# Documentation LEX and YACC

Parser.y file:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define YYDEBUG 1
%}


%token INT_CONST
%token DOUBLE_CONST
%token STRING_CONST
%token IDENTIFIER



%token plus
%token minus
%token mul
%token division
%token mod
%token less
%token lessOrEqual
%token equal
%token different
%token greaterOrEqual
```

```
%token greater
%token assign


%token READ
%token WRITE
%token IF
%token ELSE
%token WHILELOOP
%token FORLOOP
%token INT
%token DOUBLE
%token STRING


%token openBrace
%token closeBrace
%token openBracket
%token closeBracket
%token openParenthese
%token closeParenthese
%token comma
%token semicolon
```

```
%%
program : decllist cmpdstmt
          ;
decllist : declaration semicolon
            | declaration semicolon decllist
            ;
declaration : type listDeclaration
            | type IDENTIFIER
            ;
type :  INT
          | STRING
          | DOUBLE
      ;
listDeclaration : IDENTIFIER openBracket listInt closeBracket
          | IDENTIFIER openBracket listString closeBracket
          | IDENTIFIER openBracket closeBracket
          ;
listInt : INT_CONST comma listInt
          | INT_CONST
          ;
listString : STRING_CONST comma listString
          | STRING_CONST
          ;
cmpdstmt : stmtlist
          ;
stmtlist : stmt semicolon stmtlist
          | stmt semicolon
          ;
stmt : simplstmt
```

```
        | structstmt
        ;
simplstmt : assignstmt
        | iostmt
        ;
assignstmt : IDENTIFIER assign expression
        ;
expression : term plus expression
        | term minus expression
        | term
        ;
term : factor mul term
        | factor division term
        | factor mod term
        | factor
        ;
factor : openParenthese expression closeParenthese
        | IDENTIFIER
        | INT_CONST
        | DOUBLE_CONST
        | IDENTIFIER openBracket expression closeBracket
        ;
iostmt : READ IDENTIFIER
        | WRITE IDENTIFIER
        | WRITE STRING_CONST
        | WRITE INT_CONST
        | READ IDENTIFIER openBracket expression closeBracket
        ;
structstmt : ifstmt
```

```
            | whilestmt
            | forstmt
            ;
ifstmt : ifsimplestmt
            | ifelsestmt
            ;
ifsimplestmt : IF openParenthese condition closeParenthese openBrace stmtlist closeBrace
            ;
ifelsestmt : IF openParenthese condition closeParenthese openBrace stmtlist closeBrace ELSE
openBrace stmtlist closeBrace
            ;
whilestmt : WHILELOOP openParenthese condition closeParenthese openBrace stmtlist
closeBrace
            ;
forstmt : FORLOOP openParenthese assignstmt semicolon condition semicolon assignstmt
closeParenthese openBrace stmtlist closeBrace
            ;
condition : expression less expression
            | expression lessOrEqual expression
            | expression equal expression
            | expression different expression
            | expression greaterOrEqual expression
            | expression greater expression
            ;

%%

yyerror(char *s)
{
            printf("%s\n",s);
}


extern FILE *yyin;


int main(int argc, char** argv)
{
   if (argc > 1) yyin = fopen(argv[1], "r");
   if (argc > 2 && !strcmp(argv[2], "-d")) yydebug = 1;
   if (!yyparse()) fprintf(stderr, "\tProgram is syntactically correct.\n");
   return 0;
}
```

## 1. Introduction:

Flex and Bison are tools used in conjunction for lexical analysis (Flex) and parsing (Bison) in the construction of compilers and interpreters. They facilitate the process of creating a scanner and a parser, respectively, for processing programming language source code.

## 2. Lex and Flex:

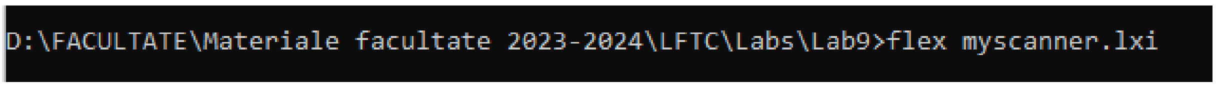
Lex typically refers to the original lexical analyzer generator, while its modern counterpart is Flex (Fast Lexical Analyzer Generator). Flex is an enhanced version of Lex, providing more features and better performance.

Flex (Fast Lexical Analyzer Generator) is a tool for generating lexical analyzers (scanners) based on regular expressions. It reads an input file containing specifications in the form of regular expressions and corresponding actions. The output is a C source code file that can be compiled to create a scanner.

In our case, the Flex file is named myscanner.lxi. The specified regular expressions define the rules for identifying tokens in your programming language.

To generate the scanner, you use the following command in the command prompt:

**$ flex myscanner.lxi**



```
D:\FACULTATE\Materiale facultate 2023-2024\LFTC\Labs\Lab9>flex myscanner.lxi
```

This command processes the Flex file and generates a C source code file named lex.yy.c.

## 3. Bison:

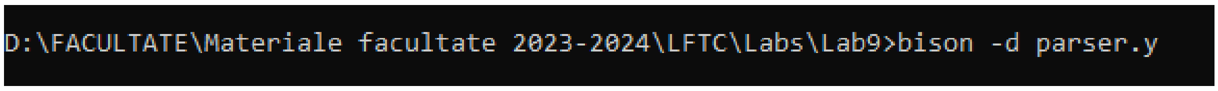
Yacc is the original parser generator, and its modern equivalent is Bison. Bison offers improvements and additional features over Yacc and is widely used for generating parsers in the construction of compilers and interpreters.

Bison is a parser generator that takes a formal grammar file as input and generates a parser in C. The grammar file contains rules specifying the syntax of the programming language.

In our case, the Bison file is named parser.y. It defines the grammar rules for your language, including how different language constructs are parsed.

To generate the parser, you use the following command in the command prompt:

**$ bison -d parser.y**



