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:

MD Mizanur Rahman

Roll: 1807063

3rd Year 2nd Semester

Department of Computer Science and Engineering

Khulna University of Engineering and Technology, Khulna

Submitted To:

Dola DAS

Assistant Professor

Department of Computer Science and Engineering

Khulna University of Engineering and Technology, Khulna

Dippanita Biswas

Lecturer

Department of Computer Science and Engineering

Khulna University of Engineering and Technology, Khulna

Objective:

After doing 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 hi.y
2. flex hi.l
3. gcc lex.yy.c hi.tab.c -o ex
4. ex

Features of this mini-compiler:

- Import section or header declaration section.
- User defined function section.
- Main part or body of the program.
- Declaration of different id or variable and assignment operation.
- If else condition.
- Arithmetic and logical operation.
- Loop (for loop and while loop).
- Switch-case condition.
- Print and show different values.
- Single line and multiple line command.
- Built-in sine function.

- Built-in cosine function.
- Built-in tan function.
- Built-in ln function.
- Built-in log10 function.
- Built-in factorial function.
- Built-in odd even function.

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 realtine meaning-

Serial no.	**Token**	**Input string**	**Realtime meaning of Token**
1.	NUM	(-|+)?[0-9]+	Any integer number either positive or negative.
2.	VAR	[a-zA_Z0-9]+	Any string using upper case alphabets, lower case alphabets and numbers.
3.	SM	”	Indicate ends of a line.
4.	INT	INT	Declaration of integer variable.
5.	FLOAT	FLOAT	Declaration of floating point variable.
6.	CHAR	CHAR	Declaration of character variable.

7.	IF	IF	If condition start
8.	ELSE	ELSE	Else condition start
9	ELIF	ELIF	Else if condition start
10	FOR	FOR	For loop line c programming
11	LOOP	LOOP	While loop in c programming
12	COL	:	Colon sign like default
13	SWITCH	SWITCH	Switch case like c programming
14	CASE	CASE	Case like c programming
15	DEFAULT	DEFAULT	Alternative option
16	SHOW	SHOW	For printing any variable
17	ASSIGN	=	Assignment operator
18	INC	INC	Used for increment any value by one
19	DEC	DEC	Used for decrement any value by one
20	LT	LT	Less than sign
21	GT	GT	Greater than sign
22	EQ	EQ	Check equal or not
23	QE	QE	Greater than or equal sign
24	MAIN	MAIN	Start of a program
25	FACT	FACT	Calculating factorial of a number
26	DEF	DEF	Function definition

27	IMPORT	#include	Needs for including header function
28	ODDEVEN	OddEven	For calculation Odd Even function
29	PLUS	ADD	Addition operation
30	MINUS	SUB	Subtraction operation
31	MULT	MUL	Multiplication operation
32.	DIV	DIV	Division operation
33	MOD	MOD	Module operation
34	AND	AND	And operation
35	OR	OR	Or operation
36	XOR	XOR	Xor operation
37	LP	((	First bracket opening
38	RP	))	First bracket closing
39	LB	{{	second bracket opening
40	RP	}}	second bracket closing
41	POW	POW	Power operation
42	MIN	MIN	Minimum operation
43	MAX	MAX	Maximum operation
44	CM	;	Semicolon

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

Structure of the project:

CFGs used in this project:

program: MAIN LP RP LB cstatement RB { ACTION }

cstatement: /* empty */ | cstatement statement | cdeclaration

cdeclaration: TYPE ID1 SM

TYPE : INT | FLOAT | CHAR ;

ID1 : ID1 CM VAR |VAR

statement: SM
| SWITCH LP switch_expr RP LB BASE RB
| SHOW LP VAR RP SM | expression SM | VAR ASSIGN expression SM
| IF LP expression RP LB expression SM RB %prec
| IF LP expression RP LB expression SM RB ELSE LB expression SM RB
| IF LP expression RP LB IF LP expression RP LB expression SM RB ELSE LB expression SM RB expression SM RB ELSE LB expression SM RB %prec IFX
| IF LP expression RP LB expression SM RB ELIF LP expression RP LB expression SM RB ELSE LB expression SM RB
| FOR LP NUM COL NUM RP LB expression RB
| FOR LP NUM COL NUM COL NUM RP LB expression RB
| WHILE LP NUM GT NUM RP LB expression RB
| DEF func

func : COL TYPE LP RP LB statement RB
BASE : Bas | Bas Dflt
Bas   : /*NULL*/ | Bas Cs
Cs   : CASE NUM COL expression SM
Dflt   : DEFAULT COL expression SM

```
expression: NUM | VAR | expression PLUS expression
            | expression MINUS expression | expression MULT expression
            | expression DIV expression| expression AND expression
            | expression OR expression | expression XOR expression
            | expression MOD expression | POW LP expression CM  expression RP
            | MAX LP expression CM  expression RP
            | MIN LP expression CM  expression RP
            | expression FACT | expression LT expression | expression GT expression
           | expression GE expression | expression LE expression
            | expression EQ expression | LP expression RP | expression INC
            | expression DEC  | expression NOT | switch_expr: NUM | VAR
            | switch_expr PLUS switch_expr| switch_expr MINUS switch_expr
            | switch_expr MULT switch_expr | switch_expr DIV switch_expr
            | switch_expr POW switch_expr
            | switch_expr FACT
            | switch_expr LT switch_expr
            | switch_expr GT switch_expr | LP switch_expr RP
            | switch_expr INC      | switch_expr DEC        | switch_expr NOT
```

Discussion:

The input code is parsed using a bottom-up parser in this compiler. Because it is only built with flex and bison, this compiler is unable to provide original functionality such as if-else, loop, and switch case features. However, when creating code in this compiler-specific style, header declaration is not required but if we need we can use header file. The float variable always returns a value in the double data type, which is a compiler requirement. Any variable's string value is not stored by this compiler. With certain modifications, the code format supported by this compiler is similar to that of the C language. This compiler is error-free while working with the stated CFG format.

Conclusion:

Every programming language has required the use of a compiler. Designing a new language without a solid understanding of how a compiler works may be a challenging endeavor. Several issues were encountered during the design phase of this compiler, such as loop, if-else, switch case functions not working as they should owing to bison limitations, character and string variable values not being stored properly, and so on. In the end, some of these issues were resolved, and given the constraints, this compiler performs admirably.

References:

- https://www.youtube.com/watch?v=fFRxWtRibC8
- Principles of Compiler Design By Alfred V.Aho & J.D Ullman