KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY

Department of Computer Science and Engineering

Report on CSE3212 PROJECT

***Course Title*:** Compiler Design Laboratory

***Topic*:** Simple Compiler using Flex and Bison

***Date of Submission*:** June 15, 2021

Submitted by:

Md Zahim Hassan

Roll: **1707007**

3rd Year 2nd Semester

Department of Computer Science and Engineering

Khulna University of Engineering & Technology (KUET), Khulna-9203

TABLE OF CONTENTS

Introduction:

In computing, a compiler is a computer program that translates computer code written in one programming language into another language. The name “Compiler” is used for programs that translate source code from a high level programming language to a lower level language to create executable program. A compiler is likely to perform many or all of the following operations: preprocessing, lexical analysis, parsing, semantic analysis, intermediate code generation, code optimization, code generation. Compilers implement these operations in phases that promote efficient design and correct transformation of source code to target code.

Flex:

For dividing the input into meaningful units such as variable, keywords, constants, operators, punctuations, etc. we need a tool called “Flex”. This tokenization step is called “Lexical Analysis”. Flex is a tool for generating scanners. Flex source is a table of regular expressions and corresponding program fragments. It generates lex.yy.c file which defines a routine yylex(). The required file for Flex is saved with “.l” extension. Structure of a file that is compatible with Flex is as follows:

```
{Definitions}

%%

{Rules}

%%

{User subroutine}
```

Bison:

Parsing is an essential stage for designing a compiler. In this stage, Context Free Grammars are used to parse input commands of a program using a parse tree. Bison is a powerful and free version of “yacc” which has the full form “yet another compiler compiler”, a UNIX parser. It is a general purpose parser generator that converts a grammar description for an LALR(1) context free grammar into a C program to parse the grammar. Bison is compatible with yacc. All properly written yacc grammars are ought to work with bison with no change. It interfaces with scanners generated by Flex. Bison input file is saved with a “.y” extension. Bison generates two files compiling a input file those are: inputfile_name.tab.h and inputfile_name.tab.c. File with “.h” extension is used in Flex input file to connect both file and the later one is used as a parser scanner. Input file structure of a bison file is as follows:

```
%}
C declarations
%}
Bison Declarations
%%
Grammar Rules Declarations in BNF form(CFG)
%%
Additional C codes
```

Procedure:

1. The code is divided into a flex file (.l) and a bison file (.y).
2. Using the bison file code is drawn from a text file (.txt).
3. After that input expressions are matched with the rules of flex file (.l) and upon satisfaction of the matching step, the CFG of the bison file (.y) is checked for a match.
4. It is a bottom up parser and the parser construct the parse tree. At first, matches the leaves node with the rules and if the CFG matches then it gradually goes to the root.

Features of this Compiler:

- Header declaration
- Body declarations
- Variable declarations (integer, character, float)
- Variable value assignment
- Arithmetic operations (+, -, *, /)
- Logical Operations (==, !=, <, <=, >, >=)
- For loop
- Switch-case
- If-else if- else
- Built-in Power Function
- Built-in Prime checking Function
- Built-in Factorial Function
- Built-in Min Function
- Built-in Max Function
- Single line comment
- Print Function

- Read Function

Token:

Tokens are second lowest meaningful instance in a language (the lowest being character). Tokens are identified by its *lexeme*, a (set of) character sequence that fulfills the token property. The parser has to recognize these as tokens: identifiers, keywords, literals, operators, punctuators, and other separators.

Tokens used in this Compiler:

The following table represents the tokens and meaning of these tokens used in this compiler design process.

Serial No.	Token	Input String	Definition of Token
1	MAIN	start()	Defines the start of the function similar as main() in C
2	INT	int	Specify the variable of integer type
3	CHAR	char	Specify the variable of character type
4	FLOAT	float	Specify the variable of float type
5	POWER	power	For calling power function similar to pow() in C
6	FACTO	facto	For calling factorial function, returns the factorial value of a integer
7	PRIME	checkprime	Checks if a number is prime or not by returning 0 for prime and 1 otherwie.
8	READ	read	Used for scanning user input value of a variable
9	PRINT	print	Prints the variable value
10	SWITCH	switch	Similar to switch() in C
11	CASE	state	Similar to case in C
12	DEFAULT	complementary	Similar to default in C
13	IF	if	Similar to if() in C
14	ELIF	elif	Similar to else if() in C
15	ELSE	otherwise	Similar to else in C
16	FROM	from	Denotes start value of a variable
17	TO	to	Denotes end value of a variable to be checked
18	INC	inc	Defines increment of variable value
19	DEC	dec	Defines decrement of variable value
20	MAX	max	For the max function call
21	MIN	min	For the min function call
22	ID	[_a-zA-Z][_a-zA-Z0-9]*	Any string combination of alphabet number and underscore.

