RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
École nationale supérieure d'informatique (ESI ex. INI)

Rapport du Projet du module Compilation

2ème Année Cycle Supérieur (2CS)
2024-2025

Option : Systèmes Informatiques et Logiciels (SIL)

Réalisation d'un mini-compilateur

Réalisé par

- **MAMOUNI Mahdi**
- **AITMEKIDECHE Mohammed Amine**
- **DABOUZ Mohamed Amine**
- **BOUCENNA Oussama**

Groupe:SIL1

Année : 2024/2025

Tables des matières

I. Introduction

Notre projet, intitulé **Madiba**, vise à développer un mini-compilateur intégrant l'analyse lexicale et syntaxico-sémantique. Ce langage, conçu pour être simple et efficace, prend en charge divers types de données, structures de contrôle, opérateurs, et fonctionnalités d'entrée/sortie. *Madiba* combine clarté et modernité, offrant une base pédagogique solide pour explorer les concepts fondamentaux de la compilation.

II. Notre langage

1. Structure générale du programme:

La structure d'un programme **Madiba** est la suivante :

```
-- Inclusion des bibliothèques permettant d'utiliser des fonctions et modules préexistants.
import "Nom_de_bibliothèque";

--Le programme commence par le mot clé "program" suivi par le nom du programme .
program Nom_du_programme

--Cette partie est consacrée pour la déclaration des constantes. elle commence avec le mot clé CONST et les déclarations se font à l'intérieur de deux accolades {}.
const
{
Type Nom_constante = valeur; --Une constante est déclaré avec le mot clé "cst".
}

--Cette partie est consacrée pour la déclaration des Variables. elle commence avec le mot clé VAR et les déclarations se font à l'intérieur de deux accolades {}.
var
{
Type Nom_variable = valeur;
}

--Cette partie est consacrée pour la déclaration Types. elle commence avec le mot clé TYPES et les déclarations se font à l'intérieur de deux accolades {}.
types
{

}

--Cette partie est consacrée pour l'écriture des différentes fonctions du programme. elle commence avec le mot clé FUNCTIONS et les déclarations se font à l'intérieur de deux
```

accolades {}.

```
functions
{
int function nom_fonction (int:mehdi){
--Corps de la fonction
};
void function nom_fonction (){
--Corps de la fonction
};
}

--Le contenu du programme principal s'écrit à l'intérieur de cette section qui commence par le mot clé MAIN
MAIN
{
--Code
{
```

2. Commentaires

Dans le langage **Madiba**, les commentaires jouent un rôle essentiel en améliorant la lisibilité et la compréhension du code. Deux styles sont pris en charge :

- Les **commentaires sur une seule ligne** sont initiés par **--** et permettent d'ajouter des remarques rapides.
  Exemple :
  ```
  -- Ceci est un commentaire sur une ligne
  ```
- Les **commentaires multilignes** sont encadrés par ** et permettent d'insérer des explications détaillées sur plusieurs lignes.
  Exemple :
  ```
  **
  Ceci est un commentaire
  sur plusieurs lignes
  **
  ```

3. Déclarations

3.1 Déclaration des variables Simples

Entier (int) :Utilisé pour représenter des nombres entiers.

```
int Nom_var; -- Variable entière [-2³², 2³²]
```

Réel (float) :Utilisé pour les nombres à virgule flottante.
float Nom_var; -- Variable réelle [3.4 * 10^{-38}, 3.4 * 10^{38}

Caractère (char) : Représente un caractère unique (encodé en ASCII).
char Nom_var; -- Variable caractère [0, 255]

Chaîne de caractères (string) :Représente une séquence de caractères.
string Nom_var; -- Variable chaîne de caractères

Booléen (bool) :Représente une valeur logique (vrai ou faux).
bool Nom_var; -- Variable booléenne {true, false}

3.2. Déclaration des types avancées

3.2.1. Structures (Objects)

Déclaration : la déclaration des objets se fait dans la section TYPES

```
Object nom_de_Objet {
   type nom_de_l'attribut1= valeur;
   type nom_de_l'attribut2;
   type nom_de_l'attribut3;
}
```

Utilisation :

```
nom_de_Objet.nom_de_l'attribut1 = "Mehdi";
nom_de_Objet.nom_de_l'attribut2 =  12 ;
nom_de_Objet.nom_de_l'attribut3 = "Homme";
```

3.2.2. Tableaux

```
type nom_de_tableau [taille_de_tableau]; --int array[ 10 ];
```

- La taille du tableau doit être un entier positif.

Accès aux élements de tableaux

```
valeur = nom_de_tableau[indice];
```

- L'indice doit être compris entre 0 et la taille du tableaux -1 .

