Khulna University of Engineering & Technology

Department of

Computer Science and Engineering

Course Title: Compiler Design Laboratory

Course Code: CSE 3212

Submitted to:

Nazia Jahan Khan Chowdhury

Assistant Professor

Department of Computer Science and Engineering

Khulna University of Engineering & Technology

Dipannita Biswas

Lecturer

Department of Computer Science and Engineering

Khulna University of Engineering & Technology

Submitted by:

Proloy Karmakar

Roll: 1907051

Year: 3rd

Semester: 2nd

Date of submission:22-11-2023

Department of Computer Science and Engineering

Khulna University of Engineering & Technology

Introduction:

Flex (fast lexical analyzer generator) is a program for generating lexical analyzers or patterns that reads the given input files for a description of the scanner to generate. The description is in the form of pairs of regular expressions and C code, called ‘rules’. Flex generates as output a C source file, lex.yy.c. On the Other hand, GNU **Bison** is a general purpose Parser Generator that converts a CFG (Context Free Grammar) Description into a C program to parse that grammar. It is also called YACC (Yet another Compiler Compiler). Bison is responsible for deducing the relationship between the tokens passed on to it by the Lexical Analyzer. Bison produces parser from the input file provided by the user. main program calls yyparse( ) that by default calls yylex(), which is automatically generated by the flex when it is provided with a .l file and this yylex() function is expected by parser to call to retrieve tokens from current/this token stream. As a result yylex puts the value of the token in a global variable named yylval.

Based on the diagram mentioned, we provide the commands along with the .l and the .y file to the Flex and the Bison respectively. Flex gives us a .c file and Bison gives one .h and one .c file as output. Now gcc compiler is used to link the C files to generate one executable program file with the extension .exe

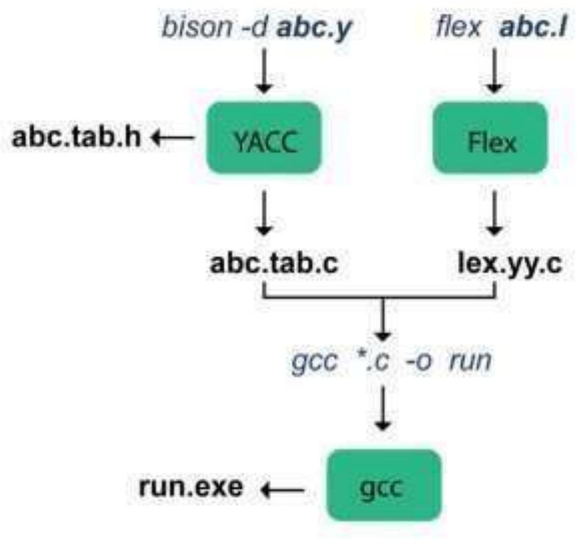

Diagram 1. Flex and Bison workflow

Tokens:

RUN: identifies the 'run' keyword, used to initiate the main program

NUMBER: identifies any of the digits from 0 to 9

INT: identifies 'int', is used to declare and assign integer type variables

DOUBLE: identifies 'double' data type, is used to declare double type variables

CHAR: identifies 'char' data type, is used to declare character type variables

PRINTVAR: identifies 'sv<<', is used to show the value inside a variable

PRINTLN: identifies 'nl', is used to print an empty line or a new line

PRINTSTR: identifies 'ss<<', is used to print a line of string

PRINTFUNC: identifies 'sf<<', used to show the output carried out by a function

UDFUNCTION: identifies 'function', is used to declare a user defined function

BB: identifies '{', is used to denote the starting of a block

BE: identifies '}', is used to denote the ending of a block

PB: identifies ')', is used to denote the starting of an expression to be evaluated

PE: identifies '(', is used to denote the ending of an expression to be evaluated
COMMA: identifies ','

SEMICOLON: identifies ';' which is used to denote the ending of a statement

ASSIGN: identifies '=' sign, is used commonly to assign the value to a variable
PLUS: identifies '+' sign which is used to perform addition

MINUS: identifies '-' sign which is used to perform subtraction

MULTIPLY: identifies '*' sign which is used to perform multiplication **DIVIDE :** identifies '/' sign which is used to perform division

MOD: identifies '%' sign which is used to perform modulus operation

LESSTHAN: identifies '<' sign which is used to compare numbers and expressions

GREATERTHAN: identifies '>' sign, is used to compare numbers and expressions.

