# KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY

## Department of Computer Science and Technology

**Title:** A simple compiler using flex and Bison.

**Course Title:** Compiler Design Laboratory

**Submitted by:**

**Shrabanti Debnath Urmi**

Roll: **1907094**

3rd Year 2nd Semester

Department of Computer Science and Engineering

Khulna University of Engineering and Technology, Khulna

**Submission date:** 22 November, 2023

## Objectives:

After completing this project, we will be able to know

- About Flex and Bison.
- About token and how to declare rules against token.
- How to declare CFG (context free grammar) for different grammar like if else pattern, loop and so on.
- About different patterns and how they work.
- How to create different and new semantic and synthetic rules for the compiler.
- About shift and reduce policy of a compiler.
- About top down and bottom up parser and how they work.

## Introduction:

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. Typically, a programmer writes language statements in a language such as Pascal or C one line at a time using an *editor*. The file that is created contains what are called the *source statements*. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements.

***Flex:*** Flex (fast lexical analyzer generator) is a free and open-source software alternative to lex. It is a computer program that generates lexical analyzers (also known as "scanners" or "lexers"). An input file describing the lexical analyzer to be generated named *lex.l* is written in lex language. The lex compiler transforms *lex.l* to *C* program, in a file that is always named *lex.yy.c*. The C compiler compiles the *lex.yy.c* file into an executable file called a.out. The output file a.out takes a stream of input characters and produces a stream of tokens.

```
/* definitions */
 ....
```

```
%%
/* rules */
....
%%
/* auxiliary routines */
....
```

***Bison:*** GNU Bison, commonly known as Bison, is a parser generator that is part of the GNU Project. Bison reads a specification of a context-free language, warns about any parsing ambiguities, and generates a parser (either in C, C++, or Java) that reads sequences of tokens and decides whether the sequence conforms to the syntax specified by the grammar. **Bison** command is a replacement for the **yacc**. It is a parser generator similar to *yacc*. Input files should follow the yacc convention of ending in *.y* format.

```
/* definitions */
 ....

%%
/* rules */
....
%%

/* auxiliary routines */
....
```

## Instruction in cmd when we use flex and bison together:

1. bison -d 1907094.y
2. flex 1907094.l
3. gcc lex.yy.c 1907094.tab.c -o ex
4. ex

## Features of this mini-compiler:

1. **Token Recognition:**
   - Recognizes keywords (int, double, char, etc.), operators (+, -, *, /, etc.), literals (numbers, strings), and identifiers.
   - Handles comments, both single-line (!- ...) and multi-line ($ ... $) comments.
   - Supports a preprocessor-like directive (@@include) for including headers.
2. **Token Definitions:**
   - Defines token patterns for various elements like identifiers, numbers, strings, and operators.

### Syntax Analysis (Parser):

1. **Variable Declarations:**
   - Supports variable declarations with optional initialization.
   - Ensures proper handling of variable redeclarations.
2. **Arithmetic Expressions:**
   - Parses arithmetic expressions with correct operator precedence and associativity.
   - Handles unary minus (-) as a separate precedence level.
3. **Control Structures:**
   - Implements if-else statements with associated blocks of code.
   - Supports for loops, allowing iteration over a range of values.
   - Implements switch-case statements for handling multiple branches of code execution.
4. **Built-in Functions:**
   - Supports various built-in functions, including mathematical functions (sin, cos, tan, etc.), string manipulation (reverse, sort), and others (gcd, lcm, power, etc.).
5. **User-Defined Functions:**
   - Allows the declaration of user-defined functions with parameters and a block of statements.
6. **Semantic Actions:**
   - Performs semantic actions to check for variable declarations, redeclarations, and initializes variables.
   - Checks for the correct usage of variables before using them.
   - Handles errors related to undeclared variables, redeclarations, division by zero, etc.
7. **Output Generation:**
   - Generates output code during parsing, especially in the case of print statements and function calls.
   - Prints the result of expressions, variables, and function calls during runtime.
8. **File Handling:**
   - Opens input and output files (input.txt and output.txt) for reading and writing, respectively.
9. **Error Handling:**
   - Provides error messages with line numbers for easier debugging.

- Reports errors related to undeclared variables, redeclarations, division by zero, etc.

10. **Preprocessor Directives:**
    - Supports a preprocessor directive (@@include) for including headers.
11. **Mathematical Functions:**
    - Implements various mathematical functions such as sin, cos, tan, gcd, lcm, etc.
12. **String Manipulation:**
    - Supports string operations like reverse and sort.
13. **Comments:**
    - Allows both single-line (!- ...) and multi-line ($ ... $) comments.
14. **Looping Structures:**
    - Implements the for loop, supporting iteration and executing a block of statements.
15. **Conditional Statements:**
    - Supports if-else statements with associated blocks of code.
16. **Switch-Case Statements:**
    - Implements switch-case statements for handling multiple branches of code execution based on a variable's value.