3.3. Appellation

● Le nom des variables dans **Madiba** doit contenir des caractères alphanumériques.
ou le caractère “_”,il doit commencer avec une lettre de l’alphabet.
● Les chaînes de caractères et les caractères doivent être écrits entre des guillemets “ ... ” .

4. Instructions de base

4.1. Affectation

L’affectation se fait à l’aide de l'opérateur.
Nom Variable = Expression;

Une *Expression* peut être :

- Une valeur
- Une variable
- Une opération arithmétique ou logique

4.2. Condition

La structure conditionnelle utilise **if** et, facultativement, **else**.

```
if (Condition) {
  -- Corps
} else {
  -- Corps
}
```

- Le **else** est facultatif
- La *Condition* doit être logique, c'est-à-dire une expression ou une valeur booléenne.
- Les blocs **if** peuvent être imbriqués pour une logique plus complexe.

4.3. Les boucles

4.3.1. Boucle Pour

La boucle **for** permet d’itérer entre une valeur initiale et une valeur finale, avec un pas.

```
for (Nom_variable = Valeur_initial, valeur_finale, Pas){
-- Corps
}
```

- Les valeurs initiale et finale, ainsi que le pas, doivent être de type entier.
- Le pas peut être positif (ordre croissant) ou négatif (ordre décroissant).

4.3.2. Boucle Tant Que

La boucle **while** continue tant que la condition reste vraie.

```
while (Condition){
-- Corps
}
```

- La condition doit être logique (Expression ou valeur booléen).
- Toutes les boucles peuvent être imbriquées.

5. Entrée / Sortie

5.1. Lecture de l'utilisateur

La commande **scan** permet de lire une valeur entrée par l'utilisateur et de l'assigner à une variable.

```
scan(Nom_variable);
```

5.2. Affichage

La commande **print** affiche la valeur d'une expression ou d'une variable dans la console.

```
print(Expression);
```

6. Opérateurs

6.1. Logique

```
Expression1 AND Expression2 ; -- le et logique
Expression1 OR Expression2 ; -- le ou logique
NOT Expression ; -- la négation logique
```

6.2. Arithmétique

```
Expression1 + Expression2; -- Addition
Expression1 - Expression2; -- Soustraction
Expression1 * Expression2; -- Multiplication
Expression1 / Expression2; -- Division (réelle ou entière)
```

Expression1 **%** Expression2; -- Le modulo

6.3. Comparaison

== : Egalité
< : inférieure
> : Supérieure
>= : Supérieure ou égale
<= : inférieure ou égale
!= : différent
● On peut utiliser les parenthèses

6.4. les différentes priorités et règles d'associativité

1. Parenthèses().
2. NOT.
3. AND.
4. OR.
5. multiplication , division réel , division entière, modulo (selon l'ordre).
6. addition , soustraction (selon l'ordre).
7. Les comparaisons.

III. Analyse lexicale:

1. Introduction

Le rôle de l'analyse lexicale est de transformer le code source (un texte brut) en une séquence de tokens (unités lexicales significatives) qui seront ensuite utilisés par l'analyseur syntaxique.

Dans notre langage, nous avons défini un ensemble de mots-clés spécifiques qui doivent être reconnus par le programme. L'ordre de déclaration de ces mots est important, car il permet de structurer correctement le code selon la grammaire du langage. Ces mots-clés sont traduits en tokens lors de l'analyse lexicale. Ces tokens servent ensuite d'entrée pour l'analyse syntaxique, où ils permettent de vérifier si la structure du programme respecte les règles de notre langage. Par exemple :

```
"import"            {  return IMPORT; }
"program"           { return PROGRAM; }
"const"             { return CONST; }
"var"               { return VAR; }
"types"             { return TYPES; }
"functions"         { return FUNCTIONS; }
"main"              { return MAIN; }
"int"               { return INT; }
"float"             { return FLOAT; }
"char"              { return CHAR; }
"bool"              { return BOOL; }
"string"            { return STRING; }
"cst"               { return CST; }
"Object"            { return OBJECT; }
"enum"              { return ENUM; }
"function"          { return FUNCTION; }
"for"               { return FOR; }
"while"             { return WHILE; }
"if"                { return IF; }
"else"              { return ELSE; }
"scan"              { return SCAN; }
"print"             { return PRINT; }
"return"            { return RETURN; }
"void"         {return VOID;}
```

