INSTITUT SUPERIEUR INFORMATIQUE ISI
المعهد العالي للإعلامية

Techniques de compilation

FEDDEOUI Wiem
BEN KHLIFA Islem
BEN AHMED Rihem

01ING04

Enseignant : MOGAADI Hayat

Etablissement / Formation : L'institut supérieur d'informatique

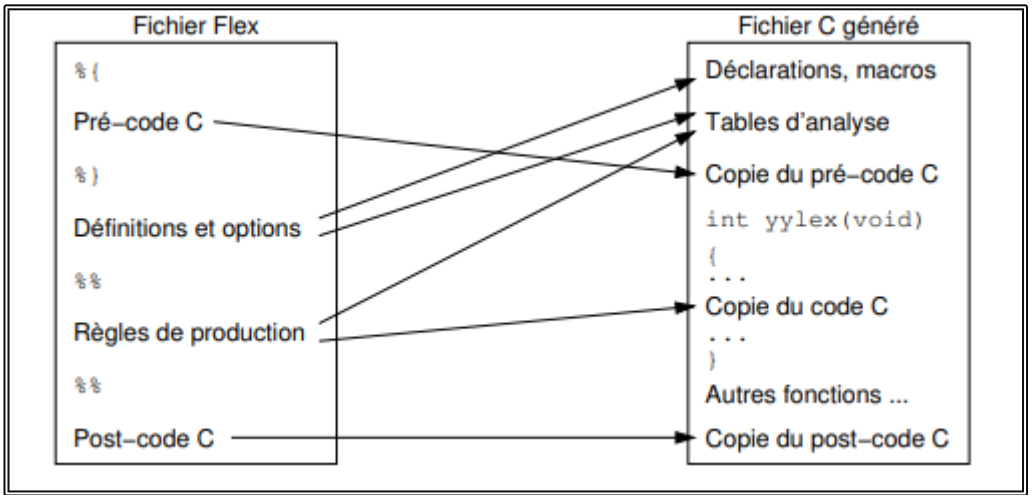
I. Objectif de projet

L'objectif de ce projet de compilation est de se familiariser aux commandes **flex** et **bison,** qui sont les deux outils de compilation par défaut sur les systèmes Unix depuis plusieurs décennies, afin de développer un interpréteur d'expressions mathématiques qui permet de calculer la valeur d'une expression formée de nombres réels et des opérateurs usuels +, -, * et /. Puis on va enrichir cet interpréteur avec les les spécifications nécessaires (lexicale et syntaxique) pour qu'il devient un interrupteur pour un langage de programmation simple acceptant des variables et des structures simples et itératives.

II. Analyse lexicale avec Flex

Le premier outil **flex** (version **gnu** de la commande **lex**) construit un ***analyseur lexical*** à partir d'un ensemble de ***règles/actions*** décrites par des ***expressions régulières***.

