REPORT

Course No : CSE 3212

Course Title : Compiler design Laboratory.

Topic : Construct a Mini Compiler.

Submitted to :

Dola Das

Lecturer,

Department of Computer Science and Engineering,

Khulna University of Engineering and Technology.

Ahsan Habib Nayan

Lecturer,

Department of Computer Science and Engineering,

Khulna University of Engineering and Technology.

Submitted by :

Md. Asiful Islam Miah

Roll: 1707082

Department of Computer Science and Engineering,

Khulna University of Engineering and Technology,

Date of Submittion : 15 June 2021.

1. About Flex :

flex is a tool for generating scanners: programs which recognize lexical patterns in text. flex reads the given input files (or its standard input if no file names are given) 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', which defines a routine yylex. Compile and link this file with the `-lfl' library to produce an executable. When the executable runs, it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding C code.

Basic Structure of Flex :

```
%{
    // Definitions
%}

%%
Rules
%%

User code section
```

2. About Bison :

Bison, also known as yacc, is a parser generator that can be used to facilitate the construction of the frontend of a compiler. Bison reads a specification of a context-free language, warns about any parsing ambiguities, and generates a parser that reads sequences of tokens and decides whether the sequence conforms to the syntax specified by the grammar. Bison by default generates LALR(1) parsers.

Basic Structure of Bison :

```
%{
    C declarations
%}

Bison declarations

%%
    Grammar rules
%%

Additional C code
```

Comments enclosed in `/* ... */' may appear in any of the sections.

3. Description of Compiler :

A manual for the compiler I have designed is given below :

Tokens :

We know that a lexical analyzer reads and converts the input into a stream of tokens to be analyzed by the parser. The tokens that I have used in the making of my compiler has been described below:

1. **MAIN :** This token is returned to the parser when a pattern **"MAIN_FUNCTION''** is matched.
2. **BS :** The token BS is returned to the parser when a pattern of **"$*"** is matched.
3. **BE :** The token BE is returned to the parser when a pattern of **"*$"** is matched.
4. **header :** header is a token that is returned to the parser when the lexical analyzer matches the pattern for a header library function. The regex for a header library function is : **"library ".***

5. **comment :** comment is a token that is returned to the parser when the lexical analyzer matches the pattern for a single line comment. The regex for a single line comment is : **[#].***

6. **commentMulti :** This token is returned to the parser when a regex pattern : **"'*"[^##]*"*'"** is matched. Anything between double quotations is a comment.

7. **digit :** This token is returned to the parser when an integer number is scanned. The regex pattern for integer is: **[0-9]+**

8. **PRINT :** This token is returned to the parser when an string **"PRINT"** is matched by the scanner.

9. **VAR :** This token is returned to the parser for a variable. A variable is something that matches the regex pattern : **[a-zA-Z]*.**

10. **PLUS :** This token is returned to the parser when a pattern **" + "** is matched.

11. **MINUS :** This token is returned to the parser when a pattern **" - "** is matched.

12. **MUL :** This token is returned to the parser when a pattern **" * "** is matched.

13. **DIV :** This token is returned to the parser when a pattern **" / "** is matched.

14. **MOD :** This token is returned to the parser when a pattern **“ MOD ”** is matched.

15. **INT :** The token INT is returned to the parser when a pattern **“ INT ”** is matched.

16. **CHAR :** The token CHAR is returned to the parser when a pattern **“ CHAR ”** is matched.

17. **FLOAT :** The token FLOAT is returned to the parser when a pattern **“ FLOAT ”** is matched.

18. **ARRAY :** The token ARRAY is returned to the parser when a pattern **“ ARRAY ”** is matched.

19. **SIN :** The token SIN is returned to the parser when a pattern **“ SIN ”** is matched.

20. **COS :** The token COS is returned to the parser when a pattern **“ COS ”** is matched.

21. **TAN :** The token TAN is returned to the parser when a pattern **“ TAN ”** is matched.

22. **LOG :** The token LOG is returned to the parser when a pattern **“ LOG ”** is matched.

23. **LOG10 :** The token LOG10 is returned to the parser when a pattern **“ LOG10 ”** is matched.

24. **IF :** The token IF is returned to the parser when a pattern **“ IF ”** is matched.

25. **ELSEIF :** The token ELSEIF is returned to the parser when a pattern **“ ELSEIF ”** is matched.

26. **ELSE :** The token ELSE is returned to the parser when a pattern **“ ELSE ”** is matched.

27. **SWITCH :** The token SWITCH is returned to the parser when a pattern **“ SWITCH ”** is matched.

28. **CASE :** The token CASE is returned to the parser when a pattern **“ CASE ”** is matched.

29. **DEFAULT :** The token DEFAULT is returned to the parser when a pattern **“ DEFAULT ”** is matched.

30. **LOOP :** The token LOOP is returned to the parser when a pattern **“ FOR_LOOP ”** is matched.

31. **REVERSE_LOOP :** The token REVERSE_LOOP is returned to the parser when a pattern **“REVERSE_FOR_LOOP ”** is matched.

32. **WHILE :** The token WHILE is returned to the parser when a pattern **“ WHILE ”** is matched.

