

Mini Projet Compilation

Réalisé par YUCEF Fatima Zohra



Présentation du mini projet

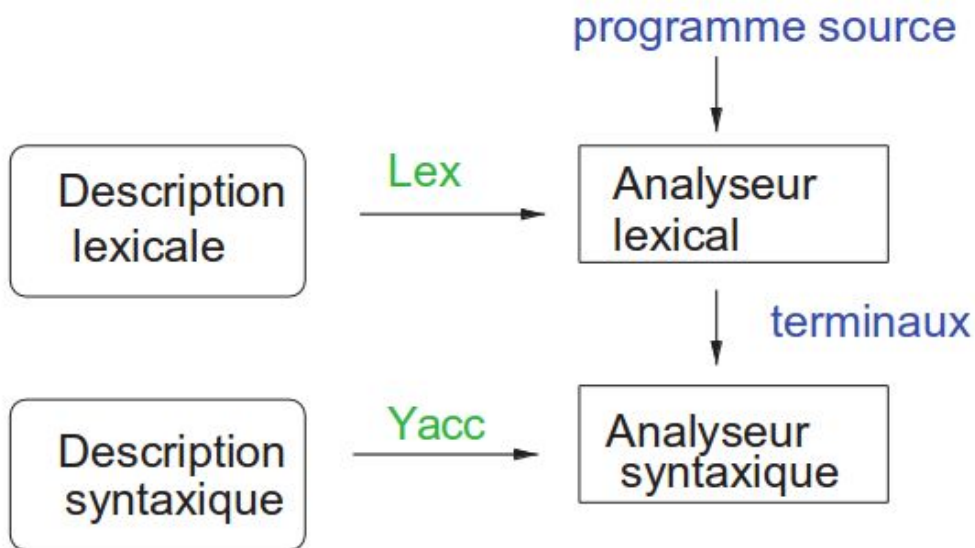
il est question d'implémenter un analyseur lexico-syntaxique en utilisant les outils Lex et Yacc, il s'agit donc de proposer un analyseur qui lit un programme source $L(G)$ et fait le traitement des phases lexicales et syntaxique de la compilation.

→ Réalisation

la phase lexicale se fait à l'aide de **lex** qui est un générateur d'analyseur lexical qui prend en entrée la définition des unités lexicales, Produit en sortie un automate fini déterministe minimal permettant de reconnaître les unités lexicales.

la phase syntaxique se fait à l'aide de **yacc** qui est un générateur d'analyseur syntaxique qui prend en entrée la définition d'un schéma de traduction (grammaire + actions sémantiques), Produit en sortie un analyseur syntaxique pour le schéma de traduction.

pour résumer :