Dans l'analyse lexicale, chaque fois qu'un token est reconnu, il peut être associé à une valeur supplémentaire pour fournir des informations utiles à l'analyse syntaxique. Pour ce faire, nous utilisons la variable globale yylval, qui permet de transmettre ces valeurs entre l'analyseur lexical et syntaxique.

```
[0-9]+                { yylval.intval = atoi(yytext); return INTEGER_LITERAL; }
[0-9]+\.[0-9]+        { yylval.floatval = atof(yytext); return FLOAT_LITERAL; }
\"([^\"]|\\.)\"         { yylval.charval = yytext[1]; return CHAR_LITERAL; }
\"([^\"]|\\.)*\"      { yylval.strval = strdup(yytext); return STRING_LITERAL; }
```

il est possible d'ignorer certains mots ou séquences qui ne sont pas nécessaires pour l'analyse syntaxique, comme les commentaires dans le code.

```
"--".*                { /* igonre les commentaires */ }
\*\*([^*]|\*+[^*/])*\*\*  { /* igonre les commentaires */ }
```

Il existe des règles qui sont utilisées pour mettre à jour le numéro de ligne et de colonne afin de localiser précisément les erreurs dans le code source.

```
[\n] {
    lineNumber++;
    columnNumber = 1;
}
[ \t] {
    columnNumber += yyleng;
}
```

La règle ci-dessus permet de détecter les mots qui n'appartiennent pas au langage (erreur lexicale).

```
. {
    printf(" Erreur lexicale concernant l'entite  %s ligne %d colonne %d \n",yytext , lineNumber , columnNumber);
}
```

2. Table de symbole

2.1. Définition

Une table de symboles est une structure centralisée qui stocke des informations sur les identificateurs d'un programme, telles que le type, l'emplacement, la portée et la visibilité. Elle optimise l'accès aux données et accélère la compilation. Dans la phase d'analyse sémantique, la table des symboles est essentielle pour vérifier la validité des types, la cohérence des déclarations et la gestion de la portée des variables et fonctions.

2.2 structure

```c
#ifndef SYMBOL_TABLE_H
#define SYMBOL_TABLE_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Définitions des classes
#define T_UNKNOWN 0          // Inconnu
#define C_CONST 1        // Constante
#define C_VAR 2          // Variable
#define C_TYPE 3         // Type utilisateur (enum, object, ARRAY.)
#define C_LOCAL 4        // Variable locale
#define C_ARGUMENT 5     // Argument d'une fonction
#define C_CHAMP 6     // Argument d'une fonction
#define C_ARRAY 7       // Tableau
int insertSymbol(const char *name, int classe, int type, int complement);
// Définitions des types
#define T_INT 1          // Type entier
#define T_FLOAT 2        // Type flottant
#define T_BOOL 3         // Type booléen
#define T_CHAR 4         // Type caractère
#define T_STRING 5       // Type chaîne de caractères
#define T_OBJECT 7       // Objet (structure utilisateur)
#define T_ENUM 8         // Enumération
#define T_FUNCTION 9     // Fonction

// Taille maximale de la table
#define MAX_DICO 1000
```

```c
// Description d'un identifiant
typedef struct {
    char *identif;   // Nom de l'identifiant
    int classe;      // Classe ( locale, argument)
    int type;        // Type (int, float, bool ...
    // char *valeur;    // Valeur associée
    int complement;  // Complément (taille pour les tableaux, nb de paramètres pour les fonctions, etc.)
} desc_identif;

// Structure de la table des symboles
typedef struct {
    desc_identif tab[MAX_DICO]; // Tableau des identifiants
    int base;
    int sommet;                 // Indice du sommet de la pile
} table_symbol;

// Déclaration de la table des symboles globale
extern table_symbol symbolTable;

// Initialisation de la table des symboles
void initTable();

// Insertion d'un identifiant dans la table
int insertSymbol(const char *name, int classe, int type, int complement);

// Recherche d'un identifiant dans la table
desc_identif* lookupSymbol(const char *name);

// Affichage de la table des symboles
void displayTable();

#endif // SYMBOL_TABLE_H
```

2.3 Méthodes

1. **insertSymbol** : est utilisée pour ajouter un nouvel élément (ou un nouveau lexème) à une table des symboles

```c
// Initialisation de la table des symboles
void initTable() {
    symbolTable.base = 0;
    symbolTable.sommet = 0;
}

// Insertion d'un identifiant dans la table
int insertSymbol(const char *name, int classe, int type, int complement) {
    if (symbolTable.sommet >= MAX_DICO) {
        printf("Erreur : Table des symboles pleine !\n");
        return -1;
    }

    symbolTable.tab[symbolTable.sommet].identif = strdup(name); // Copie du nom
    symbolTable.tab[symbolTable.sommet].classe = classe;
    symbolTable.tab[symbolTable.sommet].type = type;
    symbolTable.tab[symbolTable.sommet].complement = complement;

    symbolTable.sommet++;

    return 0;
}
```