33. **REVERSE_WHILE :** The token REVERSE_WHILE is returned to the parser when a pattern **“REVERSE_WHILE ”** is matched.

34. **ODDEVEN :** The token ODDEVEN is returned to the parser when a pattern **“ ODD_EVEN ”** is matched.

35. **FACTORIAL :** The token FACTORIAL is returned to the parser when a pattern **“ FACT ”** is matched.

36. **-+/*<>=,():;%^[] :** These symbols returned to parser by **yytext**.

Main Function :

In every language the program execution starts from the main function. In this language, the program has to be written inside the MIAN_FUCTION function. The structure is defined below:

MAIN_FUNCTION :

$*

    # write program here

*$

Data Types :

There are mainly 3 data types in my language:

INT	Stands for Integer. Supports any integer number. e.g: 47.53, 15.6, -0.05 etc.
FLOAT	Stands for Float number. Supports any real number. e.g: 10,53,47 etc e.g: 47.53, 15.6, -0.05 etc.
CHAR	Stands for Float number. Supports any character . e.g: “Himel”

Single line Comment & Multiple line Comment :

Anything starting with "#" is considered a single line comment. And anything in between '* & *' are considered as multiple line comment.

The parser has no action for the comment while parsing.

e.g:

```
# This is a single line comment.

'* This is a

Multiple line

Comment *'
```

Variable Declarations & Value Assignment :

```
INT himel, a;
himel = 5;
a = 10;

char name;
name = Himel ;

float f;
f = 20.5;
```

Variables can be declared & assigned in a separate line. The colon " = " symbol is used for assignment.

Displaying Values :

The PRINT( ) function is used to display anything that's passed inside it as a parameter. It can be an expression, variable or a string.

e.g:

PRINT ("Hello World!");

Output : Hello World!

Operators, Operations & Precedence :

The operators and their operations are described below:

1. **Assignment :** The ' = ' operator is used for assignment.
   E.g :
   INT h;
   h = 5;
   A value of 5 is assigned to variable 'h' which is of type: INTEGER.
2. **Addition (+) :** The plus (+) operator adds two expressions around it & is a binary operator.

E.g;

PRINT(80+2);

Plus operator adds 80 & 2 and then colon PRINT the value 82.

3. **Subtraction (-) :** The minus (-) operator is a binary operation. It subtracts the 2nd expression from the 1st.

   E.g:

   82-2;

   Minus operator subtract 2 from 82.

4. **Multiplication (*) :** The multiplicative (*) operator is binary, it returns the product of the two expressions around it.

   E.g:

   Mul = 5*6;

   The product of 5 & 6 which is equal to 30 is assigned to Mul.

5. **Division (/) :** The division operator is binary. It divides the 1st expression by the 2nd expression & returns the value. A division by '0' error is displayed if the 2nd expression has value equal to 0.

   E.g:

   div = 82/2;

   A value of 41 is assigned to div.

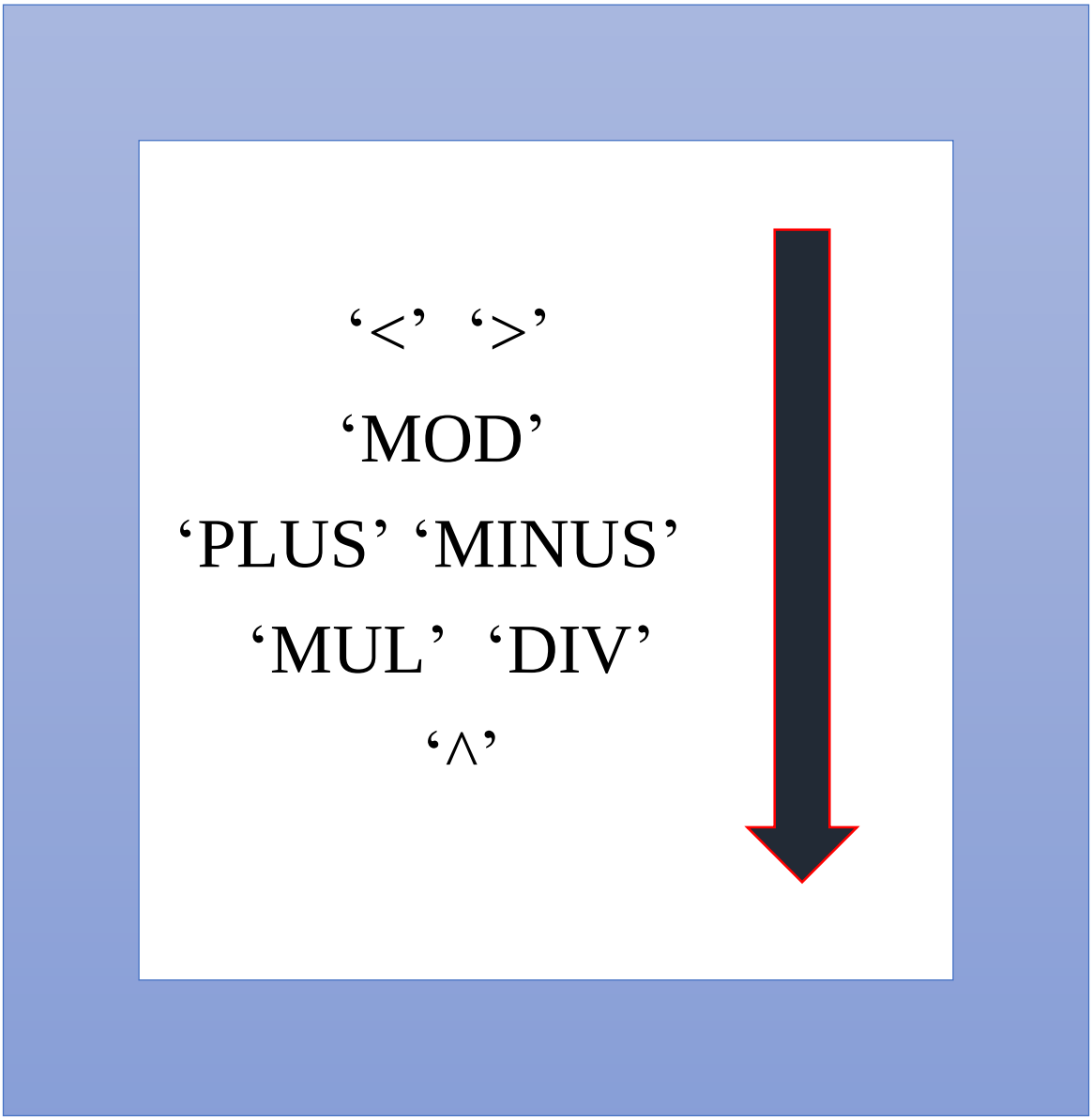
6. **Exponent (^) :** The caret(*) operator is binary, it returns the exponent of the 2nd expression over 1st expression.