❖ Application

1. en premier lieu j'ai créer un fichier.l que j'ai nommé miniproject.l dans lequel j'ai définit toute les unités lexicales du langage donné ($L(G)$)



```
%{
#include <stdio.h>
#include <stdlib.h>
#include "miniproject.tab.h"
}%

%%

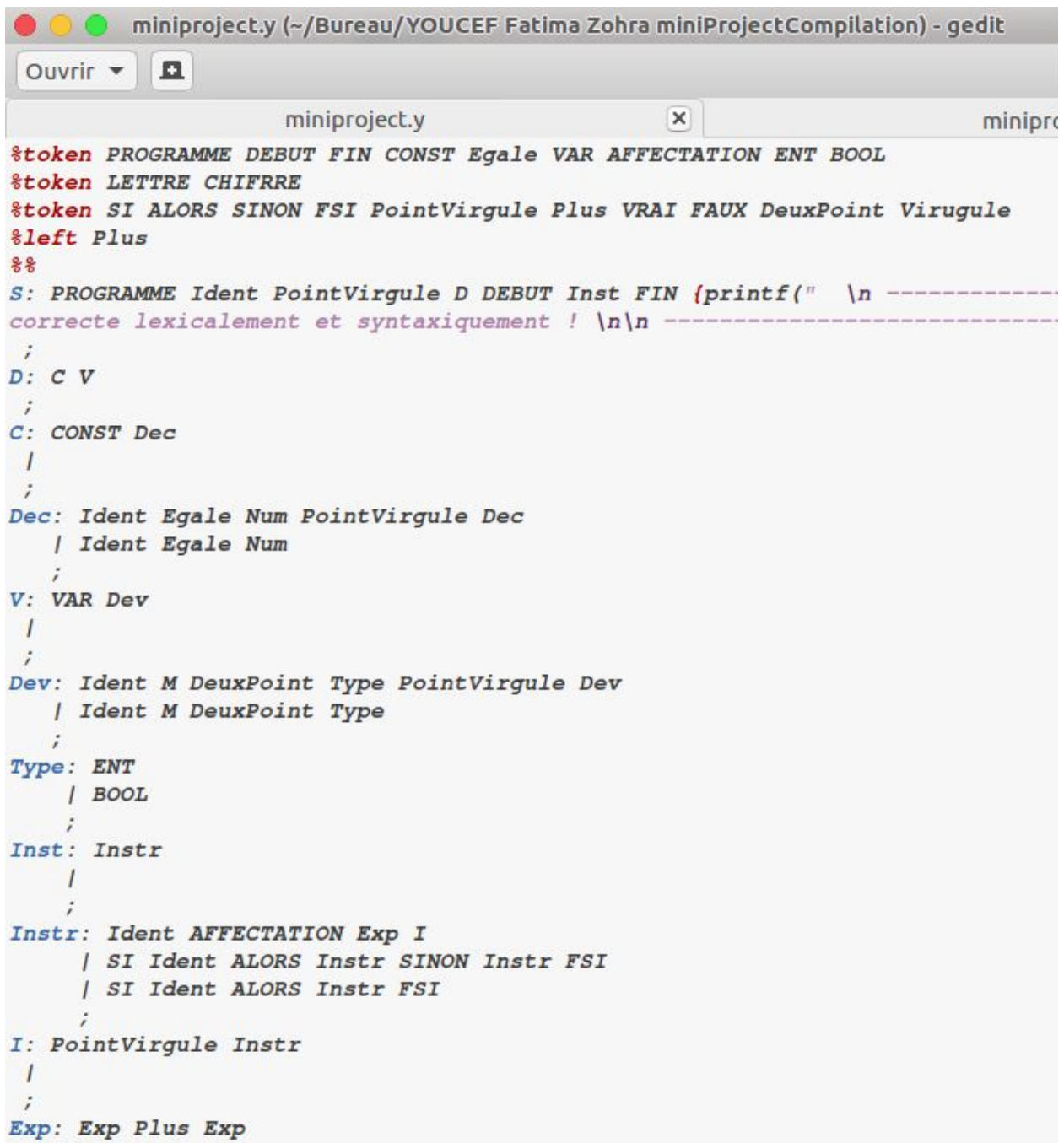
"programme" {return PROGRAMME;}
"debut" {return DEBUT;}
"fin" {return FIN;}
"const" {return CONST;}
"=" {return Egale;}
"var" {return VAR;}
":=" {return AFFECTATION;}
"si" {return SI;}
"alors" {return ALORS;}
"sinon" {return SINON;}
"fsi" {return FSI;}
";" {return PointVirgule;}
"+" {return Plus;}
"vrai" {return VRAI;}
"faux" {return FAUX;}
":" {return DeuxPoint;}
"," {return Virugule;}
[a-zA-Z] {return LETTRE;}
[0-9] {return CHIFFRE;}
"ent" {return ENT;}
"bool" {return BOOL;}
[ \t\n]+ ;

%%
```

2. en 2em lieux j'ai créer un fichier.y que j'ai nommé miniproject.y dans lequel j'ai définit l'ensemble des règles de la grammaire donné, ce fichier s'utilise conjointement à lex, qui lui fournit les unités lexicales (tokens), afin d'analyser leur organisation syntaxique pour cela on utilise la directive %token pour déclarer les terminaux pouvant être rencontrés.