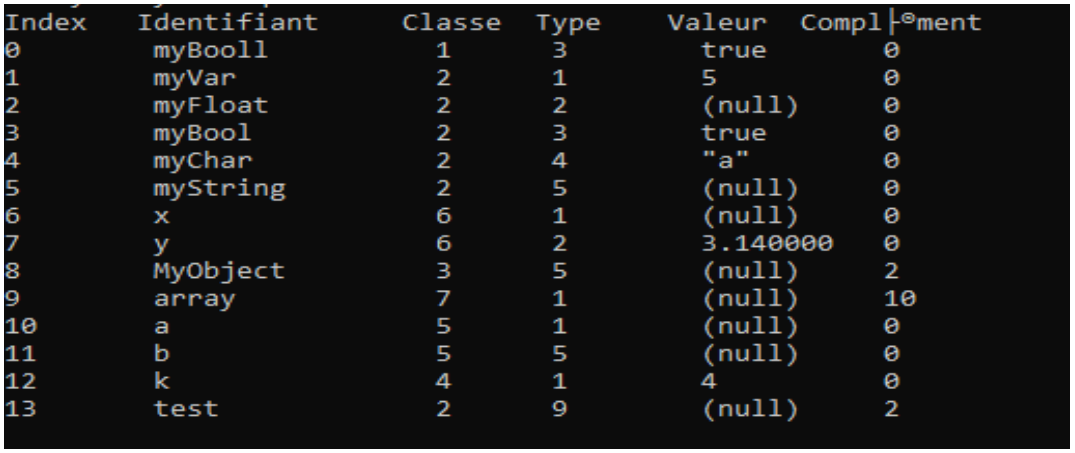
2. **displayTable** :Affiche le contenu actuel de la table des symboles de manière lisible pour les développeurs.

```c
// Affichage de la table des symboles
void displayTable() {
    printf("Index\tIdentifiant\tClasse\tType\tComplément\n");
    for (int i = 0; i < symbolTable.sommet; i++) {
        desc_identif *entry = &symbolTable.tab[i];
        printf("%-8d %-16s %-6d %-8d %-5d\n", i, symbolTable.tab[i].identif, symbolTable.tab[i].classe, symbolTable.tab[i].type,
               symbolTable.tab[i].complement);
    }
}
```

3. **lookupSymbol** : Recherche un identifiant dans la table.

```c
// Recherche d'un identifiant dans la table
desc_identif* lookupSymbol(const char *name) {
    for (int i = symbolTable.sommet - 1; i >= symbolTable.base; i--) {
        if (strcmp(symbolTable.tab[i].identif, name) == 0) {
            return &symbolTable.tab[i]; // Retourne un pointeur vers l'objet trouvé
        }
    }
    return NULL; // Retourne NULL si l'identifiant n'existe pas
}
```

exemple :



Index	Identifiant	Classe	Type	Valeur	Compl├®ment
0	myBooll	1	3	true	0
1	myVar	2	1	5	0
2	myFloat	2	2	(null)	0
3	myBool	2	3	true	0
4	myChar	2	4	"a"	0
5	myString	2	5	(null)	0
6	x	6	1	(null)	0
7	y	6	2	3.140000	0
8	MyObject	3	5	(null)	2
9	array	7	1	(null)	10
10	a	5	1	(null)	0
11	b	5	5	(null)	0
12	k	4	1	4	0
13	test	2	9	(null)	2

IV. Analyse Syntaxique:

1. Introduction

Dans la continuité de notre projet de compilation, nous allons désormais nous concentrer sur l'analyse syntaxique de notre langage **Madiba**, une étape qui fait suite à l'analyse lexicale précédemment effectuée. L'analyse syntaxique joue un rôle clé dans le traitement automatique des langues, car elle permet de vérifier si les mots d'une phrase sont agencés correctement selon les règles de la grammaire et d'en déduire leur signification.

2. Partie 1: implémentation en utilisant le C

Dans cette section, nous avons opté pour le traitement spécifique des instructions de déclaration en utilisant l'analyse LL(1). Cette approche descendante permet de construire une table d'analyse syntaxique de manière systématique, afin de vérifier la conformité des instructions de déclaration avec la grammaire du langage.

Pour ce faire, nous avons utilisé l'outil Flex pour identifier les entités lexicales présentes dans le programme. Ces entités sont ensuite transformées en tokens, afin de faciliter leur manipulation dans le programme implémenté en C.

