



**Université Abdelmalek Essaadi**

Ecole Nationale des Sciences Appliquées Al-Hoceima

**Département Mathématiques et Informatique**

---

## **Rapport de Mini projet**

# **Flex**



# **Bison**

DuarteCorporation

Réalisé Par :

- **FAIZ Oussama**
- **MALIKI Maroua**
- **MAKHLOUF Ahlam**

# Introduction

Ce rapport présente le mini-projet réalisé dans le cadre du cours de Théorie de Langage à l'École Nationale des Sciences Appliquées Al-Hoceima. Le projet se concentre sur la reconnaissance et l'analyse syntaxique des langages de programmation en utilisant des expressions régulières et des grammaires formelles. Les outils utilisés pour ce projet sont Flex et Yacc, permettant de créer des analyseurs lexical et syntaxique pour un langage de programmation simple. Nous allons définir les expressions régulières nécessaires pour identifier les différents éléments du langage, écrire les règles de grammaire correspondantes, et implémenter une table de symboles pour gérer les variables. Enfin, nous illustrerons le processus de compilation et d'exécution des analyseurs développés.

## 1. Les définitions et expressions régulières qui permettront de reconnaître les mots de langage :

Pour reconnaître les mots-clés, opérateurs, identifiants et autres jetons de votre langage, vous devrez définir des expressions régulières pour chacun. Voici les définitions et expressions régulières des éléments que vous avez décrits :

### ➤ Définitions:

- **AFFICHER**: une fonction qui affiche un message sous forme de chaîne de caractères entre guillemets doubles '' et '.
- **AFFICHER nom\_variable**: affiche la valeur de la variable passée en paramètre.
- **REEL nom\_variable**: déclare une variable de type réel (exemple: REEL a; REEL b;) dans la table des symboles.
- **ENTIER nom\_variable**: déclare une variable de type entier dans la table des symboles.
- **LIRE nom\_variable**: lit les entrées et les affecte à la variable.

### ➤ Expressions régulières:

- **['^']\*** : Correspond à toute chaîne de caractères entre guillemets doubles. Utilisé pour reconnaître la fonction AFFICHER.
- **[a-z]+** : Correspond à une ou plusieurs lettres minuscules. Utilisé pour reconnaître les noms de variables.
- **reelentierlire** : Correspond aux mots-clés 'reel', 'entier' ou 'lire'. Utilisé pour reconnaître les déclarations de type et les instructions LIRE.
- **=+\*/-** : Correspond aux opérateurs =, +, -, \*.
- **[0-9]+([.][0-9]+)?** : Correspond à un ou plusieurs chiffres, éventuellement suivis d'un . et de plus de chiffres. Utilisé pour reconnaître les littéraux numériques.
- **;** : Correspond au symbole ; pour reconnaître les séparateurs d'instruction.

### ➤ Donc, les expressions régulières complètes seraient:

- **['^']\*** : Pour la fonction AFFICHER
- **[a-z]+** : Pour les noms de variables
- **reelentierlire** : Pour les déclarations de type et les instructions LIRE
- **=+\*/-** : Pour les opérateurs
- **[0-9]+([.][0-9]+)?** : Pour les littéraux numériques
- **;** : Pour les séparateurs d'instruction

## 2. Les règles de la grammaire de langage :

Pour écrire les règles de la grammaire de ce langage, nous allons définir la syntaxe du langage en utilisant une grammaire formelle :

### ➤ **Terminaux:**

- affiche
- reel
- entier
- =, +, \*, -, /
- Identificateurs (une lettre minuscule)
- Littéraux de chaîne (texte entre guillemets doubles "")
- Nombres réels
- Nombres entiers

### ➤ **Non-terminaux:**

- programme
- instruction
- declaration
- affectation
- expression
- terme
- facteur
- affichage

### 2.1 Grammaire:

#### ➤ *Programme :*

`<programme> ::= <instruction>*`

#### ➤ *Instruction*

Une instruction peut être une déclaration, une affectation ou un affichage :

`<instruction> ::= <declaration> | <affectation> | <affichage>`

#### ➤ *Déclaration*

Une déclaration de variable peut être de type **reel** ou **entier** :

`<declaration> ::= "reel" <identificateur> ";" | "entier" <identificateur> ";"`

➤ ***Affectation***

Une affectation consiste à assigner une valeur (expression) à une variable :

$\langle \text{affectation} \rangle ::= \langle \text{identificateur} \rangle "=" \langle \text{expression} \rangle ";"$

