole passer ne paramètre avec les elts de la liste chainée*/
void Test_Symbole(TOKENS tok) {…
/* verification des VAR */
void VARS() {…
/* analyse du corps du programme */
void FONCTIONS() {…
/* verification des instruction */
void INSTS() {…
/* verification des AFF,OPTION ,SI ,POUR ,TANTQUE.....*/
void INST() {…
/* verification des AFFEC (affectation) */
void AFFEC() {…
/* verification des expresions + ou - */
void EXPR() {…
/* verification des termes */
void TERM() {…
/* verification des facteurs */
void FACT() {…
/* verification fonction */
void fonction() {…
/* verification POUR */
void pour() {…
/* verification OPTION */
void option() {…
/* verification de SI */
void si() {…
void CHOIX() {…
/* verification de la condition apres SI */
void COND() {…
/* verification de ce qui est après le  TANTQUE*/
void tantque() {…
/* verification de ce qui est après l'ecriture et lecture */
void ecrire() {…
void lire() {…
```

Figure 3.2 – La liste des fonctions syntaxiques

3.3 Résultats de l'analyseur syntaxique

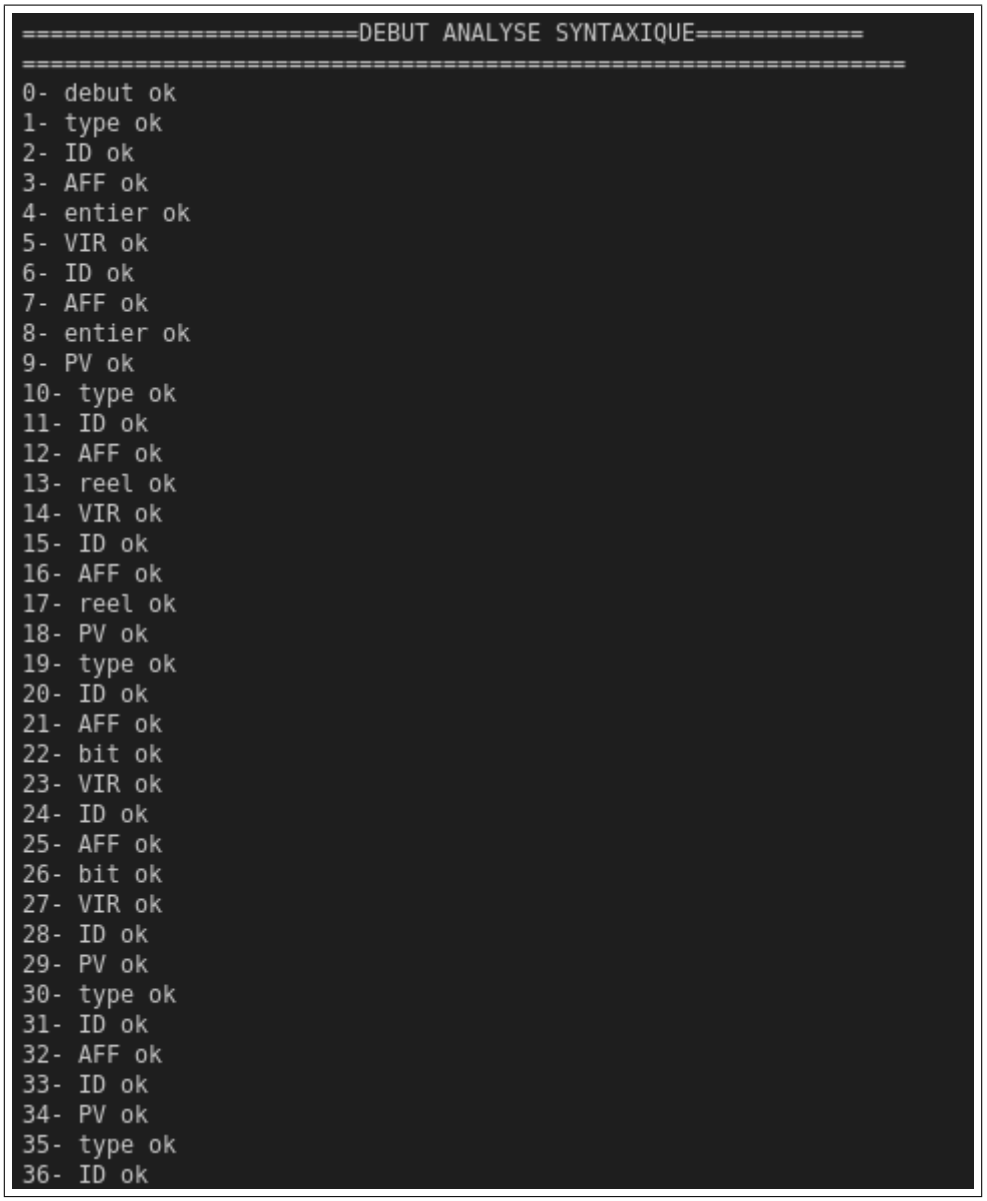

FIGURE 3.3 – Résultats de l'analyseur syntaxique

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.