A. Structure d'un programme Flex :

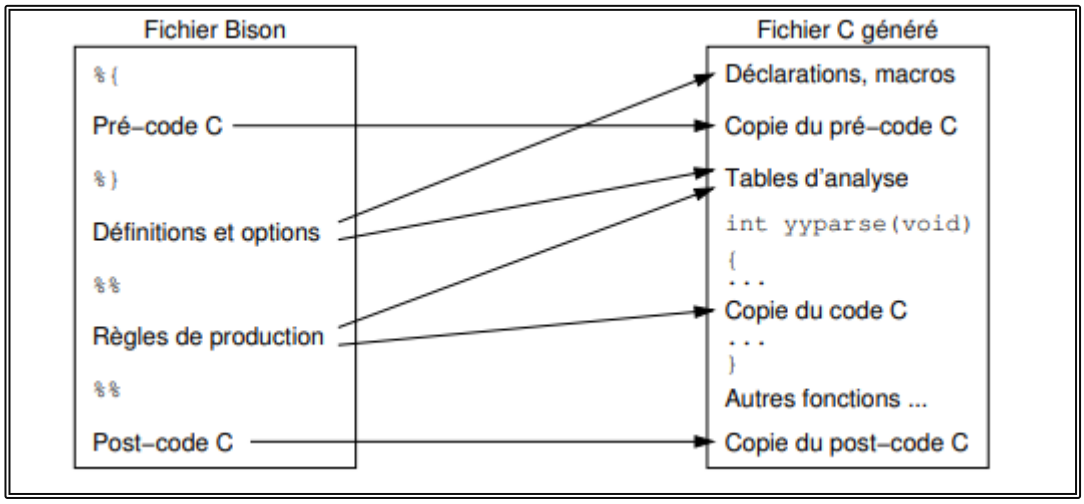


- **Pré/post-code C :** du C tout à fait classique .
- **Options :** *%quelquechose* pour paramétrer le fonctionnement .
- **Définitions :** Expressions rationnelles auxquelles on attribue un nom .
- **Règles de production :** Associations ER → code C à exécuter.

III. Analyse syntaxique avec Bison

Le second outil **bison** est un compilateur de compilateur, version gnu de la célèbre commande **yacc** acronyme de « yet another compiler of compilers ». Il construit un compilateur d'un langage décrit par un ensemble de règles et actions d'une ***grammaire LARL*** sous une forme proche de la ***forme BNF*** de Backus-Naur.

A. Structure d'un programme Bison :



- **Pré/post-code C :** du C tout à fait classique .
- **Options :** *%quelquechose* pour paramétrer le fonctionnement .
- **Définitions :** *%autrechose* pour définir les lexèmes, les priorités . . .
- **Règles de production** : Règles de grammaires et code C à exécuter.

IV. Association de Flex et Bison

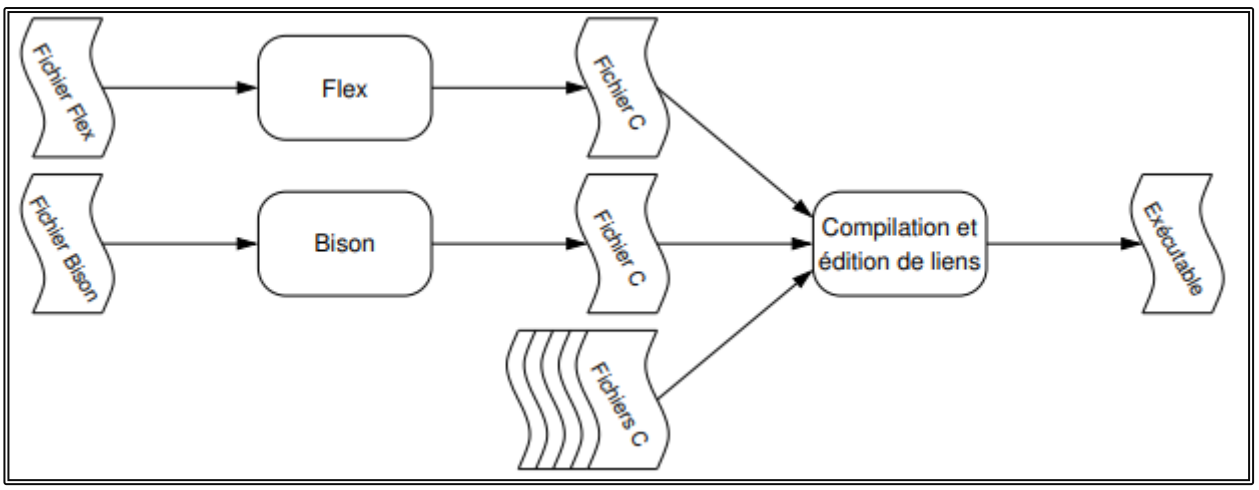
Flex et Bison sont des outils permettant de créer des programmes qui traitent des entrées structurées. À l'origine, ces outils étaient destinés à la création de compilateurs, mais ils se sont avérés utiles dans de nombreux autres domaines.
Pour que Flex et Bison puissent se parler, il suffit de leur faire partager la même liste de token disponibles. Cette tâche peut être facilement réalisée en ajoutant cette ligne de code dans le prologue du fichier flex:

#include "nom_fichier.tab.h"

Et du fichier bison :

#include "lex.yy.c"

A. Principe des outils



- Flex fourni **les lexèmes** à Bison.
- Bison invoque la fonction **yylex()** produite par Flex .
- yylex() doit renvoyer des constantes connues de Bison → %token IDENT INTEGER ... dans les définitions de Bison.
- Bison genère **un .h** definissant ces constantes (et d'autres choses) .
- Le **prè-code C** de Flex inclu **ce .h** .

B. Etapes de la construction :

> bison -d prog.y
→ Produit le code C depuis le fichier Bison prog.y
→ Option -d pour générer le .h

> flex prog.l
→ Produit le code C depuis le fichier Flex prog.l
→ Le pré-code C doit inclure prog.h

> gcc prog.tab.c prog
→ Générer un executable prog.exe

V. Réalisation de Mini-Projet

A. Enoncé de sujet 1

A) En utilisant les outils Flex/Bison, développer un interpréteur d'expressions mathématiques permettant de calculer la valeur d'une expression formée de nombres réels et des opérateurs usuels +, -, * et /. Vous avez à :
1. Donner la spécification de l'analyseur lexical correspondant (fichier Flex).
2. Développer le fichier de spécification Bison définissant l'analyseur syntaxique.
3. Enrichir l'analyseur syntaxique en rajoutant les actions sémantiques permettant d'évaluer une expression donnée en entrée.

B) A ce niveau, on souhaite concevoir et réaliser un interpréteur pour un langage de programmation simple acceptant des variables et des structures simples et itératives telles que :
- des instructions d'affectation,
- des structures conditionnelles de la forme si-alors-finsi et si-alors-sinon-finsi,
- des structures itératives de type boucle Pour, Répeter, Tantque. soit l'exemple d'entrée:
Si x=y Alors z := 12+24 Finsi
Si x=y Alors z := 3 Sinon x :=2 Finsi
1. Enrichir l'interpréteur développé tout en précisant les spécifications nécessaires au niveau lexical (fichier FLex) et au niveau syntaxique (fichier Bison).
2. Vous avez à assurer quelques fonctions de contrôle (aspect sémantiques):
- évaluer les expressions,
- détection de l'erreur 'manque de finsi' :
Si a=a Alors b := 12+24 //afficher à l'utilisateur: erreur syntaxique, manque du finsi
- détection de boucle infinie,...

Partie A :

- La compilation des fichiers **calc.l** et **calc.y**, puis l'exécution du programme mini :

```
C:\Users\wiemf\Desktop\wiem>flex calc.l

C:\Users\wiemf\Desktop\wiem>bison -d calc.y
calc.y: conflicts: 58 shift/reduce

C:\Users\wiemf\Desktop\wiem>gcc calc.tab.c -o mini

C:\Users\wiemf\Desktop\wiem>mini
```

- On teste une expression mathématique formée de nombres réels et des opérateurs usuels +, -, * et /

```
C:\Users\wiemf\Desktop\wiem>mini
9*2+3.1/5^9=
The result of line N1 is 18.000002
```

La priorité est définie par l'ordre des déclarations des tokens dans des directives **%letf.** Pour cela, si l'expression est **sans parenthèses** l'ordre choisit est du gauche vers la droite.