1 – Implementation :

on a commencé par faire nos déclarations, Définir la structure et les fonctions dont on aura besoin:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <stdbool.h>

#define MAX 100

//Declaration des fonctions
void calculer_debut(char *, char);
void calculer_suivant(char *, char);
void affich_debut();
void affich_suivant();
int  containsss(char *arr, char ch);
int  creer_table_analyse();
void affich_table_analyse(int valide);
void analyseur_ll1(char *);
void verif_intersection_debuts();
void verif_factorisation();
void verif_recursivite_gauche();
void chack_Ll1_condition();

```

```c
int top = 0;
int nb_ter,   //nombre des terminaux
    nb_noter, //nombre des non-terminaux
    chaine,   //la chaine
    nb_regle, //nombre de regles de production
    count;    //compteur

char DEBUT[MAX][MAX],     //tableau pour mettre l'ensemble des debuts
     SUIVANT[MAX][MAX];   //tableau pour mettre l'ensemble des suivants

char TER[MAX],             //tableau pour mettre l'ensemble des terminaux
     NOTER[MAX],           //tableau pour mettre l'ensemble des non terminaux
     GRAMMAIRE[MAX][MAX],  //la grammaire
     PILE[MAX];            //notre pile

int table_analyse_ll1[MAX][MAX];  //la table d'analyse

```

- En suite on a mis en place une méthode qui détecte si la grammaire récursive gauche ou pas:

```c
// Fonction pour vérifier la récursion à gauche dans la grammaire
void verifier_recursion_gauche() {
    printf("\nVérification de la récursion à gauche...\n");
```

- Après, on a réalisé une méthode qui vérifie si la grammaire est factorisée ou pas:

```
// Fonction pour vérifier la factorisation dans la grammaire
void verifier_factorisation() {
```

Après, une méthode qui vérifie l'intersection des débuts:

```
// Fonction pour vérifier l'intersection des ensembles DEBUT des productions
void verif_intersection_debuts() {
```

- Les fonctions de création et d'affichage des tables des débuts et suivants:

```
// Fonction pour calculer l'ensemble DEBUT
void calculer_debut(char *production, char non_terminal) {
```

```
// Fonction d'affichage des DEBUT
void affich_debut() {
```

```
// Fonction pour calculer l'ensemble SUIVANT
void calculer_suivant(char *production, char non_terminal) {
```

```
// Fonction d'affichage des SUIVANT
void affich_suivant() {
```

- La méthode principale qui permet de générer la table d'analyse:

```
int creer_table_analyse() {
```

```
void affich_table_analyse(int success) {
    if(!success) {
        printf("\nLa grammaire n'est pas LL(1)\n");
        return;
    }

    printf("\n*************** TABLE D'ANALYSE LL(1) ***************\n\n");
    printf("    ");
    for(int i = 0; i < nb_ter; i++) {
        printf("%c  ", TER[i]);
    }
    printf("\n");

    for(int i = 0; i < nb_noter; i++) {
```

2 – Commande pour l'exécution :

>flex des.l
>gcc lex.yy.c -o m
> ./m

3 – Résultat du test :

- La grammaire en entrée :

```
---------------- LA GRAMMAIRE ----------------------
Z->S
S->tD
D->dA
A->=V
V->n;
```

- Vérification de récursivité gauche et factorisation :

```
Verification de la recursion a gauche...
>> Aucune recursion a gauche detectee.

Verification de la factorisation...
>> La grammaire est factorisee.
```

- Calcule des ensembles des débuts et suivants :

```
*************** ENSEMBLE DES DEBUTS ***************

DEBUT( Z ): {  t }
DEBUT( S ): {  t }
DEBUT( D ): {  d }
DEBUT( A ): {  = }
DEBUT( V ): {  n }

*************** ENSEMBLE DES SUIVANTS ***************

SUIVANT( Z ): {  # }
SUIVANT( S ): {  # }
SUIVANT( D ): {  # }
SUIVANT( A ): {  # }
SUIVANT( V ): {  # }
```

- Vérification des intersections des débuts :

```
Verification de l'intersection des debuts...

Non-terminal : Z
>> Aucun conflit trouve pour Z.

Non-terminal : S
>> Aucun conflit trouve pour S.

Non-terminal : D
>> Aucun conflit trouve pour D.

Non-terminal : A
>> Aucun conflit trouve pour A.

