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: *December 19, 2022*

Submitted by:

Ankan Saha

Roll: **1807060**

3rd Year 2nd Semester

Department of Computer Science and Engineering

Khulna University of Engineering & Technology (KUET)

INDEX

PAGE NO.

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 follow:

```
%}

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 (+, -, *, /, %)
- Bitwise operation (Bitwise AND, Bitwise OR)
- Relational Operations (==, !=, <, <=, >, >=)
- Logical Operations(&&, ||)
- For loop
- Switch-case
- If-else
- Built-in Power Function
- Built-in GCD Function

- Built-in LCM Function
- Built-in Min Function
- Built-in Max Function
- Single line comment
- Multiline comment
- Print Function
- Scan Function
- User Defined 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.

Syntax used in this Compiler:

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

SL No.	Syntax	Definition of Token	Example
1	#import	Import the libraries	#import input.h #import myLibrary.h
2	<! ... !>	Multiline Or Singleline comment	<! This is comment !>
3	<InT>	Specify the variable of int type	<InT> i := 5
4	<ChaR>	Specify the variable of charecter type	<ChaR> c := A
5	<DoublE>	Specify the variable of double type	<Double> d := 4.36
6	<EmptY>	This is void like C	<ReturN> <EmptY>
7	<ReturN>	Return something from function	<ReturN> i
8	<PrintF>	Prints the variable value and Strings	<PrintF>((i)) <PrintF>(($string$))
9	<ScanF>	Used for scanning user input value of a variable	<ScanF>((d))
10	IncOnE	Incement by one	IncOnE((i))
11	DecOnE	Decrement by one	DecOnE((i))

12	<IF>	If statement like C	IF((i << 7)){{
			}}

13	<ElsE>	Else statement like C	<ElsE>{{
			}}

14	<SwitcH>	Similar to Swich in C	<SwitcH>((i))
15	<CasE>	Similar to Case in C	<CasE> 1:: {{ }}
16	<DefaulT>	Similar to Defualt in C	<DefaulT> :: {{ }}
17	<FoR>	Denotes the for loop, which can execute certain block for a certain time	<FoR> ((i:=0 ;; i << 4 ;; IncOnE((i)))) {{ }}
18	<WhilE>	While loop like C	<WhilE> ((i << 4)) {{ }}
19	GCD	Print the GCD value of two numbers	GCD ((42 ,, 35))
20	LCM	Print the LCM value of two numbers	LCM ((i ,, 7))
21	MIN	Print the MIN value of two numbers	MIN ((9 ,, i))
22	MAX	Print the MAX value of two numbers	MAX ((i1 ,, i2))
23	Function Defination	Any user can define his/her own function and call it.	<InT> func((<InT> i1:=4 ,, <DoublE> d2:= 6. 58)) {{ }} func ((i, d))
24	$$	Charecter or String declaration	$String$

25	+op+
	-op-
	op
	/op/
	%op%
	|op|
	&op&
	//op//
		Arithmatic Operators	i1 = i2 +op+ i3
			d1 = i2 -op- d3
			d1 = i2 /op/ d3
			i1 = i2 &op& i3

26	==
	!!==
	<<
	>>
	<<==
	>>==
		Relational Operator	if ((i << 10))
			{{
			    <! code !>
			}}

27	&&
	||
	!
		Logical Operator	if ((i<< 10 && j >> 6))
			{{
			  <! code !>
			}}

28	(( )) {{ }} [[ ]]	Brackets	i1:=((i2+op+ i3)) *op*i4

Table 1.1: Table of used tokens and meanings of token

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.

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. 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. flex & bison by Jobn Levine

2. LEX & YACC TUTORIAL by Tom Niemann

3. https://stackoverflow.com/questions

4. https://www.youtube.com/watch?v=U0hkkGy6_ig&t=3019s

5. https://www.oreilly.com/library/view/flex-bison/9780596805418/ch01.html