Remarque:

sachant que que Yacc produit un code C, permettant de reconnaître un langage décrit par un ensemble de règles d'une grammaire pseudo-LALR, l'on a pas besoin d'enlever la récursivité gauche car cette dernière ne pose aucun problème lorsqu'il s'agit de LALR (selon les notions qu'on a vu en cour).



```
%token PROGRAMME DEBUT FIN CONST Egale VAR AFFECTATION ENT BOOL
%token LETTRE CHIFFRE
%token SI ALORS SINON FSI PointVirgule Plus VRAI FAUX DeuxPoint Virugule
%left Plus
%%
S: PROGRAMME Ident PointVirgule D DEBUT Instr FIN {printf("  \n -----
correcte lexicalement et syntaxiquement ! \n\n -----
;
D: C V
;
C: CONST Dec
|
;
Dec: Ident Egale Num PointVirgule Dec
| Ident Egale Num
;
V: VAR Dev
|
;
Dev: Ident M DeuxPoint Type PointVirgule Dev
| Ident M DeuxPoint Type
;
Type: ENT
| BOOL
;
Instr: Instr
|
;
Instr: Ident AFFECTATION Exp I
| SI Ident ALORS Instr SINON Instr FSI
| SI Ident ALORS Instr FSI
;
I: PointVirgule Instr
|
;
Exp: Exp Plus Exp
```

S est l'axiome, dans le cas ou tout ses règles seront respectées un message s'affiche indiquant que le programme est correcte lexicalement et syntaxiquement.

```
miniproject.y (~/Bureau/YOUCCEF Fatima Zohra miniProjectCompilation) - gedit
Ouvrir [icon]

miniproject.y [x] mini

/
;
Exp: Exp Plus Exp
    | Ident
    | Cste
;
Ident: Lettre SuitL
;
SuitL: Lettre SuitL
    | Chiffre SuitL
    |
;
Lettre: LETTRE
;

Chiffre: CHIFFRE
;
Cste: Chiffre SuitC
;
SuitC: SuitC Chiffre
    |
;
M: Virugule Ident M
    |
;
Num: Cste
    | VRAI
    | FAUX
;
%%
void yyerror(char *s) {
printf("%s\n", s);
}
void main(void) {
yyparse();
printf("fin d'evaluation");
}
```

la suite des règles et le programme principale qui effectue un appel de l'analyseur syntaxique par la méthode `yyparse()`

voici un petit programme a tester qui respect les règles du langage donné.



```
fatima@eva: ~/Bureau/YOUCF Fatima Zohra miniProjectCompilation
[fatima]—[~/Bureau/YOUCF Fatima Zohra miniProjectCompilation](master)
[$]→ bison -d miniproject.y
[fatima]—[~/Bureau/YOUCF Fatima Zohra miniProjectCompilation](master)
[$]→ flex miniproject.l
[fatima]—[~/Bureau/YOUCF Fatima Zohra miniProjectCompilation](master)
[$]→ gcc -o analyseur lex.yy.c miniproject.tab.c -ll -ly
miniproject.tab.c: In function 'yyparse':
miniproject.tab.c:1179:16: warning: implicit declaration of function 'yylex' [-Wimplicit-function-declaration]
    yychar = yylex ();
                   ^
[fatima]—[~/Bureau/YOUCF Fatima Zohra miniProjectCompilation](master)
[$]→ ./analyseur <ProgTest.txt

-----
Votre programme est correcte lexicalement et syntaxiquement !
-----

fin d'evaluation [fatima]—[~/Bureau/YOUCF Fatima Zohra miniProjectCompilation](master)
[$]→
```

