Compiler Lab Report

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

Course No : CSE 3212

Submitted to :

Dola Das

Lecturer

Md. Ahsan Habib Nayan

Lecturer

Department of Computer Science and Engineering

Khulna University of Engineering and Technology

Submitted By :

Lamia Hossain

Roll : **1707110**

Department of Computer Science and Engineering

Khulna University of Engineering and Technology

Submission Date : 15 - 06 - 2021

Introduction of Flex and Bison :

Flex (fast lexical analyzer generator) :

Flex is a software alternative to lex which generates lexical analyzers (also known as "scanners" or "lexers").

Flex takes a program written in a combination of Flex and C, and it writes out a file (called lex.yy.c) that holds a definition of function yylex().

yylex reads from the file stored in variable yyin. When yylex is finished, it call function yywrap().

If yywrap() returns 1, then yylex returns 0 to its caller. That means "end of file".

If yywrap() returns 0, then yylex assumes that you have stored a different file into yyin, and it starts reading that file.

A Felx input file has 3 sections :

{ definitions }

%%

{ rules }

%%

{ user subroutines }

Bison :

GNU Bison, commonly known as Bison, is a parser generator. Bison reads a specification of a context-free language, warns about any parsing ambiguities, and generates a parser (either in C, C++, or Java) that reads sequences of tokens and decides whether the sequence conforms to the syntax specified by the grammar. The generated parsers are portable: they do not require any specific compilers. Bison converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar. Bison is upward compatible with Yacc: all properly-written Yacc grammars ought to work with Bison with no change.

The input file consists of three sections, separated by a line with just `%%' on it. A Bison input file is given below :

```
%{
C declarations (types, variables, functions, preprocessor commands)

%}

/* Bison declarations (grammar symbols, operator precedence decl., attribute data type) */

%%

/* grammar rules go here */

%%

/* additional C code goes here */
```

About Project :

For this project we have to design a compiler. Compiler is a a program that converts instructions into a machine-code or lower-level form so that they can be read and executed by a computer. The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. The phases are : *Lexical Analysis* , *Syntax Analysis* , *Syntax Analysis* , *Intermediate Code Generation* , *Code Optimization* , *Code Generation* , *Symbol Table* . If these phases are performed in sequence, then a compiler can be designed perfectly.

Equipment :

Flex, Bison .

Description :

In the project "1707110.l" (contains token) is the lex file, "1707110.y" (contains grammar) is the yacc file, in.txt is the input file.

Flex divides the inputs into meaningful units. For a C program the units are variables, constants, keywords, operators, punctuation etc. These units also called as tokens.

The **tokens** I have used in my compiler are :

1. **INT :** This token is returned when "Int" pattern is found.
2. **CHAR :** This token is returned when "Char" pattern is found.
3. **FLOAT :** This token is returned when "Float" pattern is found.
4. **PRINT :** This token is returned when "print" pattern is found.

5. **LOG10 :** This token is returned when “Log10” pattern is found.
6. **LN :** This token is returned when “Ln” pattern is found.
7. **SIN :** This token is returned when “sin” pattern is found.
8. **COSEC :** This token is returned when “cosec” pattern is found.
9. **COS :** This token is returned when “cos” pattern is found.
10. **SEC :** This token is returned when “sec” pattern is found.
11. **TAN :** This token is returned when “tan” pattern is found.
12. **COT :** This token is returned when “cot” pattern is found.
13. **FACTORIAL :** This token is returned when “fact” pattern is found.
14. **SUM :** This token is returned when “sum” pattern is found.
15. **TO :** This token is returned when “to” pattern is found.
16. **MAX :** This token is returned when “max_between” pattern is found.
17. **MIN :** This token is returned when “min_between” pattern is found.
18. **GCD :** This token is returned when “gcd” pattern is found.
19. **LCM :** This token is returned when “lcm” pattern is found.
20. **ROOT :** This token is returned when “root” pattern is found.
21. **FIBONACCI :** This token is returned when “fib” pattern is found.
22. **PALINDROME :** This token is returned when “palindrome” pattern is found.
23. **MUL :** This token is returned when “mul” pattern is found.
24. **REVERSE :** This token is returned when “reverse” pattern is found.
25. **LEAP :** This token is returned when “leap” pattern is found.
26. **INC :** This token is returned when “++” pattern is found.
27. **DEC :** This token is returned when “--” pattern is found.
28. **MULC :** This token is returned when “**” pattern is found.
29. **DIVC :** This token is returned when “//” pattern is found.
30. **POWC :** This token is returned when “^^” pattern is found.
31. **LESS_THAN :** This token is returned when “<<” pattern is found.