- On teste maintenant une expression mathématique contenant **les parenthèses :**

```
-(5+2)-2.6*3^2/7=
The result of line N2 is -10.342857
```

- On peut aussi tester les variables signées :

```
-2=
The result of line N5 is -2.000000
(-2)=
The result of line N6 is -2.000000
```

- Lorsqu'on test la division sur 0, une erreur est générée automatiquement, et le résultat sera la même valeur que le numérateur :

```
C:\Users\wiemf\Desktop\wiem>mini
5/0=
**ERROR Can not divide by zero
**Warning : This expression: { /0.000000 }  is not considered !
The result of line N1 is 5.000000
```

- Lorsqu'on test un réel sans la partie entière , une erreur syntaxique est générée automatiquement.

```
.17
**ERROR: Unexpected input . in line line 2 :
**Error happend : syntax error
```

Partie 2 :

✓ Les instructions d'affectation :

Pour que l'instruction d'affectation soit acceptée elle doit être sous cette forme :

Identifiant := expression mathématique ;

```
C:\Users\wiemf\Desktop\wiem>mini
v:=6;
SUCCESSUF Affectation
v:=6; b:=6;
SUCCESSUF Affectation
SUCCESSUF Affectation
v:=6; b:=6; var:=9;
SUCCESSUF Affectation
SUCCESSUF Affectation
SUCCESSUF Affectation
```

- On peut écrire plusieurs instructions dans la même ligne.

```
y:=6; x:=-6; z:=9*3+4;
SUCCESSUF Affectation
SUCCESSUF Affectation
SUCCESSUF Affectation
```

✓ Les structures conditionnelles

- On teste la structure conditionnelle simple : ***si-alors-finsi***
  Sans variables :

```
C:\Users\wiemf\Desktop\wiem>mini
si (5>6) alors 6*2 finsi
Struct Si Accepted
INVALID Condition
si (5<6) alors 6*2 finsi
Struct Si Accepted
Si result 12.000000
```

Avec variables :

```
c:=1; b:=9; si c>b alors 9+4*6 finsi
SUCCESSUF Affectation
SUCCESSUF Affectation
Struct Si Accepted
INVALID Condition
c:=1; b:=9; si c<b alors 9+4*6 finsi
SUCCESSUF Affectation
SUCCESSUF Affectation
Struct Si Accepted
Si result 33.000000
c:=1; b:=9; si c<=b alors 9+4*6 finsi
SUCCESSUF Affectation
SUCCESSUF Affectation
Struct Si Accepted
Si result 33.000000
c:=1; b:=9; si c>=b alors 9+4*6 finsi
SUCCESSUF Affectation
SUCCESSUF Affectation
Struct Si Accepted
INVALID Condition
c:=6; b:=9; si c!=b alors 9+4*6 finsi
SUCCESSUF Affectation
SUCCESSUF Affectation
Struct Si Accepted
Si result 33.000000
c:=6; b:=9; si c==b alors 9+4*6 finsi
SUCCESSUF Affectation
SUCCESSUF Affectation
Struct Si Accepted
INVALID Condition
```

➢ On teste la structure conditionnelle complexe : ***si-alors-sinon-finsi,***

```
var2:=4.2; var1:=7/2; si c==b alors 9+4*6 sinon -1.5 finsi
SUCCESSUF Affectation
SUCCESSUF Affectation
Struct Si/Sinon Accepted
Sinon result -1.500000
var2:=4.2; var1:=7/2; si c!=b alors 9+4*6 sinon -1.5 finsi
SUCCESSUF Affectation
SUCCESSUF Affectation
Struct Si/Sinon Accepted
Si result 33.000000
var2:=4.2; var1:=7/2; si c>=b alors 9+4*6 sinon -1.5 finsi
SUCCESSUF Affectation
SUCCESSUF Affectation
Struct Si/Sinon Accepted
Si result 33.000000
var2:=4.2; var1:=7/2; si c<=b alors 9+4*6 sinon -1.5 finsi
SUCCESSUF Affectation
SUCCESSUF Affectation
Struct Si/Sinon Accepted
Sinon result -1.500000
var2:=4.2; var1:=7/2; si c<b alors 9+4*6 sinon -1.5 finsi
SUCCESSUF Affectation
SUCCESSUF Affectation
Struct Si/Sinon Accepted
Sinon result -1.500000
var2:=4.2; var1:=7/2; si c>b alors 9+4*6 sinon -1.5 finsi
SUCCESSUF Affectation
SUCCESSUF Affectation
Struct Si/Sinon Accepted
Si result 33.000000
```