➤ ***Expression***

Une expression peut être une addition ou une soustraction de termes :

$\langle \text{expression} \rangle ::= \langle \text{terme} \rangle (( "+" | "-" ) \langle \text{terme} \rangle )^*$

➤ ***Terme***

$\langle \text{terme} \rangle ::= \langle \text{facteur} \rangle (( "*" | "/" ) \langle \text{facteur} \rangle )^*$

➤ ***Facteur***

Un facteur peut être un identificateur, un nombre réel, un nombre entier ou une expression entre parenthèses :

$\langle \text{facteur} \rangle ::= \langle \text{identificateur} \rangle | \langle \text{nombre\_reel} \rangle | \langle \text{nombre\_entier} \rangle | "(" \langle \text{expression} \rangle ")"$

➤ ***Affichage***

L'affichage peut être d'une chaîne de caractères ou de la valeur d'une variable :

$\langle \text{affichage} \rangle ::= "affiche" \langle \text{chaîne} \rangle ";" | "affiche" \langle \text{identificateur} \rangle ";"$

➤ ***Identificateur***

Un identificateur est une lettre minuscule :

$\langle \text{identificateur} \rangle ::= [a-z]$

➤ ***Chaîne de caractères***

Une chaîne de caractères est du texte entre guillemets doubles :

$\langle \text{chaîne} \rangle ::= "\" [^\"]^* "\"$

➤ ***Nombre réel***

Un nombre réel est une séquence de chiffres avec un point décimal :

$\langle \text{nombre\_reel} \rangle ::= [0-9]^+ "." [0-9]^+$

➤ ***Nombre entier***

Un nombre entier est une séquence de chiffres :

$\langle \text{nombre\_entier} \rangle ::= [0-9]^+$

### 3. Implémenter une table de symbole qui permettra d'enregistrer les différentes variables utilisées dans le programme :

#### ➤ Déclaration :

Nous allons d'abord définir la structure *sym* :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  struct sym {
6      char id;
7      int type;
8      int intValue;
9      float floatValue;
10     struct sym *suivant;
11 };
12
13 struct sym *table = NULL;
```

#### ➤ Fonction d'ajout :

```
1
2  void ajouter_symbole(char id, int type) {
3      struct sym *nouveau = (struct sym *)malloc(sizeof(struct sym));
4      nouveau->id = id;
5      nouveau->type = type;
6      nouveau->intValue = 0;
7      nouveau->floatValue = 0.0;
8      nouveau->suivant = table;
9      table = nouveau;
10 }
```

➤ Fonction de recherche :

```
1
2 struct sym *rechercher_symbole(char id) {
3     struct sym *actuel = table;
4     while (actuel != NULL) {
5         if (actuel->id == id) {
6             return actuel;
7         }
8         actuel = actuel->suivant;
9     }
10    return NULL;
11 }
```

➤ Fonction de modification :

```
1
2 void modifier_valeur(char id, int intValue, float floatValue) {
3     struct sym *symbole = rechercher_symbole(id);
4     if (symbole != NULL) {
5         if (symbole->type == 1) {
6             symbole->intValue = intValue;
7         } else if (symbole->type == 2) {
8             symbole->floatValue = floatValue;
9         }
10    } else {
11        printf("Erreur : Symbole %c non trouvé\n", id);
12    }
13 }
14
```

## 4. Implémenter en utilisant Flex et Yacc les analyseurs lexical et syntaxique :

Pour implémenter les analyseurs lexical et syntaxique en utilisant Flex et Yacc, nous devons suivre quelques étapes clés :

- définir les règles de grammaire.
- écrire les actions associées dans le fichier Yacc.
- créer le fichier Flex pour la génération des Tokens.

### 4.1 Définir le Fichier Flex (lexer.l) :

Le fichier Flex scanne l'entrée et produit des jetons pour Yacc :

```
%{
#include "y.tab.h"
%}

%%

"affiche"      { return AFFICHE; }
"reel"         { return REEL; }
"entier"        { return ENTI; }
"="            { return EQUALS; }
"+"           { return PLUS; }
"*"            { return MULTIPLY; }
"-"            { return MINUS; }
"/"            { return DIVIDE; }
[a-z]          { yylval.id = yytext[0]; return IDENTIFIER; }
\"[^\"]*\"      { yylval.string = strdup(yytext); return STRING_LITERAL; }
[0-9]+\.[0-9]+ { yylval.real = atof(yytext); return REAL_LITERAL; }
[0-9]+         { yylval.integer = atoi(yytext); return INTEGER_LITERAL; }
[ \t\n]        { }
.              { printf("caractere inconnue: %s\n", yytext); }