32. **GREATER_THAN :** This token is returned when “>>” pattern is found.
33. **NOT :** This token is returned when “!” pattern is found.
34. **IF :** This token is returned when “jodi” pattern is found.
35. **ELSE :** This token is returned when “nahoy” pattern is found.
36. **ELSE_IF :** This token is returned when “nahoy_jodi” pattern is found.
37. **FOR :** This token is returned when “for” pattern is found.
38. **MOVE :** This token is returned when “move” pattern is found.
39. **CASE :** This token is returned when “when” pattern is found.
40. **BREAK :** This token is returned when “our” pattern is found.
41. **DEFAULT :** This token is returned when “otherwise” pattern is found.
42. **WHILE :** This token is returned when “while” pattern is found.
43. **START :** This token is returned when “{” pattern is found.
44. **END :** This token is returned when “}” pattern is found.
45. **VAR :** This token is returned when “[a-zA-Z]([a-zA-Z0-9])*” pattern is found.
46. **HI :** This token is returned when “HI” pattern is found.
47. **BYE :** This token is returned when “BYE” pattern is found.
48. **MAIN :** This token is returned when “main” pattern is found.
49. **NUM :** This token is returned when “[-+]?[0-9]+” pattern is found.
50. **LEAP :** This token is returned when “leap” pattern is found.
51. **+ :** This token is returned when “ + ” pattern is found.
52. **- :** This token is returned when “ - ” pattern is found.
53. **/ :** This token is returned when “ / ” pattern is found.
54. *** :** This token is returned when “ * ” pattern is found.
55. **< :** This token is returned when “ < ” pattern is found.
56. **> :** This token is returned when “ > ” pattern is found.

57. = : This token is returned when " = " pattern is found.

58. , : This token is returned when " , " pattern is found.

59. _ : This token is returned when " _ " pattern is found.

60. (: This token is returned when " (" pattern is found.

61.) : This token is returned when ") " pattern is found.

62. : : This token is returned when " : " pattern is found.

63. ; : This token is returned when " ; " pattern is found.

64. % : This token is returned when " % " pattern is found.

65. ^ : This token is returned when " ^ " pattern is found.

66. Spaces, new lines, all other characters except the tokens are ignored.

Main Function :

In this compiler the main function starts with " **main:** " and then whole input must be put between " **HI** " and " **BYE** ". " **HI** " means **starting** of the program and " **BYE** " means **ending** of the program.

Variable Data Types and Assign its Value :

I have used three type of data types named " **Int** " (means integer type) , " **Char** " (means character type) , " **Float** "(means float type).To assign an integer type variable the type **Int** must be declared first and as every statement ends with a " **;** " so this sign need to be given at last for a valid declaration. Same goes for **Float** and **Char** type variable. If the statement is correctly placed in input file "**Valid declare**" will be printed.
Example : Int aa, b; Float a; Char z;

Then at the time of assignment " **=** " sign need to be put between the variable and value. After each valid assignment, "**Value of the variable**" and the assigned value will be printed.

Example : aa = 1; b = 5;

If the variable is already declared before, "**variable is already declared**" will be printed.

If Else if and Else :

In my compiler, " **jodi** " works same as **if** works in C program. Also " **nahoy_jodi** " works as **else if** and " **nahoy** " works as **else**. The conditions for " **jodi** " and " **nahoy_jodi** " should be given between " **(** " and " **)** ". And the operation must be given between " **{** " and " **}** ".

In my compiler I have given the opportunity to nested if-else condition.