➢ On teste la structure conditionnelle simple: ***si-alors-sinonSi-alors-sinon-finsi,***

```
C:\Users\wiemf\Desktop\wiem>mini
SI (6>7) alors 1 sinon_si (6<10) alors 2 sinon 3 finsi
Struct Si/Sinon_Si/Sinon Accepted
Sinon Si result 2.000000
SI (6>7) alors 1 sinon_si (6==10) alors 2 sinon 3 finsi
Struct Si/Sinon_Si/Sinon Accepted
Sinon result 3.000000
SI (6>1) alors 1 sinon_si (6==10) alors 2 sinon 3 finsi
Struct Si/Sinon_Si/Sinon Accepted
Si result 1.000000
```

✓ Les structures itératives de type boucle

➢ On teste la boucle : ***For***
Sans variables :

```
C:\Users\wiemf\Desktop\wiem>mini
pour i de 1 a 3 faire c:=2^4; finpour
Pour Struct Accepted
FOR RESULT: 16.000000
FOR RESULT: 16.000000
FOR RESULT: 16.000000
```

Dans l'exemple suivant il y a une erreur syntaxique : au lieu d'écrire « a » on a tapé « & » :

```
pour i de 1 & 3 faire c:=2^4; finpour
**ERROR: Unexpected input & in line line 8 :
**Error happend : syntax error
```

Dans l'exemple ci-dessous il y a une erreur syntaxique : le manque de « finsi » :

```
C:\Users\user\Desktop\w>prog.exe
si (6>5) alors 5+5 finsi
Struct Si Accepted
Si result 10.000000
si (6>5) alors 5+5
manque de finsi^C
C:\Users\user\Desktop\w>
```

Avec les variables :

```
c:=1; b:=9; POUR i de c a b Faire 9+4*6 finpour
SUCCESSUF Affectation
SUCCESSUF Affectation
Pour Accepted
FOR RESULT: 33.000000
FOR RESULT: 33.000000
FOR RESULT: 33.000000
FOR RESULT: 33.000000
FOR RESULT: 33.000000
FOR RESULT: 33.000000
FOR RESULT: 33.000000
FOR RESULT: 33.000000
FOR RESULT: 33.000000
```

- On teste la boucle : ***While***
  Avec variables :

```
C:\Users\wiemf\Desktop\wiem>mini
z:=1; m:=9; tantque z>m Faire -2+4.3 fintantque
SUCCESSUF Affectation
SUCCESSUF Affectation
Tantque Struct Accepted
Condition not Checked
```

Boucle **while infini** :

```
C:\Users\wiemf\Desktop\wiem>mini
z:=1; m:=9; tantque z<m Faire -2+4.3 fintantque
```