## Token:

A **token** is a pair consisting of a **token** name and an optional attribute value. The **token** name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or sequence of input characters denoting an identifier. The **token** names are the input symbols that the parser processes.

## Tokens used in this mini-project:

Bellow in the table it is shown those tokens that is used in this mini-project and their real time meaning-

| Serial no. | Token | Input string | Realtime meaning of Token |
|---|---|---|---|
| 1 | NUMBER | [0-9]+ | Any integer number. |
| 2 | ID | [a-zA-Z0-9$_@]+ | Identifier, including letters, digits, $, _, and @. |
| 3 | SEMI_COLON | ; | Semicolon indicating the end of a statement. |
| 4 | COMMA | , | Comma used to separate elements. |
| 5 | PRINTVAR | output>> | Print variable statement. |
| 6 | PRINTSTR | outputs>> | Print string statement. |
| 7 | PRINTLN | outputn>> | Print new line statement. |
| 8 | PRINTFUNC | outputf>> | Print function statement. |
| 9 | FUNCTION | func-> | Function declaration. |
| 10 | INT | int | Declaration of an integer variable. |
| 11 | DOUBLE | double | Declaration of a double variable. |
| 12 | CHAR | char | Declaration of a character variable. |
| 13 | PB | ( | Left parenthesis. |
| 14 | PE | ) | Right parenthesis. |
| 15 | BB | { | Left curly brace. |
| 16 | BE | } | Right curly brace. |
| 17 | ASSIGN | = | Assignment operator. |
| 18 | PLUS | + | Addition operator. |
| 19 | MINUS | - | Subtraction operator. |
| 20 | MULTIPLY | * | Multiplication operator. |
| 21 | DIVIDE | / | Division operator. |

| | | | |
|---|---|---|---|
| 22 | MOD | % | Modulus operator. |
| 23 | LESSTHEN | < | Less than operator. |
| 24 | GREATERTHEN | > | Greater than operator. |
| 25 | LESSEQUAL | <= | Less than or equal to operator. |
| 26 | GREATEREQUAL | >= | Greater than or equal to operator. |
| 27 | EQUAL | == | Equal to operator. |
| 28 | NOTEQUAL | != | Not equal to operator. |
| 29 | MAXNUMBER | MAX | Maximum value operator. |
| 30 | MINNUMBER | MIN | Minimum value operator. |
| 31 | COMPARE | CMP | Compare operator. |
| 32 | COMPAREREVERSE | CMPR | Reverse compare operator. |
| 33 | REVERSE | REV | Reverse a string. |
| 34 | SORT | SORT | Sort a string. |
| 35 | FACT | FACT | Factorial function. |
| 36 | SINFUNC | sin | Sine function. |
| 37 | COSFUNC | cos | Cosine function. |
| 38 | TANFUNC | tan | Tangent function. |
| 39 | LOG10FUNC | log10 | Logarithm base 10 function. |
| 40 | LOGFUNC | log | Natural logarithm function. |

| | | | |
|---|---|---|---|
| 41 | GCDFUNC | gcd | Greatest common divisor function. |
| 42 | LCMFUNC | lcm | Least common multiple function. |
| 43 | POWERFUNC | pow | Power function. |
| 44 | ELSEIF | elseif | Else if condition. |
| 45 | IF | if | If condition. |
| 46 | ELSE | else | Else condition. |
| 47 | FOR | for | For loop. |
| 48 | INC | IncrementBy | Increment by statement. |
| 49 | TO | ... | Range operator in for loop. |
| 50 | SWITCH | switch | Switch statement. |
| 51 | DEFAULT | default | Default case in switch statement. |
| 52 | COLON | : | Colon sign. |
| 53 | STR | "..." | String literal. |
| 43 | POWERFUNC | pow | Power function. |
| 44 | ELSEIF | elseif | Else if condition. |
| 45 | IF | if | If condition. |
| 46 | ELSE | else | Else condition. |
| 47 | FOR | for | For loop. |
| 48 | INC | IncrementBy | Increment by statement. |
| 49 | TO | ... | Range operator in for loop. |
| 50 | SWITCH | switch | Switch statement. |

| | | | |
|---|---|---|---|
| 51 | DEFAULT | default | Default case in switch statement. |
| 52 | COLON | : | Colon sign. |
| 53 | STR | "..." | String literal. |

