KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY

Department of Computer Science and Technology

Report on CSE-3212

Course Title: Compiler Design Laboratory

Submitted to:

Dola Das

Assistant Professor

Department of Computer Science and Engineering

Khulna University of Engineering & Technology, Khulna

Dipannita Biswas

Lecturer

Department of Computer Science and Engineering

Khulna University of Engineering & Technology, Khulna

Submitted by:

Nabil Faiyaz Sadi

Roll: **1807073**

3rd Year 2nd Semester

Department of Computer Science and Engineering

Khulna University of Engineering & Technology, Khulna

Objective:

After completing this project, we will learn about

- 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 */
....
```

Commands to write in cmd:

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

Features of this compiler:

- Import section or header declaration section.
- Declaration of integer, double and string type variables and assignment operation.
- If else condition.
- Arithmetic and logical operation.

- Loop (for loop and while loop).
- User defined function section.
- Print and show different values.
- Single line and multiple line comment.
- Built-in power function.
- Built-in sine function.
- Built-in cosine function.
- Built-in tan function.
- Built-in ln function.
- Built-in log10 function.
- Built-in log2 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 project:

Serial no.	**Token**	**Input string**	**Realtime meaning of Token**
1	IDENTIFIER	[a-zA-Z][_a-zA-Z0-9]*	Declare variable name.
2	INTEGER	[0-9]*	Any integer value.
3	DOUBLE	[0-9]+[.][0-9]+	Any floating point value.
5	EOL	;	Indicates end of a line.

6	NUMBER_TYPE	number	Declaration of integer variable.
7	DECIMAL_TYPE	decimal	Declaration of floating-point variable.
8	STRING_TYPE	string	Declaration of string variable.
9	SCOMMENT	/>	Single line comment.
10	MCOMMENT	/* */	Multi line comment.
11	IF	if	If condition start.
12	ELIF	elif	Else if condition start.
13	ELSE	else	Else condition start.
14	FOR	for	For loop.
15	WHILE	while	While loop.
16	INC	++	Used for incrementing any variable by one.
17	DEC	--	Used for decrementing any variable by one.
18	LT	<	Less than sign.
19	GT	>	Greater than sign.
20	EQ	===	Check equal or not.
21	NEQL	!!!	Check not equal.
22	GEQL	>=	Greater than or equal sign.

23	LEQL	<=	Less than or equal sign.
24	DEF	def	Function definition.
25	CALL	call	Function call.
26	INPUT	input	Take input.
27	PRINT	print	Print variables.
28	HEADER	file.h	Header function.
29	POW	**	Power operation.
30	SIN	sin	Sine function.
31	COS	cos	Cosine function.
32	TAN	tan	Tangent function.
33	LN	ln	ln function.
34	LOG10	log10	Log function.
35	LOG2	log2	Log2 function.
36	SQRT	sqrt	Square Root function.
37	AND	&&	Logical AND operation.
38	OR	||	Logical OR operation.
39	XOR	^^	Logical XOR operation.
40	NOT	~~	Logical NOT operation.
41	‘+’	+	Addition operation.

42	‘-’	-	Subtraction operation.
43	‘*’	*	Multiplication operation.
44	‘/’	/	Division operation.
45	‘%’	%	Modulo operation.
46	‘(’	(	First bracket opening.
47	‘)’	)	First bracket closing.
48	‘{’	{	second bracket opening.
49	‘}’	}	second bracket closing.
50	‘[’	[	Third bracket opening.
51	‘]’	]	Third bracket closing.
52	‘,’	,	Comma.

Table 1. Used Tokens and their meanings

Grammars Used in this Project:

```
program:
    HEADER statements {}
    |statements {}
;
statements:
    {}
    |statements statement
;
```

```
statement:
    EOL
    |statement EOL statement
    |SCOMMENT
    |MCOMMENT
    |input EOL
    |print EOL
    |declarations EOL
    |assignments EOL
    |if_blocks
    |for_loop
    |while_loop
    |function_declare
    |function_call EOL
;
print:
    PRINT '(' output_variable ')' {}
;
output_variable:
    output_variable ',' VARIABLE {}
    |VARIABLE {}
;

input:
    INPUT '(' input_variable ')' {}
;
input_variable:
    input_variable ',' VARIABLE {}
    |VARIABLE {}
;
function_declare:
    DEF function_name '(' function_variable ')' ARROW return_types '{'
    statement '}' {}
;
return_types:
    NUMBER_TYPE
    |DECIMAL_TYPE
    |STRING_TYPE
;
```