%%

int yywrap(void) {
    return 1;
}
```

### 4.2 Définir le Fichier Yacc (parser.y) :

Le fichier Yacc définit la grammaire et les actions associées :





```
1  %{
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  struct sym {
7      char id;
8      int type;
9      int intValue;
10     float floatValue;
11     struct sym *suivant;
12 };
13
14 struct sym *table = NULL;
15
16 void ajouter_symbole(char id, int type);
17 struct sym* rechercher_symbole(char id);
18 void modifier_valeur(char id, int intValue, float floatValue);
19 void afficher_table();
20
21 extern int yylex(void);
22 extern int yyparse(void);
23 extern FILE *yyin;
24 void yyerror(const char *s);
25
26 %}
27
28 %union {
29     char id;
30     char *string;
31     int integer;
32     float real;
33 }
34
```



```
1 %token <id> IDENTIFIER
2 %token <string> STRING_LITERAL
3 %token <integer> INTEGER_LITERAL
4 %token <real> REAL_LITERAL
5
6 %token AFFICHE REEL ENTI EQUALS PLUS MINUS MULTIPLY DIVIDE
7
8 %%
9
10 programme:
11     instruction_list
12     ;
13
14 instruction_list:
15     instruction_list instruction
16     |
17     ;
18
19 instruction:
20     declaration
21     | affectation
22     | affichage
23     ;
24
25 declaration:
26     REEL IDENTIFIER ';' { ajouter_symbole($2, 2); }
27     | ENTI IDENTIFIER ';' { ajouter_symbole($2, 1); }
28     ;
29
```

```

1  affectation:
2      IDENTIFIER EQUALS expression ';' {
3          struct sym *symbole = rechercher_symbole($1);
4          if (symbole) {
5              if (symbole->type == 1) {
6                  if ($3.type == 1) {
7                      modifier_valeur($1, $3.integer, 0.0);
8                  } else {
9                      yyerror("Type mismatch in assignment to integer variable");
10                 }
11             } else if (symbole->type == 2) {
12                 if ($3.type == 2) {
13                     modifier_valeur($1, 0, $3.real);
14                 } else if ($3.type == 1) {
15                     modifier_valeur($1, 0, $3.integer);
16                 }
17             }
18         } else {
19             yyerror("Variable not declared");
20         }
21     }
22     ;
23
24 expression:
25     term { $$ = $1; }
26     | expression PLUS term {
27         if ($1.type == 1 && $3.type == 1) {
28             $$ = (union YYSTYPE){.integer = $1.integer + $3.integer, .type = 1};
29         } else if ($1.type == 2 && $3.type == 2) {
30             $$ = (union YYSTYPE){.real = $1.real + $3.real, .type = 2};
31         } else {
32             yyerror("Type mismatch in expression");
33         }
34     }
35     | expression MINUS term {
36         if ($1.type == 1 && $3.type == 1) {
37             $$ = (union YYSTYPE){.integer = $1.integer - $3.integer, .type = 1};
38         } else if ($1.type == 2 && $3.type == 2) {
39             $$ = (union YYSTYPE){.real = $1.real - $3.real, .type = 2};
40         } else {
41             yyerror("Type mismatch in expression");
42         }
43     }
44     ;