ProgTest.txt

```
programme aTester;

const x=26;
      z=vrai

var x1:bool;
      b6:ent

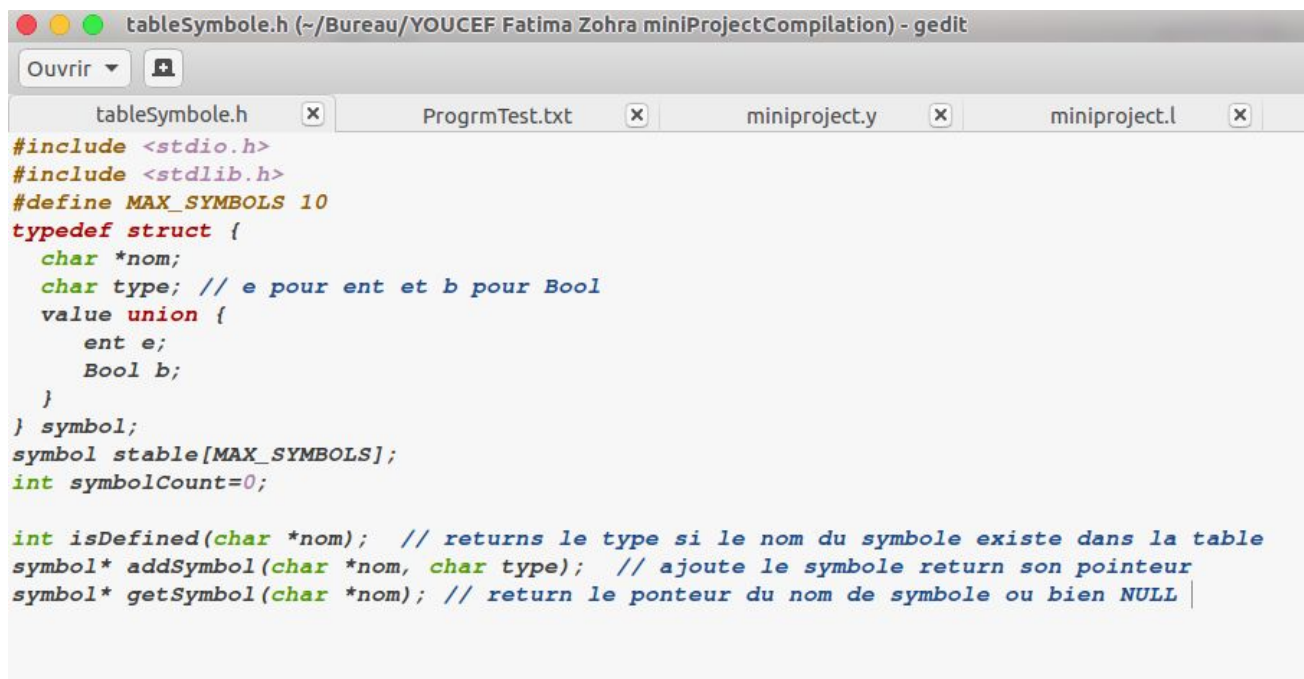
debut

    si b6 alors
        b6:= 5
    fsi

fin
```

table des symboles

j'ai essayer de l'implémenter en C (sous forme fichier.h) en utilisant une structure qui définit le nom et le type d'un symbole, le type est soit ent soit bool est soit ent soit Bool.



```
tableSymbole.h (~/Bureau/YOUCF Fatima Zohra miniProjectCompilation) - gedit
Ouvrir [icon]

tableSymbole.h x ProgTest.txt x miniproject.y x miniproject.l x
#include <stdio.h>
#include <stdlib.h>
#define MAX_SYMBOLS 10
typedef struct {
    char *nom;
    char type; // e pour ent et b pour Bool
    value union {
        ent e;
        Bool b;
    }
} symbol;
symbol stable[MAX_SYMBOLS];
int symbolCount=0;

int isDefined(char *nom); // returns le type si le nom du symbole existe dans la table
symbol* addSymbol(char *nom, char type); // ajoute le symbole return son pointeur
symbol* getSymbol(char *nom); // return le ponteur du nom de symbole ou bien NULL
```

j'ai ensuite créer un tableau ou je compte stocker les symboles, j'ai défini 3 méthodes

que j'ai décrit que fait chacune en commentaire, je comptais inclure ce fichier dans lex après de faire l'implémentation des méthodes, après un long essai je n'ai malheureusement pas pu aller plus loin que ça.