23	NUM	[-]?[0-9][0-9]*[.]?[0-9]*	Any number with or without decimal point
24	PLUS	+	Denotes plus operation
25	MINUS	-	Denotes minus operation
26	MUL	*	Denotes multiplication operation
27	DIV	/	Denotes division operation
28	EQUAL	==	Denotes equality checking operation
29	NOTEQUAL	!=	Denotes not equality checking operation
31	GT	>	Represent greater than logical operation
32	LT	<	Represent less than logical operation
33	GOE	>=	Represent greater than or equal logical operation
34	LOE	<=	Represent less than or equal logical operation

Table 1.1: Table of used tokens and meanings of tokens

Context Free Grammar (CFG):

Context Free Grammars (CFGs) are used to describe context-free languages. A context-free grammar is a set of recursive rules used to generate patterns of strings. A context-free grammar can describe all regular languages and more, but they cannot describe all possible languages. The production rule of a CFG is of the form,

A $\rightarrow$ b where A is a non-terminal and b is a terminal.

CFGs used in this Compiler:

program: MAIN '{'code'}'

;

code: declaration code

|assignment code

|condition code

|for_code code

|switch_code

|print_code code

|read_code code

|power_code code

|factorial_code code

```
            |prime_code code
            |min_code code
            |max_code code
            |
            ;
power_code: POWER ‘(‘NUM ‘,’NUM ‘)’’;’
            ;
factorial_code: FACTO ‘(‘NUM’)’’;’
            ;
prime_code: PRIME ‘(‘NUM’)’’;’
            ;
max_code: MAX ‘(‘ID ‘,’ ID ‘)’’;’
            ;
min_code: MIN ‘(‘ID ‘,’ ID ‘)’’;’
            ;
print_code: PRINT ‘(‘ID’)’’;’
            ;
read_code: READ ‘(‘ID’)’’;’
            ;
switch_code: SWITCH '(' ID ')' '{' case_code '}'
            ;
case_code: casenum_code default_code
            ;
casenum_code: CASE NUM '{' code '}' casenum_code
            ;
default_code: DEFAULT '{' code '}'
            ;
for_code: FROM ID TO NUM INC NUM '{' code '}'
```

```
            | FROM ID TO NUM DEC NUM '{' code '}'
            ;
condition: IF'(' bool_expression ')''{'code'}' else_if elsee
            ;
else_if: ELIF '(' bool_expression ')''{' code '}' else_if
            :
elsee: ELSE '{' code '}'
            ;
bool_expression: expression EQUAL expression
            | expression EQUAL expression
            | expression NOTEQUAL expression
            | expression GT expression
            | expression GOE expression
            | expression LT expression
            | expression LOE expression
            ;
declaration: TYPE ID1 ';'
            ;
TYPE: INT
            |FLOAT
            |CHAR
            ;
ID1: ID1 ',' ID
            |ID
            ;
assignment: ID '=' expression ';'
            ;
expression: e
```

```
        ;
e: e PLUS f
        | e MINUS f
        |f
        ;
f: f MUL t
        | f MUL t
        |t
        ;
t: '(' e ')'
        |ID
        |NUM
        ;
```

Terminal commands to run the program:

1. bison –d pf.y
2. flex pf.l
3. gcc lex.yy.c pf.tab.c –o outputfile
4. outputfile

Discussion:

This compiler uses a bottom up parser to parse the input code. This compiler is unable to provide original functionality of if else, loop, switch case features as it is only build using flex and bison. However header declaration is not mandatory while writing a code in this compiler specific format. The float variable always returns value in double data type which is a specification of this compiler. This compiler doesn’t stores string value of any variable. The code format that is supported by this compiler is close to that of C language with some modification. This compiler works perfectly with the declared CFG format without any error.

Conclusion:

Compiler has been an essential part of every programming language. Without a sound knowledge about how a compiler works, designing a new language can be very difficult task. Several difficulties are faced during design period of this compiler such as loop, if else, switch case functions not working as it should be due to limitation of bison, character and string variable value isn't storing properly, etc. At the end, some of this problems have been fixed and considering the limitations this compiler works just fine.

References:

1. https://github.com/Shisimanu007/Simple-Compiler-using-FLex-Bison