PROJECT REPORT

ON

COMPILER DESIGN

COURSE NAME: COMPILER DESIGN LABORATORY

COURSE NO: CSE 3212

DATE OF SUBMISSION: 19.12.2022

SUBMITTED TO

DOLA DAS

ASSISTANT PROFESSOR

DIPANNITA BISWAS

LECTURER

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY

SUBMITTED BY

MAHMUDUL HASAN

ROLL: 1807017

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

KHULNA UNIVERSITY OF ENGINEERING & TECHNOLOGY

Introduction:

A compiler is a computer program that translates computer code written in one programming language into another language. The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language to create an executable program.

Flex and Bison:

FLEX (Fast Lexical analyzer generator) is a tool for generating scanners. Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. Scanners perform lexical analysis by dividing the input into meaningful units. For a C program the units are *variables, constants, keywords, operators, punctuation* etc. These units also called as tokens.

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Bison is used to perform semantic analysis in a compiler. Bison is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar. Parsing involves finding the relationship between input tokens. Bison is upward compatible with Yacc: all properly-written Yacc grammars ought to work with Bison with no change. Interfaces with scanner generated by Flex. Scanner called as a subroutine when parser needs the next token.

The following diagram illustrates the workflow of flex and bison:

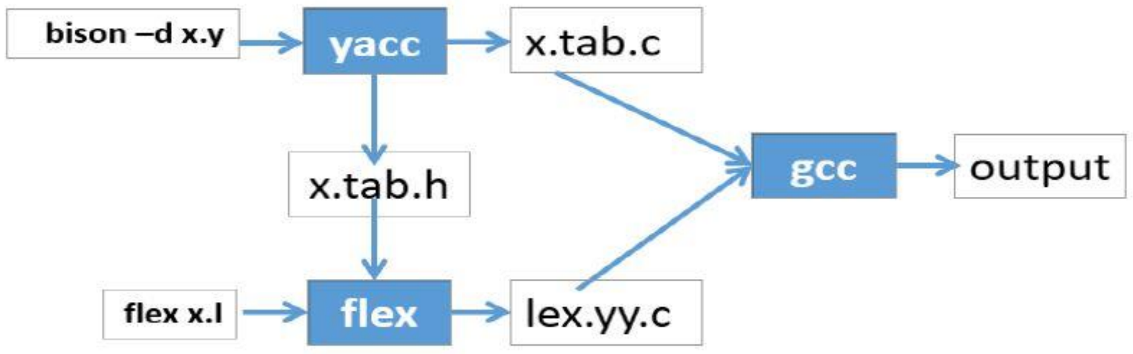

Fig 1: A diagram of how compiler build with flex and bison works.

Running Bison and Flex Program:

The command prompt script for running bison and flex program is given step by step:

1. bison -d 1807017.y
2. flex 1807017.l
3. gcc lex.yy.c 1807017.tab.c -o abc
4. abc

PROJECT

In this lab, we were assigned to design our own unique programming language using flex and bison. To complete my task, I have designed a source code which is my own language, a lexical analyzer and a syntax analyzer.

A source code with my own keywords has been written. It is passed to the lexical analyzer. The lexical analyzer program matches the syntax and generates token. Then in the syntax analyzer, there are grammar rules, data structure and other functionalities are defined. Yacc reads the grammar and generate C code for a syntax analyzer or parser. The syntax analyzer uses grammar rules that allow it to analyze tokens from the lexical analyzer and create a syntax tree.

My language has three parts. They are:

- Header Section
- User defined
- Main Function

Header Section includes all header files. The syntax for including header file is:

#shamil math.h

User defined Function:

User defined function can have parameters or no parameters and inside the function there will be multiple statements. The syntax for creating user defined function is:

blank showMe < number a, number b > [// statements //]

Before starting to our main function let's introduce some basic signs with their tokens and examples:

Sign	Token	Syntax	Example
plus	PLUS	number or variable plus number or variable;	1 plus 2 a plus 3 a plus b;
minus	MINUS	number or variable minus number or variable;	3 minus 1 a minus b
mult	MULT	number or variable mult number or variable;	1 mult 2 a mult b
div	DIV	number or variable div number or variable;	9 div 3 a div b
mod	MODULUS	number or variable mod number or variable;	10 mod 3 a mod b
>>	GT	number or variable >> number or variable;	a>>b 4>>2
<<	ST	number or variable << number or variable;	a<<b 2<<5
~	ASSIGN	variable ~ expression	v ~3 c ~ 45%3
//.*\\	COMMENT	comment	//this is comment\\
getout	BREAK	break statement	getout

Main Function:

Normally a program starts with main function. In my program, it will also start from the main function. The syntax and token are given below with example:

Function Name	Token	Syntax	Example
main	MAIN	main <> [statements;]	main [number a,b$ a~6$]

Inside the main function the following features are added:

- Data type
- Variable declaration
- Assign values to variables
- Input
- Output function (printing values)
- Single line comment
- If
- Else
- For loop
- While loop
- Arithmetic operation
- Switch
- Increment
- Decrement
- Square Root function
- Trigonometric function
- Factorial function

Data Types & Variable Declaration:

The data type of my program is given below:

Data Type	Token	Syntax	Example (Variable Declaration)
number (for integer)	INT	number variable_name	number a,b,c$
decimal (for float)	FLOAT	float variable_name	float aB$;
character (for character)	CHAR	char variable_name	character a,B,c$

Assign Values to the Variables:

Syntax: number a ~ 7, bb ~ 8;

Printing Variables/Values:

Name of function	Token	Syntax	Example

display (for printing data)	PRINT	display(variables);
		display(values);
		display(expression);
			display[a]$
			display[89]$
			display[1+2]$
			display ["Hello"]$

Comments:

// single line comment \\

Arithmetic Operations:

Syntax: a plus b$ a minus b$ 7 mult b$ 4 div 6$ 78 mod 3$

If-Else:

Name of function	Token	Syntax	Example
condition (work as if)	IF	if(variable or number or expression > or < variable or number or expression) { statements; }	condition[a>>b] { a~4+b$ }

ifnot (work as else)	ELSE	or {
		statements;
		}
			or { a minus
			b$
			}

For Loop:

Name of function	Token	Syntax	Example
jab	FOR	jab <variable or number ~ number | variable or number >< number| variable or number relop number? [statement]	jab<a ~ 1| a<~3 | a plus 1> [display[“for loop”$]

While Loop:

Name of function	Token	Syntax	Example
jabki	WHILE	jabki <variable or number < or > variable or number> [statement]	jabki< a << bb > [display[“for loop”$]]

Switch-Case:

Name of function	Token	Syntax	Example
change	SWITCH	change<number> [cases$]	change <2> [ ]
occassion	CASE	occassion case_number: statements$ break$	occassion 1: a+b$ br$
default	DEFAULT	default: statements$	default: a+7$

Increment-Decrement:

Name of function	Token	Syntax	Example
inc (for increment)	INC	variable inc$	aA inc$
dec (for decrement)	DEC	variable dec$	bB dec$

Some Functions:

Name of function	Token	Syntax	Example
root (for square root)	SQRT	root(variable)$ root(number)$	root<a>$ root<49>$
ssin (for finding the sin of number)	SIN	ssin<variable> ssin<number>	ssin<45>$
ccos (for finding the cos of number)	COS	ccos<variable> ccos<number>	ccos<45>$
log (for finding the cos of number)	LOG	log<variable> log<number>	log<7>$
fact (Factorial)	FACT	fact(variable or number)	fact<5>$
pow (power)	POW	Varible or number pow variable or number$	a pow 4$

A Simple Input and Output to My program:

Input	Output

#shamil include.h
 	
blank showMe<>[
        number tt$
        tt ~ 33$
        display[tt]$
    ]
 	
number main <> [
    number a,bb,c, d, aA, bB$
    a ~ 4$
    bb ~ 2$
 	
    d ~ a plus bb$
    display[d]$
 	
    d ~ a minus bb$
    display[d]$
 	
    d ~ a mult bb$
    display[d]$
 	
    d ~ a div bb$
    display[d]$
 	
    d ~ a mod bb$
    display[d]$
 	
    d ~ d incr$
    display[d]$
 	
    d ~ d decr$
    display[d]$
 	
    c ~ bb mult a minus 3$
    display[c]$
	Variable declared
	
	Print expression 33
	
	User defined function declared
	
	Variable declared
	
	Addition of 4+2 = 6
	
	Print expression 6
	
	Minus of 4-2 = 2
	
	Print expression 2
	
	Multiplication of 4*2 = 8
	
	Print expression 8
	
	Division of 4/2 = 2
	
	Print expression 2
	
	Modulus of 4 % 2 = 0
	
	Print expression 0
	
	Print expression 1
	
	Print expression 0
	
	Multiplication of 2*4 = 8
	
	Minus of 8-3 = 5
	
	Print expression 5

```
   //if else\\

   condition < 10 ~~ 10 > [
      1 plus 2 div 3$
   ]
   ifnot_cond < a ~~ 2 > [
      3 minus 3$
   ]
   ifnot [
      10 mod 3$
   ]

   ssin<45>$

   log<5>$

   ln<10>$

   bb pow a$

   root<49>$

   factorial<5>$

   jabki< a << bb > [
      condition< 1<<2 >[
         display["This is inside
while if-else block"]$
      ]
      ifnot[
         200 minus 2$
      ]
   ]

   jab<a ~ 1| a <~3 | a plus 1>[
      display["This is inside For
loop"]$
   ]

   //Switch case implement\\
```

```


This is a comment

Equal: 10==10

Division of 2/3 = 0

Addition of 1+0 = 1

Value of expression: 1

Equal: 4==2

Minus of 3-3 = 0

Value of expression: 0

Modulus of 10 % 3 = 1

Value of expression: 1

In If condition


The value of Sin(45) is: 0.71
Value of expression: 0
Value of Log(5) is 0.698970

Value of expression: 0
Value of ln(10) is 2.302585

Value of expression: 2

Power of: 2^4 is: 16
Value of expression: 16


Square root of the value = 7.000000

Value of expression: 7
```

change<2> [
    occassion 1:
        1 plus 2$
        getout$
    occassion 2:
        3 mult 3$
        getout$
    default:
        4 div 2$
]
]
	Factorial of 5 is 120
	
	Value of expression: 120
	
	Smaller than: 4<2
	
	Smaller than: 1<2
	
	String is: This is inside while if-else block
	
	Minus of 200-2 = 198
	
	Value of expression: 198
	
	In If condition
	
	String is: This is inside For loop
	
	Expression value : 1
	
	Expression value : 2
	
	Expression value : 3
	
	This is a comment
	
	Addition of 1+2 = 3
	
	Multiplication of 3*3 = 9
	
	Matched for Case: 2
	
	Division of 4/2 = 2
	
	Switch case is declared
	
	Main function END
	Header file added

Discussion and Conclusion:

This lab was very interesting to all of us as we are now able to create our own language and compiler. So far, we code in normally C/C++/java/python but now we are able to run our own program. This is a bottom up parser and the parser generate a set of tokens. In a program Conditional logic, Loops, Variable declaration, Mathematical function, array, header file are used. If any grammar matches with the input text then the compiler shows the token is declared. This compiler is written into C programming language.

References:

1. Flex, version 2.5 by Vern Paxson

2. https://whatis.techtarget.com/definition.