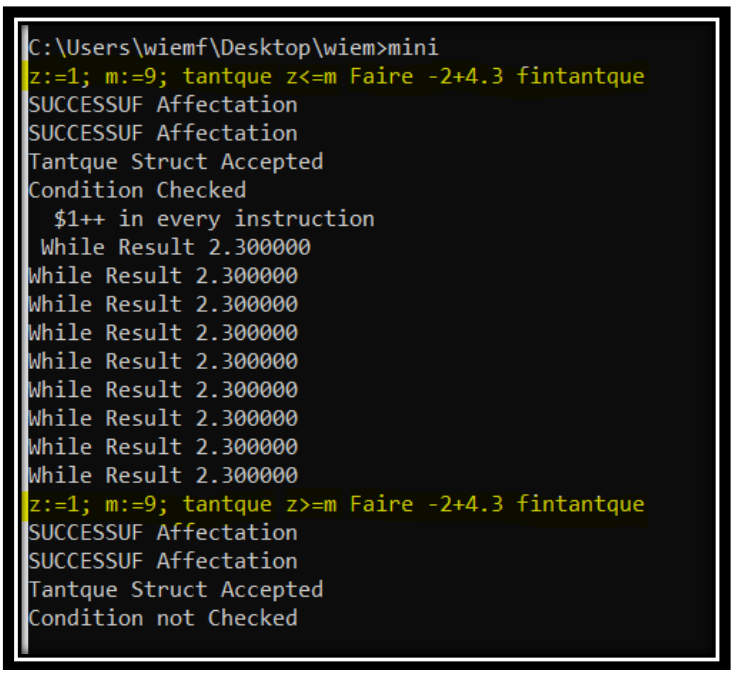
Le résultat sera comme le suivant :

```
Invite de commandes - mini
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
**ERROR: Boucle INFINIE Condition $1>$2 is always checked try $1++ :While Result 2.300000
```

En effet la condition $1<$2 qui est égal à 1<5 dans cet exemple est tjrs valide et de plus on a choisi de décrémenter la valeur de $1=1 d'où il est impossible de valider la condition $1>=$2 pour arrêter la boucle.

Pour cette erreur on a proposé une solution qui est l'incrémentation de la valeur de $1=1 pour arriver jusqu'à 5 ou bien décrémenter $2=5 pour arriver à 1. Grace à cette solution la boucle infinie va disparaitre et le problème sera résolu.

Si la condition de la boucle while est bien vérifiée alors la boucle va de 1 à 9 (c'est-à-dire 8 fois) qui est montré par l'exemple suivant :



```
C:\Users\wiemf\Desktop\wiem>mini
z:=1; m:=9; tantque z==m Faire -2+4.3 fintantque
SUCCESSUF Affectation
SUCCESSUF Affectation
Tantque Struct Accepted
Condition not Checked

C:\Users\wiemf\Desktop\wiem>mini
z:=1; m:=9; tantque z!=m Faire -2+4.3 fintantque
SUCCESSUF Affectation
SUCCESSUF Affectation
Tantque Struct Accepted
Condition Checked
$1++ in every instruction
 While Result 2.300000
$1++ in every instruction
 While Result 2.300000
$1++ in every instruction
 While Result 2.300000
$1++ in every instruction
 While Result 2.300000
$1++ in every instruction
 While Result 2.300000
$1++ in every instruction
 While Result 2.300000
$1++ in every instruction
 While Result 2.300000
$1++ in every instruction
 While Result 2.300000
```