A person can use seven combinations (jodi, jodi - nahoy, jodi -nahoy_jodi - nahoy, jodi (jodi-nahoy) -nahoy_jodi - nahoy , jodi (jodi - nahoy_jodi - nahoy) - nahoy_jodi - nahoy , jodi -nahoy_jodi (jodi-nahoy) - nahoy , jodi - nahoy_jodi (jodi - nahoy_jodi - nahoy) - nahoy)

A simple example is given below:

```
jodi(d>c) { 5+100; }
nahoy_jodi(d>aa) { 4*10; }
Nahoy { 5-4; }
```

For Loop :

In my project, " **for** " works same as **for loop** in C program. Here one can use a for loop and the output will be for how many times the loop is visited and inside operation will also be changed each time. One has to assign a value of a variable which can be used in the loop statement and there 5 operations can be executed which are addition **(++)** , subtraction **(--)** , multiplication **(**)** , division **(//)** , power **(^^)** .

Example :

```
for 1 to 3 (s=100) { s = s**3; }
```

The output will be :

For Loop statement for 1 : 300

For Loop statement for 2 : 900

For Loop statement for 3 : 2700

FOR LOOP DONE MULTIPLICATIO

While Loop :

In my project, **“ while ”** works same as **while loop** in C program. Three kind of conditions can be used for the loop execution. They are not **(!)** , greater then **(>)** , less then **(<)** . If the condition is true, program will re-enter the loop until the values become equal and the output will be for how many times the loop is visited. And lastly the output of the inside statement will be shown in the terminal.

Example :

while (11!8) { 5+1000; }

The output will be :

While condition is true for not sign.

FIRST one is greater than the SECOND.

WHILE loop no :

8

9

10

WHILE LOOP DONE DONE DONE

Result is 1005

Switch Loop :

In my compiler, “ **move-when** ” works same as switch-case loop. Here “ **move** ” , “ **when** ”, “ **out** ” and “ **otherwise** ” is used instead of **switch**, **case**, **break** and **default** respectively.

Fourteen kind of switch situations can be given for switch condition. They are : Any value, any variable which is declared before, addition/ subtraction/ multiplication/ division/ mod/ power of two numbers or variables, root/ increment/ decrement/ factorial for any number or variable, summation or multiplication for one value/variable to another value/variable.

Unlimited “ **when** ” can be added. If any “ **when** ” does not match then the “ **otherwise** ” operation will be executed.

Example :

```
move (fact 6)
{
when 1 : 1+5; out;
when 3 : 1 + 25; out;
when 6 : 5 + 10; out;
otherwise : 25 + 5; out;
}
```

The output will be :

```
CASE no :  and the Output is : 15
SWITCH LOOP DONE DONE DONE
```

Some Built in Functions :

In my compiler, there are 20 built-in functions. They are :

1. Log10 (variable/ number) : This function will return the value of the ***log of base 10*** of the given preassigned variable or the number.

2. Ln (variable/ number) : This function will return the value of the ***log of base e*** of the given preassigned variable or the number.

3. sin (number) : This function will give the output of ***sine*** value of the given degree.

4. cosec (number) : This function will give the output of ***cosecant*** value of the given degree.

5. cos (number) : This function will give the output of ***cosine*** value of the given degree.

6. sec (number) : This function will give the output of ***secant*** value of the given degree.

7. tan (number) : This function will give the output of ***tangent*** value of the given degree.

8. cot (number) : This function will give the output of ***cotangent*** value of the given degree.

9. fact variable/number : This function will return the ***factorial*** of the given preassigned variable or the number.

10. sum variable/number to variable/number : This function will return the ***summation*** of the given first preassigned variable or the value to the second preassigned variables or the numbers.

11. mul variable/number to variable/number : This function will return the ***multiplication*** of the given first preassigned variable or the value to the second preassigned variables or the numbers.

12. gcd variable/number , variable/number : This function will return the largest integer that can exactly divide both the preassigned variables or the numbers (without a remainder).

13. **lcm variable/number , variable/number :** This function will return the smallest positive integer that is perfectly divisible by both the preassigned variables or the numbers (without a remainder).

14. **root variable/number :** This function will return the value of the ***square root*** of the given preassigned variable or the number.