LESSEQUAL : identifies ‘<=’ sign, is used to compare numbers and expressions

GREATEREQUAL : identifies ‘>=’ sign, is used to compare numbers and expressions.

EQUAL : identifies ‘===’ sign, is used to compare numbers or expressions if equal.

NOTEQUAL : identifies ‘!==’ sign, is used to compare numbers or expressions if not equal

MAXNUMBER : identifies ‘MAX’ denoting the name of the function that gives number greater in the magnitude.

MINNUMBER : identifies ‘MIN’ denoting the name of the function that gives the number lesser in magnitude .

COMPARE: identifies ‘And’ which is used as a keyword to use the MAX function.

COMPAREREVERSE: identifies ‘and’ which is used to perform the MAX function.

REVERSE : identifies ‘REVSTR’ that is used to use the function that reverses a string.

SORT: identifies ‘SORT’ that is used to use the function that sorts the letters of a string alphabetically

FACT : identifies ‘FACT’ function to output the factorial of a given integer

ODDEVEN : identifies ‘ODDEVEN’ function that checks whether an integer is odd or even.

SUMDIGIT: identifies ‘SUMDIGIT’ function which gives the summation of the digits of the number provided

REVNUM : identifies ‘REVNUM’ function. This is used to output a number reversed

SINFUNC : identifies ‘sin’, is used to carry out the trigonometric sine function

COSFUNC : identifies ‘cos’, is used to carry out the trigonometric cosine function

TANFUNC : identifies ‘tan’, is used to carry out the trigonometric tangent function

LOGFUNC : identifies ‘log’, is used to perform the logarithmic function of base ‘e’

LOG10FUNC : identifies ‘log’, is used to perform natural logarithmic function

GCDFUNC : identifies ‘gcd’, is used to carry out the GCD operation

LCMFUNC : identifies ‘lcm’, is used to carry out the LCM operation

POWERFUNC : identifies ‘^’, is used to find the power of an integer

STR : identifies anything written enclosed by double quotes sign “” and is used to represent a string

ID : identifies the name of a variable

IF : identifies ‘if’, is used to denote the if clause

ELSE : identifies ‘else’, is used to denote the else clause

ELSEIF : identifies ‘elseif’, is used to denote the else if clause

FOR : identifies ‘for’ keyword, is used to initiate the operation of a for loop

TO : identifies ‘->’, is used as a keyword to execute the loop successfully

COLON : identifies ‘:’, is used as a keyword to perform the loop successfully

SWITCH : identifies ‘switch’ that is used to initiate the switch-default operation

DEFAULT : identifies ‘default’ that is used to perform the instructions to be executed by default if no previous cases of value is matched

Features:

Header files

We can add header files starting with the keyword import with a leading ‘!’ and after import one or more spaces are mandatory. The name of the header file has to include at least one uppercase or lowercase letter with optional digits afterwards. We must add a trailing ‘!’ right after the name of the header file.

Valid Syntax: !import abc!, !import essentials.h!, !import maths789!

Single line comment

A single line comment starts with the symbol ‘#’ and after this symbol any letter or digits or special character in the particular line will be ignored by the compiler

Valid Syntax: # This is a single line comment

Single Line Comment

Multi line comment

Multi line comment starts as well as ends with the symbol ‘// ‘ and within this symbol any letter or digits or special character will be ignored by the compiler.