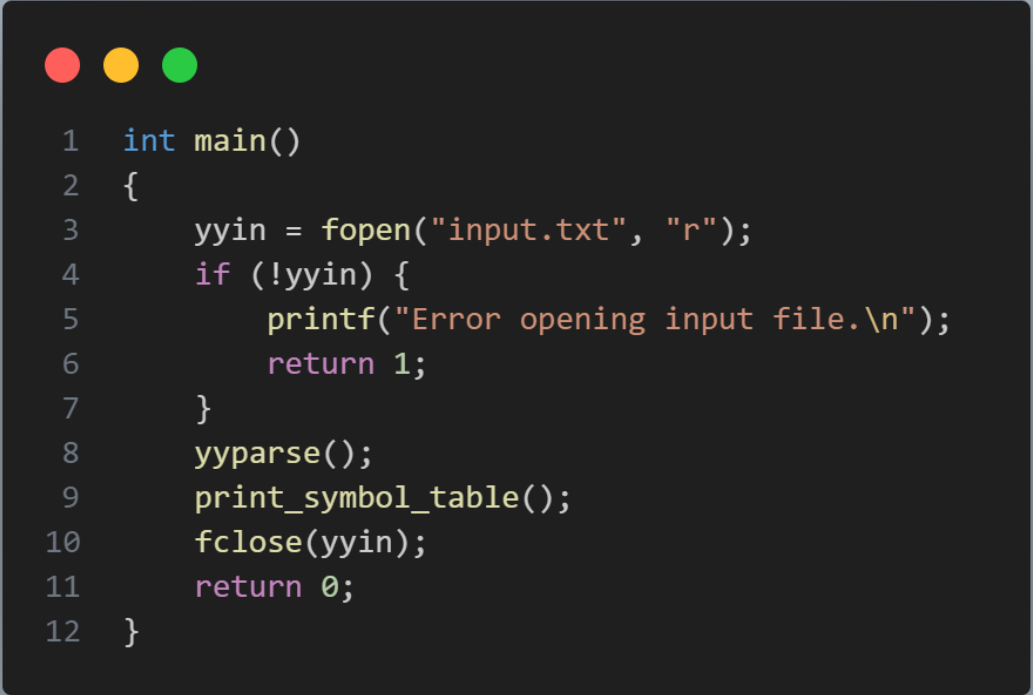
```

```

1
2 term:
3     factor { $$ = $1; }
4     | term MULTIPLY factor {
5         if ($1.type == 1 && $3.type == 1) {
6             $$ = (union YYSTYPE){.integer = $1.integer * $3.integer, .type = 1};
7         } else if ($1.type == 2 && $3.type == 2) {
8             $$ = (union YYSTYPE){.real = $1.real * $3.real, .type = 2};
9         } else {
10            yyerror("Type mismatch in term");
11        }
12    }
13    | term DIVIDE factor {
14        if ($1.type == 1 && $3.type == 1) {
15            $$ = (union YYSTYPE){.integer = $1.integer / $3.integer, .type = 1};
16        } else if ($1.type == 2 && $3.type == 2) {
17            $$ = (union YYSTYPE){.real = $1.real / $3.real, .type = 2};
18        } else {
19            yyerror("Type mismatch in term");
20        }
21    }
22    ;
23
24 factor:
25     IDENTIFIER {
26         struct sym *symbole = rechercher_symbole($1);
27         if (symbole) {
28             if (symbole->type == 1) {
29                 $$ = (union YYSTYPE){.integer = symbole->intValue, .type = 1};
30             } else if (symbole->type == 2) {
31                 $$ = (union YYSTYPE){.real = symbole->floatValue, .type = 2};
32             }
33         } else {
34             yyerror("Variable not declared");
35         }
36     }
37     | INTEGER_LITERAL { $$ = (union YYSTYPE){.integer = $1, .type = 1}; }
38     | REAL_LITERAL { $$ = (union YYSTYPE){.real = $1, .type = 2}; }
39     | '(' expression ')' { $$ = $2; }
40     ;
41
42 affichage:
43     AFFICHE STRING_LITERAL ';' { printf("%s\n", $2); }
44     | AFFICHE IDENTIFIER ';' {
45         struct sym *symbole = rechercher_symbole($2);
46         if (symbole) {
47             if (symbole->type == 1) {
48                 printf("%d\n", symbole->intValue);
49             } else if (symbole->type == 2) {
50                 printf("%f\n", symbole->floatValue);
51             }
52         } else {
53             yyerror("Variable not declared");
54         }
55     }
56     ;
57
58 %%

```

Programme principale :



```
1  int main()
2  {
3      yyin = fopen("input.txt", "r");
4      if (!yyin) {
5          printf("Error opening input file.\n");
6          return 1;
7      }
8      yyparse();
9      print_symbol_table();
10     fclose(yyin);
11     return 0;
12 }
```

## 5. Compilation et Exécution :

Compilez les fichiers `lexer.l` et `parser.y` avec Flex et Bison pour générer les fichiers C correspondants avec les commandes ci-dessous:

`flex lexer.l`

`bison -d parser.y`

## Conclusion

En conclusion, ce mini-projet a permis de mettre en pratique les concepts théoriques de la reconnaissance de langages formels et de l'analyse syntaxique. L'utilisation de Flex et Yacc a démontré leur efficacité dans la construction d'analyseurs lexicaux et syntaxiques robustes. La définition précise des expressions régulières et des règles de grammaire a facilité la reconnaissance et le traitement des éléments du langage. De plus, l'implémentation de la table de symboles a illustré l'importance de la gestion des variables dans un langage de programmation. Ce projet constitue une étape significative dans la compréhension et l'application des principes de la théorie des langages et de la compilation.