- On teste la boucle : ***Répéter jusqu'à***

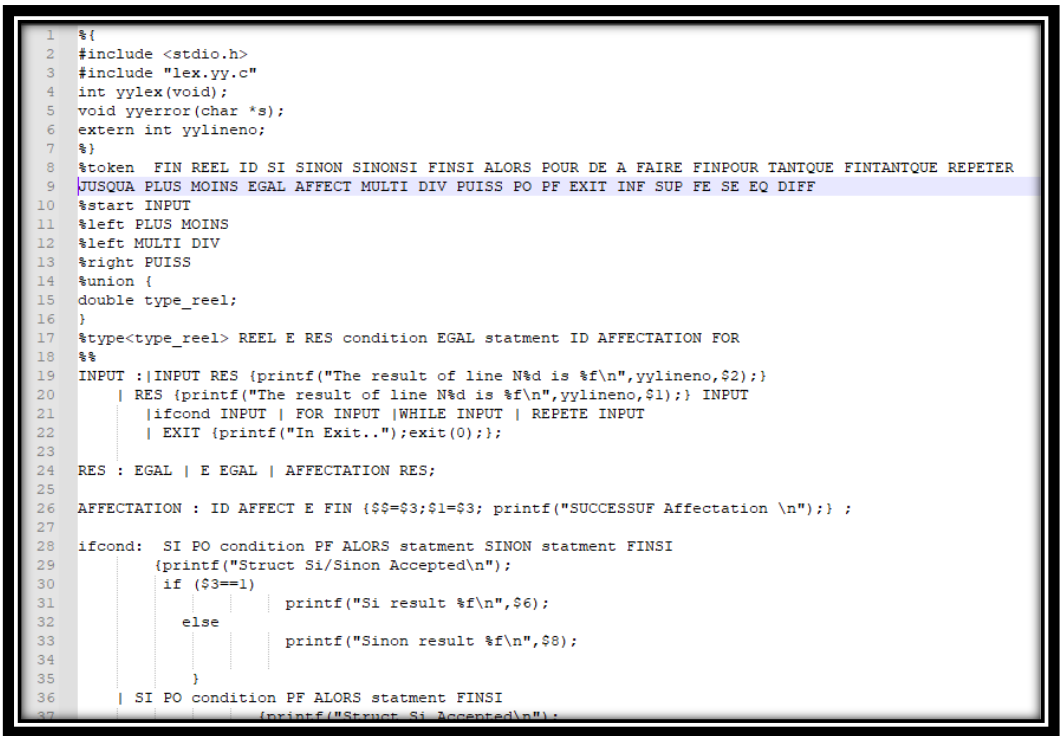
```
C:\Users\wiemf\Desktop\wiem>mini
repeter 2/9+4*7-6 jusqua 9>7;
REPETE Struct Accepted
repeter b:=2/9+4*7-6; jusqua 9>7;
REPETE Struct Accepted
```

- Pour quitter le programme on doit taper **exit**

```
C:\Users\wiemf\Desktop\wiem>mini
exit
In Exit..
C:\Users\wiemf\Desktop\wiem>
```

Description des fichiers sources :

La capture suivante présente une partie du code source du fichier **calc.y**



```
%{
#include <stdio.h>
#include "lex.yy.c"
int yylex(void);
void yyerror(char *s);
extern int yylineno;
%}
%token  FIN REEL ID SI SINON SINONSI FINSI ALORS POUR DE A FAIRE FINPOUR TANTQUE FINTANTQUE REPETER
JUSQUA PLUS MOINS EGAL AFFECT MULTI DIV PUISS PO PF EXIT INF SUP FE SE EQ DIFF
%start INPUT
%left PLUS MOINS
%left MULTI DIV
%right PUISS
%union {
double type_reel;
}
%type<type_reel> REEL E RES condition EGAL statment ID AFFECTATION FOR
%%
INPUT :|INPUT RES {printf("The result of line N%d is %f\n",yylineno,$2);}
    | RES {printf("The result of line N%d is %f\n",yylineno,$1);} INPUT
      |ifcond INPUT | FOR INPUT |WHILE INPUT | REPETE INPUT
      | EXIT {printf("In Exit..");exit(0);};

RES : EGAL | E EGAL | AFFECTATION RES;

AFFECTATION : ID AFFECT E FIN {$$=$3;$1=$3; printf("SUCCESSUF Affectation \n");} ;

ifcond:  SI PO condition PF ALORS statment SINON statment FINSI
        {printf("Struct Si/Sinon Accepted\n");
         if ($3==1)
                        printf("Si result %f\n",$6);
           else
                        printf("Sinon result %f\n",$8);

            }
    | SI PO condition PF ALORS statment FINSI
                (printf("Struct Si Accepted\n");
```

Dans le fichier **calc.y** on a défini les spécifications nécessaires de l'analyseur syntaxique :

- **yylineno** : c'est un compteur de lignes sert à présenter la ligne d'erreur.
- **<type_reel>** : c'est pour definir le type reel.
- **INPUT :** présente l'ensemble de programme.
- **RES :** contient une expression mathématique avec le caractère = à la fin ou bien tout simplement = (résultat vide) ou bien elle contient plusieurs affectations des variables.