Table 1. Realtime meaning of tokens that are used in project

**Input:**

```
@@include<"main program">

@@include<"header">

int func->Init ( int Var_a, int Var_b ){}

int main ( ){

        int x;

          int a = 6 + 5 ;

        int ab = a;

        int b = -5,c = b;

        output>> ( a );

        outputn>> ( );

        output>>(b);

        outputn>>();

        output>> ( ab );

        outputn>> ( );
outputn>> ( );
  outputs>> ( "--------------" );
outputn>> ( );
 !- Single line comment
  $.........
....
........ multiline comment
....
..........$
!- new input
!- gcd and lcm
b = 60 gcd 25 ;
output>> ( b ) ;
outputn>>();
c = 60 lcm 25 ;
output>> ( c ) ;
outputn>>();
outputn>>();

!- manual function

outputf>> sin(90) ;
outputf>> cos(45) ;
outputf>> tan(45) ;

outputf>> log10(2) ;

outputf>> log(2) ;

outputn>>();

!- Mod function

a = 5 % 3 ;

output>> (a) ;

outputn>>();


!-power function

int p = 5 pow 3 ;

output>> ( p ) ;

outputn>>();


!-Factorial function

outputf>> FACT(5);



!-max and min of two numbers

outputf>> MAX ( 100 CMP 21 ) ;

outputf>> MIN ( 400 CMPR 23 ) ;

outputn>> ();
```

```
        !-reverse and sort a string

        outputf>> REV ( "Khulna" ) ;

        outputn>> ( );

        outputf>> SORT ( "zwabgdertef" ) ;

        outputn>> ( );


    !- Addition

        a = 25 + 53 ;

        outputs>> ( "Addition of 25 + 53: " );

        output>> ( a );

        outputn>> ( );


    !- Substraction

        int sub = 10 - 3 ;

        outputs>> ( "Substraction of 10 - 3: " );

        output>> ( sub ) ;

        outputn>> ( );


        !-Multiplication

        int mul = 5 * 7 ;

        outputs>> ( "Multiplication of 5 * 7: " ) ;

        output>> ( mul ) ;

        outputn>> ( ) ;
```

```
!-Division

int div = 300 / 15 ;

outputs>> ( "division of 300 / 15: " ) ;

output>> ( div ) ;

outputn>> ( ) ;


!-end new input

if ( 5 > 4 )

{

outputs>> ( "if Executed" ) ;

}

elseif ( 5 < 4 )

{

outputs>> ( "else if executed" ) ;

}

else{

outputs>>("else executed");

}

outputn>>();


int start = 1 + 0;
```

```
        int end = 6;


        for ( start ... end : 2 )

        {

        outputs>>( "hi " );

        }

        outputn>>();


        int stw = 2;


        switch ( stw )

        {1:     {

                }

        2:       {

                outputs>>("switch variable 7");

                  }

           default:  {}

        }

}
```

**Output:**

```
User DEfine Function Declared

11

-5

11

--------------

5

300

1.000000

0.707105

1.000004

0.301030

0.693147

2

125

120

100

23

anluhK

abdeefgrtwz

Addition of 25 + 53: 78

Substraction of 10 - 3: 7

Multiplication of 5 * 7: 35
```

```
division of 300 / 15: 20

if Executed

hi hi

switch variable 7

****Compilation Successful****
```

**Discussion:**

My mini-compiler demonstrates a comprehensive set of features, allowing users to write programs in a language that supports fundamental programming constructs. The inclusion of @@include preprocessor directives is a valuable feature, enhancing code organization and promoting reusability by enabling the incorporation of external files.The support for user-defined functions is a key aspect of your compiler, even though the function bodies are currently empty. This feature lays the groundwork for modular programming and code abstraction, providing a structure for developers to create and use their own functions.The compiler handles output effectively, covering various scenarios such as printing variable values, strings, and the results of mathematical functions. This flexibility makes the language versatile and capable of performing a wide range of operations.The mathematical functions, including sin, cos, tan, log, gcd, lcm, power, and factorial, enhance the language's utility by providing users with built-in mathematical tools. Additionally, the inclusion of string manipulation functions like reverse and sort expands the language's capabilities beyond basic arithmetic.Control flow structures, such as if-else statements, for loops, and switch-case structures, are correctly implemented. This ensures that developers can create structured and complex programs using your mini-compiler.The error handling mechanisms, such as checking for undeclared variables, redeclarations, and division by zero, contribute to the compiler's reliability. Clear error messages help developers identify and rectify issues in their code effectively.

## Conclusion:

In conclusion , my mini-compiler is a commendable project that serves as a foundation for understanding language design and implementation. While some features are currently placeholders, they provide a roadmap for future expansion. To enhance the language further, I may consider implementing the functionality for user-defined functions, refining error messages for better diagnostics, and expanding the standard library with more functions. Overall, my mini-compiler offers a solid starting point, and with continued development, it has the potential to become a more feature-rich and versatile programming language.

## References:

- https://whatis.techtarget.com/definition/compiler
- Principles of Compiler Design By Alfred V.Aho & J.D Ullman