Non-terminal : V
>> Aucun conflit trouve pour V.
```

- Table LL1

```
**************** TABLE D'ANALYSE LL(1) ****************

     t   d   =   n   ;
 Z   0   -   -   -   -
 S   1   -   -   -   -
 D   -   2   -   -   -
 A   -   -   3   -   -
 V   -   -   -   4   -
```

```
Ligne originale : TOKEN_INT IDENTIFIER ASSIGN VAL SEMICOLON
Cha├«ne simplifi├⌐e : td=n;

S├⌐quence d'analyse et actions

PILE                    RESTE DE LA CHAINE                    ACTION
-------------------------------------------------------------------------------
#Z                      td=n;#                D├⌐calage: Z->S
#S                      td=n;#                D├⌐calage: S->tD
#Dt                     td=n;#                R├⌐duit: t
#D                      d=n;#                 D├⌐calage: D->dA
#Ad                     d=n;#                 R├⌐duit: d
#A                      =n;#                  D├⌐calage: A->=V
#V=                     =n;#                  R├⌐duit: =
#V                      n;#                   D├⌐calage: V->n;
#;n                     n;#                   R├⌐duit: n
#;                      ;#                    R├⌐duit: ;
#               #

Analyse termin├⌐e avec succ├¿s
```

3. Partie 2: Analyse ascendante avec BISON

Dans un second temps, nous allons étendre l'analyseur syntaxique pour couvrir l'ensemble de la grammaire de notre langage en utilisant l'outil **BISON**. BISON est un générateur

d'analyseur syntaxique descendant qui opère sur des grammaires **LALR** (Look-Ahead LR), un sous-ensemble de la famille des grammaires LR.

Le fichier BISON est structuré comme suit :

1 – Partie de définition :

Cette partie comporte :

– l'inclusion de la bibliothèque "symbol_table.h"

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "symbol_table.h"
```

– Déclaration des tokens , ces derniers constituant les unités lexicales utilisées pour générer notre langage. Ces tokens sont définis dans l'analyseur lexical et servent à identifier les différents éléments syntaxiques du langage.

```
%token IMPORT PROGRAM CONST VAR TYPES FUNCTIONS MAIN INT FLOAT CHAR BOOL STRING CST OBJECT ENUM FUNCTION FOR WHILE IF ELSE SCAN PRINT VOID TRUE
%token INTEGER_LITERAL FLOAT_LITERAL CHAR_LITERAL STRING_LITERAL IDENTIFIER ARRAY OBJECT_FIELD DOT
%token AND OR NOT
%token EQUAL NOT_EQUAL LESS_THAN GREATER_THAN LESS_EQUAL GREATER_EQUAL
%token PLUS MINUS MULTIPLY DIVIDE MODULO
%token COMMA OPEN_PAREN CLOSE_PAREN OPEN_BRACKET CLOSE_BRACKET OPEN_BRACE CLOSE_BRACE SEMICOLON QUOTE ASSIGN TWO_POINT RETURN
```

– Cette configuration permet de résoudre les ambiguïtés lors de l'analyse syntaxique, en indiquant comment Bison doit traiter les opérateurs dans des expressions complexes.

```
%left AND OR
%left EQUAL NOT_EQUAL LESS_THAN GREATER_THAN LESS_EQUAL GREATER_EQUAL
%left PLUS MINUS
%left MULTIPLY DIVIDE MODULO
%right NOT
```

Cette partie définit les types de données associés aux tokens et non-terminaux.

```
%union {
    int intval;         // Pour INTEGER_LITERAL
    float floatval;     // Pour FLOAT_LITERAL
    char charval;       // Pour CHAR_LITERAL
    char* strval;       // Pour STRING_LITERAL et IDENTIFIE
    int type;           // Pour les types (INT, FLOAT, etc.

    struct expressionTYPE {
    int type ;
    char val[256];
    } expressionTYPE;
}
```

2 – Partie des régle de production ：

Dans cette section, nous avons défini les règles de production qui permettent de **décrire la structure syntaxique du langage MADIBA**. Ces règles servent à analyser un programme écrit dans ce langage et à vérifier qu'il respecte sa grammaire. L'exemple suivant illustre la règle principale qui englobe l'ensemble du programme.

```
program:
    import_section PROGRAM IDENTIFIER const_section var_section types_section functions_section main_section
    {
        printf("Analyse syntaxique correcte.\n");
        displayTable(); // Affiche la table des symboles après l'analyse
        YYACCEPT;
    }
    ;
```

4 – Gestion des erreurs

Les erreurs syntaxiques sont prises en charge grâce à la fonction **yyerror**, qui permet de signaler les anomalies détectées tout en indiquant leur emplacement exact dans le code source. Cela facilite l'identification et la correction des erreurs par l'utilisateur.Voici un exemple d’erreur générée par la fonction yyerror.

```
C:\Users\ad\Desktop\FINI>t.exe <test.txt

Analyse syntaxique en cours...
Erreur syntaxique ligne 9, colonne 2 : syntax error
```

V. Analyse sémantique

Pour l'analyse syntaxique, on utilise une forme de quadruplet pour représenter les instructions du programme.