E.g :

2^4;

It return 16.

The precedence of operators



IF, ELSEIF or ELSE Statements :

This is the if, else if, else statement equivalent of the C language. The statement has three structures which are as follows:

1. Only one IF statement :

```
IF(expression)
$*
        # if expression is true, execute this block
*$
```

2. One IF statement and ELSE statement :

```
IF (expression)
$*
        # if expression is true, execute this block
*$

ELSE
$*
        # Otherwise execute this block
*$
```

3. One IF statement, one/multiple ELSEIF statements and one ELSE statement :

IF (expression1)

$*

    # if expression 1 is true then execute this block

***$**

ELSEIF (expression2)

$*

    # if expression 2 is true then execute this block

***$**

.

.

.

ELSEIF (expression n)

$*

    # if expression n is true then execute this block

***$**

ELSE

$*

    # Otherwise execute this block

***$**

For loop statements :

The For loop has 3 parts separated by two comma in between its parenthesis. The structure is as follows:

FOR_LOOP (start_val, end_val, increment)

$*

    # do something

***$**

start_val => start loop at “start_val”.

end_val => end loop at “end_val”.

increment => increment start_val by “increment” value after every loop.

Reverse For loop statements :

The Reverse For loop has 3 parts separated by two comma in between its parenthesis. The structure is as follows:

REVERSE_FOR_LOOP (start_val, end_val, deccrement)

$*

    # do something

***$**

start_val => start loop at “start_val”.

end_val => end loop at “end_val”.

deccrement => decrement start_val by “deccrement” value after every loop.

While loop statements :

The structure of While loops is as follows :

WHILE (start_val > end_val)

$*

    # do something here

***$**

The loop starts at “start_val” and ends at “end_val”. After every turn the value of start_val is incremented by 1.

This loop continuous until the statement “start_val > end_val” satisfies.

Reverse While loop statements :

The structure of Until do loops is as follows:

REVERSE_WHILE (start_val < end_val)

$*

    # do something here

***$**

The loop starts at “start_val” and ends at “end_val”. After every turn the value of start_val is decremented by 1.

This loop continuous until the statement “start_val < end_val” satisfies

Switch Case statements :

The SWITCH statement has one argument in which it takes an expression. There are one or more Match cases which matches the expression with certain values. If no value is matched then contents of DEFAULT block is executed. The structure is as follows:

SWITCH (expression1)

$*

CASE expression2 : # if expression1 matches with expression2 then execute this block.

CASE expression3 : # if expression1 matches with expression3 then execute this block.

.

.

.

CASE expression n : # if expression1 matches with expression n then execute this block.

DEFAULT : # if no expressions match with exp1 then execute this block.

***$**

Some Built-in Functions :

1. **FACT (x) :** The FACT ( ) function takes one argument and displays the factorial of that value.
2. **SIN (x) :** The SIN ( ) function take one argument & displays the sin of that value.
3. **COS (x) :** The COS( ) function take one argument & displays the cosine of that value.
4. **TAN (x) :** The TAN( ) function take one argument & displays the tangent of that value.
5. **LOG (x) :** The LOG( ) function take one argument & displays the logarithm of that value.
6. **LOG10 (x) :** The LOG10( ) function take one argument & displays the logarithm of power 2 of that value.