As soon as the trailing ‘//’ is found the compiler will start checking for tokens.

```
Valid Syntax:  // mul
                   ti
                   li ne comm
          ent //
```

The ‘main’ function

This is the bare minimum function to be able to compile a program successfully. This function can contain zero or more statements inside it it execute the program. The **run** keyword is equivalent of the ‘main’ keyword used in C programming language. The following curly braces after the run keyword will contain the statements to be executed. We have to be careful about the semicolon after the ending curly brace.

```
Valid Syntax:  run {  statement 1 ;
                   };
```

Arithmetic operators

The convention of mostly used arithmetic operators are followed in this compiler. Available operators are listed below :

Operator Sign	**Operation**	**Syntax**
+	Addition	3 + 5

-	Subtraction	7 - 6
*	Multiplication	2 * 4
/	Division	8 / 2
^	X to the power of n	5 ^ 3
gcd	GCD	4 gcd 3
lcm	LCM	9 lcm 3

Assignment & Conditional operators

The following six conditional operators are available for this project :

Operator sign	Operation	Syntax
<	Less than	3 < 5
>	Greater than	7 > 6
<=	Less than or equal to	2 <= 4
>=	Greater than or equal to	8 >= 2
===	Equal to	5 === 5

!=	Not equal to	5 != 7

Data types

This compiler supports three of the most commonly used data types : int, double and char referring to integer, floating point values and characters respectively. We can declare variables based on the above mentioned three variations of data types.
Valid Syntax: int a, int abc, double bbb, char ccc ;

Variables

Variables can include either letters or numbers or the three special characters :

the 'at' sign @, the dollar sign $, the underscore sign _ but no spaces are allowed in the name of the variable.
We are allowed to name the variable with only digits with atleast one or more letters or special characters mentioned above.
Valid Syntax: int _a, int $@abc56, double @bb34b, char cc_c_, int 345__ ;

Assigning values in a variable

We can initialize a value inside a variable while we declare it. We can also provide expressions to initialize a variable. Previously initialized variables can also be copied to another variable using the assignment operator. For example, the following styles are considered valid: int a = 30, int p = 4 + 5, int c = p , int count = c + 3 ; etc.

Printing the value of a variable

In order to print the value that is contained in a previously declared variable we are to write the keyword sv<< with parenthesis afterwards that ends with a semicolon. Here sv refers to 'show variable'. Inside the parenthesis we have to mention the name the of the variable. For example, sv<< (abc) ; is considered a valid syntax to print the value of a variable where abc is the name of the variable.

Printing a string

To print a simple string or a sentence we are to write the keyword ss<< with parenthesis afterwards that ends with a semicolon. Here ss refers to 'show string'. Inside the parenthesis we have to provide our desired string. We have to be careful that the string we are going to provide must be enclosed with double quotes sign within the parenthesis.
Valid Syntax: ss<< ("this is a valid string") ;

Printing the output of a function

To print the output from the available built in functions we are to write the keyword sf<< with the name of the functions available. Here sf refers to 'show function'. After the keyword we directly call the function and provide required arguments within the parenthesis to get an output value. This output value of the function is going to be printed on the console. Valid Syntax: sf<< FACT (5);

Printing newline

We are to use the nl(); keyword to add a newline to our code. Here nl refers to newline.

If - else

The structure of if else is quite similar with that of the C programming language. The keyword if is used with parenthesis that takes expressions and upon evaluating the expression if it gives a valid result then it enters the following curly braces, otherwise it moves on to the next set of curly braces to perform further instructions.

Valid Syntax: if (4 < 5) { #executing if block } else {
#executing else block } **If - elseif - else**

The structure of if elseif else is quite similar with that of the C programming language. The keyword if is used with parenthesis that takes expressions and upon evaluating the expression if it gives a valid result then it enters the following curly braces, otherwise it moves on to the next set of curly braces upon validating the expression preceding elseif keyword to perform further instructions. If none of the expressions corresponding to the if or elseifs turn out to be true then the execution will move to the else block.