1.Structure de quadruplet:

```
//Structure du quadruplet
typedef struct quadruplet quadruplet;
struct quadruplet
{
    char oprt[30];
    char op1[30];
    char op2[30];
    char result[30];
};

quadruplet tab_quad[1000];

pile  stack;

int qc = 0 ; // qc indice quadriple
int ti = 0 ; //var temp pour garder
```

Cette partie définit des structures et des variables utilisées pour la génération de code intermédiaire dans un compilateur :

1. **Structure `quadruplet`** : Elle représente une quadruplet, un format standard pour stocker des instructions intermédiaires. Chaque quadruplet contient :
    - `oprt` : L'opérateur (ex. : `+`, `-`, etc.).

    - `op1` : Le premier opérande.
    - `op2` : Le deuxième opérande.
    - `result` : Le résultat de l'opération.
2. **`tab_quad`** : Un tableau qui stocke jusqu'à 1000 quadruplets.
3. **`stack`** : Une pile utilisée pour la gestion des instructions (ex. : expressions imbriquées, retour à des points précis).
4. **`qc`** : Un compteur pour suivre l'indice actuel dans `tab_quad`.
5. **`ti`** : Un compteur utilisé pour générer des variables temporaires nécessaires lors de la création de quadruplets.

2. Pile

2.1 Structure

```c
#define MAX 128

typedef struct pile pile;
struct pile{
    int sommet;
    int table[MAX];
};

void initPile(pile *P);


void empiler(pile *p, int x);

int depiler(pile *p);
int estVide(pile *p);
int estPleine(pile *p);
```

3. explication des routines

Pour les routines, nous allons expliquer le fonctionnement général de l'instruction `if-else` :

1. **Vérification de la condition** : Si la condition est de type booléen, l'adresse d'un saut potentiel est enregistrée dans une pile. Sinon, une erreur est levée.
2. **Bloc `if`** : Les instructions du bloc `if` sont exécutées si la condition est vraie.
3. **Saut conditionnel** : Si la condition est fausse, un saut est généré vers la fin du bloc `if` ou vers un éventuel bloc `else`.
4. **Mise à jour des adresses** : Les adresses des sauts (dans les quadruplets) sont ajustées dynamiquement à l'aide de la pile pour garantir un flux d'exécution correct.

Exemple des routines d'instruction if-else :

```
if_stmt:
    IF OPEN_PAREN expression CLOSE_PAREN {
        if ($3.type == T_BOOL) {
            empiler(&stack, qc-1);
        } else {
            yyerrorS("Condition must evaluate to a boolean value");
        }
    }
    OPEN_BRACE instruction_list CLOSE_BRACE else_part  {
        if (!estVide(&stack)) {
            char addrEndIf[10];
            int addrFalseJump = depiler(&stack);

            sprintf(addrEndIf, "%d", qc);
            strcpy(tab_quad[addrFalseJump].op1, addrEndIf);
        } else {
            printf("Erreur : La pile est vide.\n");
        }
    } ;
```

```
else_part:
| ELSE {
        strcpy(tab_quad[qc].oprt, "BR");
        strcpy(tab_quad[qc].op1, "");
        strcpy(tab_quad[qc].op2, "");
        strcpy(tab_quad[qc].result, "");

        if (!estVide(&stack)) {
            char addrEndIf[10];
            int addrFalseJump = depiler(&stack);
            empiler(&stack, qc);
            qc++;
            sprintf(addrEndIf, "%d", qc);
            strcpy(tab_quad[addrFalseJump].op1, addrEndIf);
        } else {
            printf("Erreur : La pile est vide.\n");
        }
    }
    OPEN_BRACE instruction_list CLOSE_BRACE
    ;
```