15. **fib variable/number :** This function will return a ***fibonacci series*** and the number of the elements in the series will be equals to the given preassigned variable or the number.

16. **max_between variable/number variable/number :** This function will return the ***maximum*** value between the given preassigned variables or the numbers.

17. **min_between variable/number variable/number :** This function will return the ***minimum*** value between the given preassigned variables or the numbers.

18. **palindrome variable/number :** This function will find whether the given preassigned variable or the number a palindrome number or not.

19. **reverse variable/ number :** This function will output the ***reverse*** of the given preassigned variable or the number.

20. **leap variable/ number :** This function will return the whether the given preassigned variable or the number is a leap year or not.

Operators :

1. **variable/number + variable/number :** This function will return the ***summation*** of the given preassigned variables or the numbers.

2. **variable/number - variable/number :** This function will return the ***subtraction*** of the given preassigned variables or the numbers.

3. **variable/number * variable/number :** This function will return the ***multiplication*** of the given preassigned variables or the numbers.

4. variable1/number1 / variable2/number2 : This function will ***divide*** variable1/number1by variable2/number2 and return the result. If variable2/number2 is zero, this function will return "Divided by 0".

5. variable/number % variable/number : This function will ***mod*** variable1/number1by variable1/number1 and return the result. If variable2/number2 is zero, this function will return "Mod by zero".

6. variable1/number1 ^ variable2/number2 : This function will find the variable2/number2 ***power*** of variable1/number1 and return the result.

7. variable/ number++ : This function will return the given preassigned variable or the number ***incremented*** by 1.

8. variable/ number-- : This function will return the given preassigned variable or the number ***decremented*** by 1.

9. variable/ number!variable/ number : This function will return the whether the given preassigned variable1 or the number1 is equal to preassigned variable2 or the number2 or not.

10. variable1/ number1 >> variable2/ number2 : This function will return the whether the given preassigned variable1 or the number1 is greater than the preassigned variable2 or the number2 or not.

11. variable1/ number1 << variable2/ number2 : This function will return the whether the given preassigned variable1 or the number1 is less than the preassigned variable2 or the number2 or not.

12. variable1 = variable2/ number2 : This function is used to ***assign*** the value of variable2 or the number2 to variable1.

Precedence and Associativity :

Here " * " and " / " has highest and same precedence. Then " + " and " - " has lowest and same precedence. All these operators have left associativity.

Comment and Header :

- **Single Line comment :** When in input file "**!s_cmnt!.***" this pattern is found the program will print "~~Single Line Comment~~".
- **Multiple Line comment :** When in input file "**!m_cmnt![a-zA-z0-9\n ~`\t@#$%^&*<>?,()./'':;['{}\|+=_-]*!m_cmnt!**" this pattern is found the program will print "~~Multiple Line Comment~~".
- **Header :** When in input file "**#import[ ][<][^\n]+[.h][>]**" this pattern is found the program will print "Its Header.....".

Conclusion :

Here a basic compiler has been built using flex and bison.

Reference :

1. https://www.tutorialspoint.com/compiler_design/compiler_design_phases_of_compiler.htm
2. https://www.google.com/search?q=what+is+compiler&safe=active&rlz=1C1CHBF_enBD896BD896&sxsrf=ALeKk001qJ1G-GVZfGDqzgn9HFynYI5NzQ%3A1623528433408&ei=8RPFYLC-GKzD3LUPwOGDKA&oq=what+is+compiler&gs_lcp=Cgdnd3Mtd2l6EAMyBwgAEEcQsAMyBwgAEEcQsAMyBwgAEEcQsAMyBwgAEEcQsAMyBwgAEEcQsAMyBwgAEEcQsAMyBwgAEEcQsANQyB1YyB1g2iBoAHAEeACAAQCIAQCSAQCYAQKgAQGqAQdnd3Mtd2l6yAEIwAEB&sclient=gws-wiz&ved=0ahUKEwjw-rjc8pLxAhWsIbcAHcDwAAUQ4dUDCA4&uact=5
3. https://www.gnu.org/software/bison/#:~:text=Bison%20is%20a%20general%2Dpurpose,LR(1)%20parser%20tables.