Valid Syntax: if (4 > 5) { #executing if block } elseif (5<8) { #executing elseif block } else {
#executing else block }

For loop

Unlike for loop in C Programming language, in this project the for loop takes three values altogether. The first one denotes the start of the counting, the second one refers to the ending of the counting with a trailing colon and the last value indicates the step of counting inside the for loop. Between the starting count variable and the ending count variable we have to use the 'to' sign. We have to be careful that the starting count and ending count variable have to be initialized previously outside the original for loop syntax. For example, int starting = 1, ending = 5;

```
for (  starting -> ending  :  1 ) {
            statement1 ;
 }
```

Switch-default

Similar to the C programming switch-case mechanism here we have 'switchdefault'. The keyword switch is used to initiate the mechanism. All it takes is an expression enclosed by parenthesis and it checks line by line with the value of the expression and if it finds the value then it executes the instructions inside the corresponding block. If none of the values match with the switch-variable then the block corresponding to default will get executed. If there's no default statement, and no value match is found, none of the statements in the switch body get executed. There can be at most one default statement.

Valid syntax :

```
          int stw = 4*2;                switch
( stw )    {
                3:  { ss<<("switch variable 3"); }              4:   {
ss<<("switch variable 4"); }                      5:  { ss<<("switch variable 5"); }
                8:  { ss<<("switch variable 8"); }              default: {
ss<<("default is executed");}    }
```

Function Declaration

We can declare functions only before the run keyword. Firstly, we are to specify the return type followed by the keyword function and then one or more space have to be added. Then comes the name of the function that can include one or more number of letters. After the name the of the function we can specify the arguments of the functions enclosed by parenthesis; multiple arguments have to be separated by commas.

```
Valid Syntax:      int function init ( int p, int q ) {
                                     statement1 ;
                       }
```

Built in functions

1. Minimum of two given integers :

Takes two arguments separated by the keyword 'and' and gives the value that is lower in magnitude between the two given values.

sf<< MIN (400 and 23) ;   gives the output : 23 .

2. Maximum of two given integers :

This function takes two arguments separated by the keyword 'And' and gives the value that is greater in magnitude between the two given values.

sf<< MAX (100 And 21) ; gives us the output : 100

3. Reversing a string :

We can use this function to reverse a string given.

sf<< REVSTR ("terabyte"); gives the output : etybaret**.**

4. Sorting the characters of a string alphabetically :

This function turns out to be useful to sort the letters of a given string alphabetically.

sf<< SORT ("zwabgdrtef") ; gives us the output : abdefgrtwz

5. Checking if given integer is odd or even: Useful to figure out if a given integer is odd or even.

sf<< ODDEVEN(5); gives us the output : odd

6. Factorial of a number :

Gives a value equal to the factorial of the given integer as input   sf<< FACT(5); gives us the output : 120

7. Reversing the digits of an integer :   Outputs reversing the number entered as input.

sf<< REVNUM(345235); gives us the output : 532543

8. Sum of digits of an integer :

This function sums up the digits of an input integer value and shows the output   sf<< SUMDIGIT(999);   gives us the output : 27

9. Trigonometric functions : The basic three trigonometric functions can be used like following

Operation	Valid Syntax
sinx	sf<< sin(90) ;
cosx	sf<< cos(45) ;
tanx	sf<< tan(30) ;

10. Logarithmic functions : Popular logarithmic functions are available as a built in function
Operation   Valid Syntax   logx   sf<< log(2) ;   log10x sf<< log10(2) ;

Discussion:

In this project, I have designed a programming language as I wish to have. Here, I implemented variable declarations, error checking, redeclaration checking, comment, header file including, if, else, else if condition, for loop, while loop, functions, switch case from my own point of view and logic to make it more user friendly.

Conclusion:

To implement this lab project, I have used flex and bison as the required tools. Flex is a fast lexical analyzer generator and identifies tokens and bison is a tool which receives the token and parses them according to a certain context free grammar (CFG). It has a starting symbol and from there we can expand our grammar as required.

Lastly, it can be concluded that, this grammar will be so much easier and efficient to the users as we are already introduced to the C programming language.

References:

1. https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator
2. Lab manual