```
D:\FACULTATE\Materiale facultate 2023-2024\LFTC\Labs\Lab9>bison -d parser.y
```

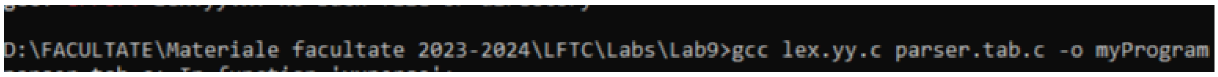
This command processes the Bison file and generates two files: parser.tab.c (the parser source code) and parser.tab.h (the parser header file).

## 4. Compilation:

After generating the scanner and parser source code files, you need to compile them along with your main program. In your case, the main program is named myProgram.exe.

Use the following commands in the command prompt to compile the files:

**$ gcc lex.yy.c parser.tab.c -o myProgram**



```
D:\FACULTATE\Materiale facultate 2023-2024\LFTC\Labs\Lab9>gcc lex.yy.c parser.tab.c -o myProgram
```

This command compiles the generated scanner and parser files along with your main program, producing an executable file named myProgram.exe.

## 5. Running the Program:

To run your program with an input file (e.g., p1.in), use the following command:

```
$ myProgram.exe p1.in
```

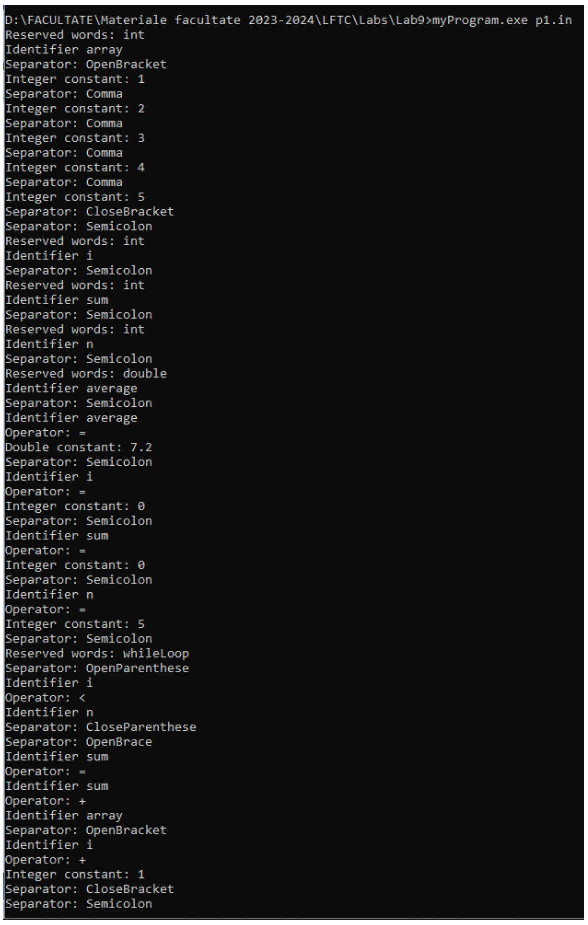
or

```
$ myProgram.exe p2.in
```

or

```
$ myProgram.exe p3.in
```

This assumes that your input file is named p1.in. Adjust the filename accordingly.



```
D:\FACULTATE\Materiale facultate 2023-2024\LFTC\Labs\Lab9>myProgram.exe p1.in
Reserved words: int
Identifier array
Separator: OpenBracket
Integer constant: 1
Separator: Comma
Integer constant: 2
Separator: Comma
Integer constant: 3
Separator: Comma
Integer constant: 4
Separator: Comma
Integer constant: 5
Separator: CloseBracket
Separator: Semicolon
Reserved words: int
Identifier i
Separator: Semicolon
Reserved words: int
Identifier sum
Separator: Semicolon
Reserved words: int
Identifier n
Separator: Semicolon
Reserved words: double
Identifier average
Separator: Semicolon
Identifier average
Operator: =
Double constant: 7.2
Separator: Semicolon
Identifier i
Operator: =
Integer constant: 0
Separator: Semicolon
Identifier sum
Operator: =
Integer constant: 0
Separator: Semicolon
Identifier n
Operator: =
Integer constant: 5
Separator: Semicolon
Reserved words: whileLoop
Separator: OpenParenthese
Identifier i
Operator: <
Identifier n
Separator: CloseParenthese
Separator: OpenBrace
Identifier sum
Operator: =
Identifier sum
Operator: +
Identifier array
Separator: OpenBracket
Identifier i
Operator: +
Integer constant: 1
Separator: CloseBracket
Separator: Semicolon
```

## 6. Conclusion:

Flex and Bison are powerful tools for building compilers and interpreters. They streamline the process of lexical analysis and parsing, making it easier to develop robust language processors. The commands provided in this documentation illustrate the typical workflow for using Flex and Bison in conjunction.