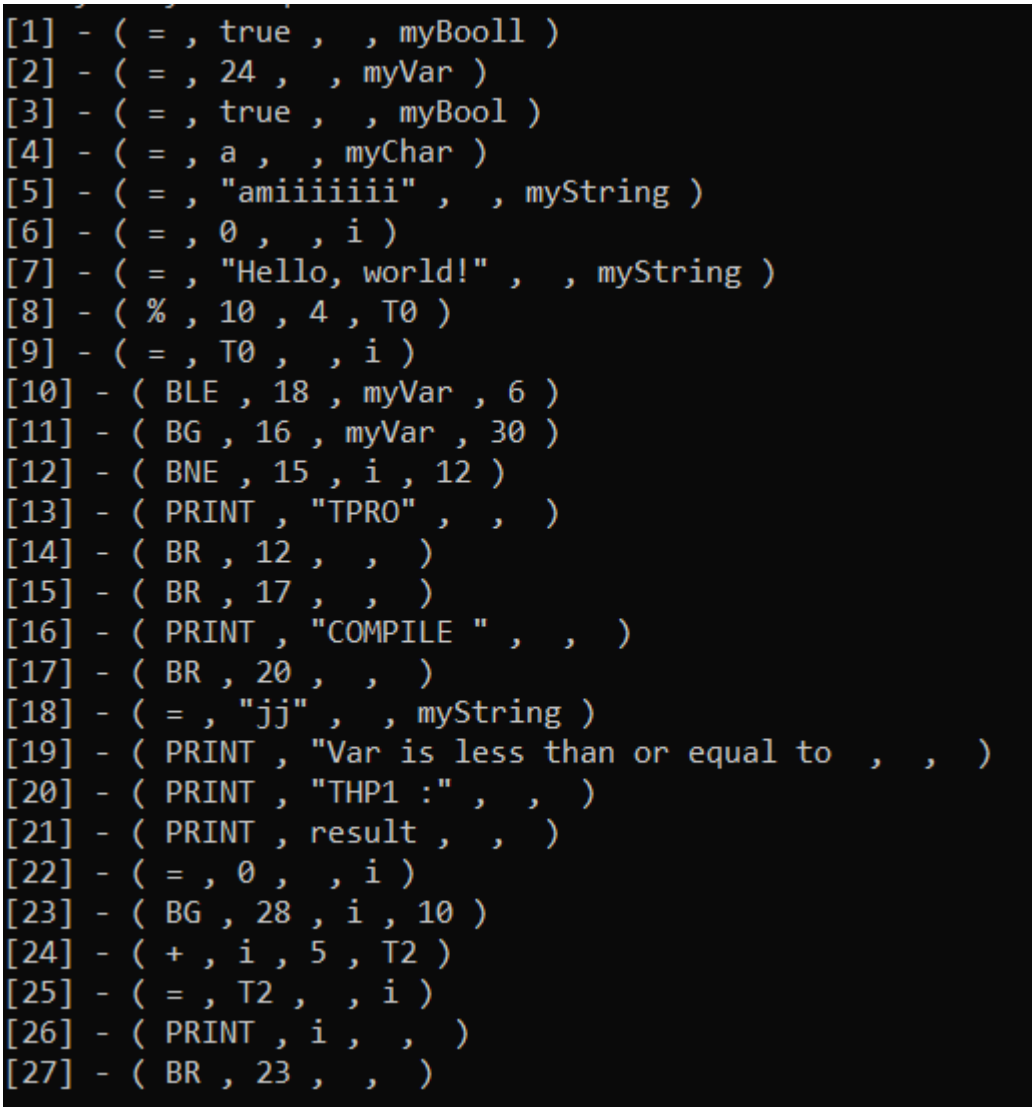
4.Test

1.Code du test

```
import "stdio.h";
program MyProgram
const {
    -- int MyConstant = 10;
        bool myBooll = true;
}
var {
    int myVar = 24;
    bool myBool = true;
    char myChar = "a";
    string myString = "amiiiiiii";
    int i = 0 ;
    string result ;
}
main {
    myString = "Hello, world!";
    i = 10 % 4;
    if (myVar > 6) {
        if ( myVar <= 30) {
           while (  i == 12) {
               print("TPRO");
           }
        }else {
           print("COMPILE ");
        }
    } else { myString = "jj";
        print("Var is less than or equal to 5"); }
    print("THP1 :", result);
        for (i = 0,10, 5) {print(i);}
}
```

2.Resultat du Test

```
Analyse syntaxique correcte.
Index   Identifiant     Classe  Type    Compl├®ment
0        myBooll          1       3       0
1        myVar            2       1       0
2        myBool           2       3       0
3        myChar           2       4       0
4        myString         2       5       0
5        i                2       1       0
6        result           2       5       0
Analyse syntaxique TERMINER...
```



VI conclusion

En conclusion, ce rapport a présenté de manière détaillée les différentes étapes de la construction d'un compilateur, englobant l'analyse lexicale, syntaxique et sémantique, jusqu'à la génération de code intermédiaire pour un langage que nous avons conçu. En exploitant des outils puissants tels que **Flex** pour identifier les lexèmes et gérer les erreurs lexicales, ainsi que **Bison** pour analyser la syntaxe, exécuter les routines sémantiques et produire un code intermédiaire structuré, nous avons pu valider la cohérence et la conformité des programmes à notre langage. Ce travail met en lumière l'importance de chaque étape dans le processus de compilation et leur rôle dans la construction d'un compilateur robuste et fonctionnel.