```
function_name:
    VARIABLE {}
;
function_variable:
    |function_variable ',' single_variable
    | single_variable
;
single_variable:
    NUMBER_TYPE VARIABLE {}
    |DECIMAL_TYPE VARIABLE {}
;
function_call:
    CALL user_function_name '(' parameters ')' {}
;
user_function_name:
    VARIABLE {}
;
parameters:
    parameters ',' single_parameter
    |single_parameter
;
single_parameter:
    VARIABLE {}
;
for_loop:
    FOR '(' VARIABLE IN '[' expr ',' expr ',' expr ']' ')' '{'
    statement '}' {}
;
while_loop:
    WHILE '(' while_conditions ')' '{' statement '}' {}
;
while_conditions:
    VARIABLE INC LT expr {}
    |VARIABLE INC LEQL expr {}
    |VARIABLE INC NEQL expr {}
    |VARIABLE DEC GT expr {}
    |VARIABLE DEC GEQL expr {}
    |VARIABLE DEC NEQL expr {}
;
```

```
if_blocks:
    IF if_block else_block {}
;
if_block:
    '(' expr ')' '{' statement '}' {}
;
else_block:
    |elif_block
    |elif_block single_else_block
    |single_else_block
;
elif_block:
    elif_block single_elif_block
    | single_elif_block
;
single_elif_block:
    ELIF '(' expr ')' '{' statement '}' {}
;
single_else_block:
    ELSE '{' statement '}' {}
;
declarations:
    NUMBER_TYPE num_vars
    |DECIMAL_TYPE dec_vars
    |STRING_TYPE str_vars
;
str_vars:
    str_vars ',' str_var
    |str_var
;
str_var:
    VARIABLE '=' STRING_VALUE {}
    |VARIABLE {}
;
dec_vars:
    dec_vars ',' dec_var
    |dec_var
;
```

```
dec_var:
    VARIABLE '=' expr {}
    |VARIABLE {}
;
num_vars:
    num_vars ',' num_var
    |num_var
;
num_var:
    VARIABLE '=' expr {}
    |VARIABLE {}
assignments:
    assignments ',' assignment
    |assignment
;

assignment:
    VARIABLE '=' expr {}
;
expr:
    NUMBER_VALUE {}
    |DECIMAL_VALUE {}
    |VARIABLE {}
    |'+' expr {}
    |'-' expr {}
    |INC expr {}
    |DEC expr {}
    |expr '+' expr {}
    |expr '-' expr {}
    |expr '*' expr {}
    |expr '/' expr {}
    |expr '%' expr {}
    |expr POW expr {}
    |expr EQL expr {}
    |expr NEQL expr {}
    |expr LT expr {}
    |expr GT expr {}
    |expr LEQL expr {}
    |expr GEQL expr {}
    |expr AND expr {}
    |expr OR expr {}
    |expr XOR expr {}
    |NOT expr {}
    |VARIABLE INC {}
    |VARIABLE DEC {}
    |'(' expr ')' {}
    |SIN '(' expr ')' {}
    |COS '(' expr ')' {}
    |TAN '(' expr ')' {}
    |LOG10 '(' expr ')' {}
    |LOG2 '(' expr ')' {}
    |LN '(' expr ')' {}
    |SQRT '(' expr ')' {}
;
```

Discussion:

The input code is parsed using a bottom-up parser. As it is built with flex and bison, this compiler is unable to provide original functionality for if-else, loop, and switch case features. Some basic functionalities of a language were implemented. There are three basic data types that can be parsed using this compiler and they are Integer, Float and String. We can perform as basic arithmetic and logical operations. Some built-in functions such as sine, cosine, tangent, exponent, log etc. were also implemented. This compiler is error-free while working with the stated CFG format.

Conclusion:

Designing a new language without a solid understanding of how a compiler works is a challenging task. In this project, a simple compiler was implemented using flex and bison. Several issues were encountered during the design phase of this compiler such as loop, if-else, functions etc. not working as they should owing to bison limitations. In the end, some of these issues were resolved. The language and the compiler could be further extended by adding additional data types, data structures and writing proper grammar rules to parse them.

References:

- Principles of Compiler Design By Alfred V.Aho & J.D Ullman