- **ifcond :** contient les structures conditionnelles ainsi que l'affichage des résultats et le test sur le condition (SI/SINON_si/sinon) dans cette partie on a détecté l'erreur : manque fin si .

- **condition :** permet de tester les conditions inclus dans les structures conditionnelles, les boucles..
  Si resultat $$ = 1 donc test est vrai sinon test est faux.

- **E :** c'est l' expressions mathématiques avec et sans parenthèses et avec les priorités de + - * / ^ , elle contient une erreur de division sur 0 et une alerte de la non-considération de cette partie si elle vaut 0.

- **statment** : c'est les instructions que nous allons utiliser dans les structures conditionnelles, les boucles..

- **FOR:** définit la structure de la boucle « pour » et contient des actions sémantiques

permettent de tester cette structure et afficher le résultat de la boucle « pour ».

- **WHILE:** définit la structure de la boucle « tant que » est contient des actions sémantiques permettent de tester cette structure et afficher le résultat de la boucle « tant que » si sa condition est vérifiée. Dans cette boucle, on a testé les 2 cas de la boucle infinie.

- **REPETE :** définit la structure de la boucle répéter jusqu'à et afficher un message de validation si cette structure est bien défini via la sémantique.

La capture suivante présente une partie du code source du fichier **calc.l**

```
%option yylineno
%{
#include <stdio.h>
#include<math.h>
#include <stdlib.h>
#include "calc.tab.h"
%}
entier [0-9]+
ident [a-zA-Z_][a-zA-Z0-9_]*
reel ({entier}[\.]{entier}|{entier})
blank [ \t\n]+
Commentaire (#" "*(.)*|"/*"(.)*"*/"|"//"(.)*)
reel_mal [\.]{entier};
%%
{reel} {yylval.type_reel = atof(yytext); return(REEL);}
"<" {return(INF);}
">" {return(SUP);}
"<=" {return(FE);}
">=" {return(SE);}
"==" {return(EQ);}
"!=" {return(DIFF);}
"Si"|"SI"|"si" {return(SI);}
"Sinon"|"SINON"|"sinon" {return(SINON);}
"Sinon_Si"|"SINON_SI"|"sinon_si" {return(SINONSI);}
"FinSi"|"FINSI"|"finsi" {return(FINSI);}
"Pour"|"POUR"|"pour" {return(POUR);}
"De"|"DE"|"de" {return(DE);}
"A"|"a" {return(A);}
"Alors"|"alors"|"ALORS" {return(ALORS);}
"Faire"|"FAIRE"|"faire" {return(FAIRE);}
"FinPour"|"FINPOUR"|"finpour" {return(FINPOUR);}
"Tantque"|"TANTQUE"|"tantque" {return(TANTQUE);}
"FinTantque"|"FINTANTQUE"|"fintantque" {return(FINTANTQUE);}
"Repeter"|"REPETER"|"repeter" {return(REPETER);}
"Jusqua"|"JUSQUA"|"jusqua" {return(JUSQUA);}
";"          {return(FIN);}
"+"     {return(PLUS);}
"-"     {return(MOINS);}
```

Dans le fichier **calc.l** on a défini les spécifications nécessaires de l'analyseur lexical :

- Les nombres entiers
- Les nombres réels et les réels mal définis tout en affichant une erreur exp .3 il doit avoir un entier avant le "." exp 3.3

- Les identifient
- Les espaces , les tabulations les retours à la ligne et les commentaires qui vont être éliminées
- Les opérateurs-rel < > <= >= == != :=
- Les opérateurs-arith + - * / - ^
- Les mots clés des structures conditionnelles et des boucles pour , tantque , repeter jusqu'à
- Les parenthèses ouvrantes et fermentes () et le ; désignant la fin d'une affectation.

RQ : toute autre spécification non déclarée dans calc.l va permettre d'afficher un message d'erreur en indiquant le numéro de la ligne et le caractère mal défini.