John Kucera

Prof. Tim Boyd

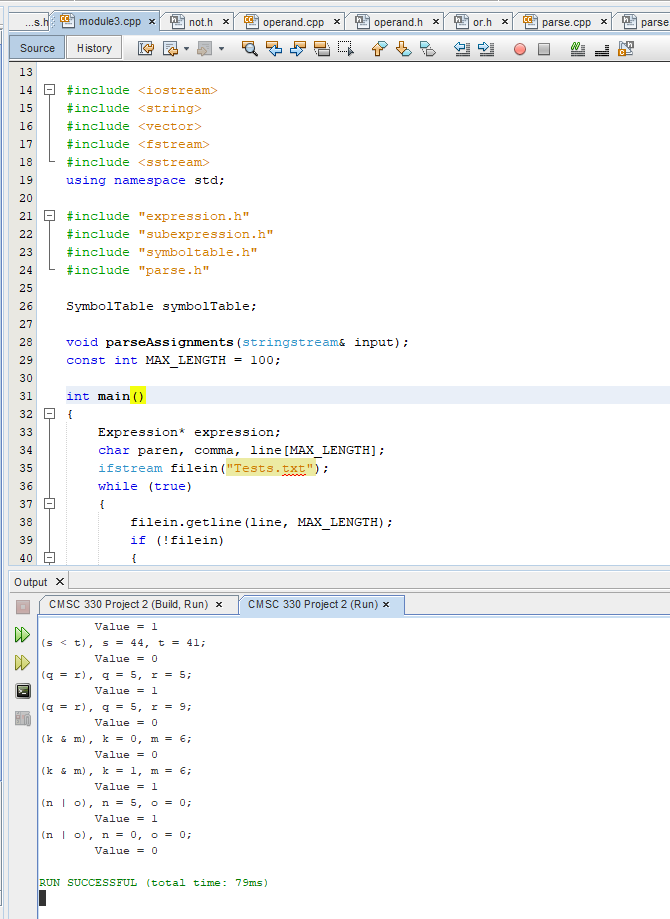
CMSC 330 Advanced Programming Languages

7 May 2020

Week 8: Project 2 Solution Description

**C++ Expression Interpreter**

**Screenshot of successful program compilation, execution, and output in Netbeans IDE:**



**Description of Process:**

Module 3 contained 14 files which I made full use of for this project: expression.h, literal.h, module3.cpp, operand.h, operand.cpp, parse.h, parse.cpp, plus.h, subexpression.h, subexpression.cpp, symboltable.h, symboltable.cpp, variable.h, and variable.cpp. I created 10 files to extend the program for this project’s requirements: and.h, conditional.h, divide.h, equals.h, greater.h, less.h, minus.h, not.h, or.h, and times.h. They come together to form a full program which extends the Module 3 Expression Interpreter, including all arithmetic, logical, relational, and conditional expression operators according to the provided grammars. The program has been modified to accept input from a file which has multiple expressions arranged one per line. Finally, all results have been changed from double to integer and the evaluate function returns an Integer type.

The main method is in module3.cpp. The user must ensure that the ifstream among the first lines of the main method directs to the correct input file. The flow of the program is like this: when the user presses “Run”, the main method will start and open the input file. As it begins to read through each line, it looks for a beginning parentheses to ensure the grammar is correct, then prints out the line. It then parses the expression and subexpression(s) inside using parse() of Subexpression class. With this, it analyzes the left operand using Operand class’s parse(). If it is a left parentheses, that means there is another subexpression inside to evaluate, then uses Subexpression parse() on that. This continues until the left operand is indeed a numerical digit, and in this case Operand’s parse() returns the literal value using Literal() of the literal.h header file.

With that left operand returned, the next character will be parsed which should always be an OPERATOR. This operator is analyzed for what it is – a switch block in Subexpression looks at the character and accordingly uses the Operator Header files to conduct the appropriate operation: Plus, Minus, Times, Divide, Greater, Less, Equals, And, Or. In the case of conditional operators, there is an if block to account for the Colon, Question Mark, and Negation Operator since their expressions are formatted differently than the typical arithmetic and relational operators. Conditional and Not (from their respective header files) are used for this.

The comma following the expression is then read in, signifying that the variable assignments are upcoming. The next character should always be a character (variable) which is parsed and associated with the value that comes after (assignment). These are INSERTED into the Symbol Table. The next comma is parsed through and the second variable assignment is Inserted into the Symbol Table as well.

Next in the main method, “Value = “ is printed and what follows is evaluate() from class Expression. Evaluate is from class Variable which uses lookup() in symbolTable which returns the appropriate value assigned. The evaluate() from the appropriate operator header file is used to correctly interpret the expression fully and produce the resultant Value. This Value is printed out in the Output under the respective Expression. The Symbol Table is cleared so that following expressions can use the same symbols without affecting each other. This repeats (in a While loop) until all lines are read and parsed. The program then ends, having printed all Expressions with their resultant Values in the Output.

**Lessons Learned:**

This is the first time I have used C++ to write a full program (not just Discussion questions) and it really took a big learning curve to get it done. I began by looking for C++ introductory material and videos online and from our textbooks, but the sense of unfamiliarity was still quite overwhelming when I had this 24-file Expression Interpreter looming in front of me. The understandings of C++ really came to mind when I looked at the provided Module 3 program.

So, the biggest lesson I learned from this project (more like something I learned about myself) is that it is much, much more efficient to learn a language by looking at a program already written in that language. Of course, when it comes to learning your first programming language, it’s best to be told the meaning of every single line, what a variable is, what this certain method does, etc. This is when the logic of programming is still trying to settle in my head. But later, when it comes to learning different languages (like going from Java/Python to C++), it really all has the same logic behind it – just with different syntax. When I was looking up C++ introductory material, I didn’t really learn anything substantial because it all treated me like I didn’t understand programming logic. On the other hand, when I took a look at Module 3 closely to interpret each line in the program, seeing how C++ worked became so simple because it was just learning new syntax for logic I already knew. Figuring out how to write the rest of the program became very simple just from looking at the syntax in the C++ of Module 3. With this in mind, I am beyond grateful that my first C++ program was accompanied with a starter like Module 3.

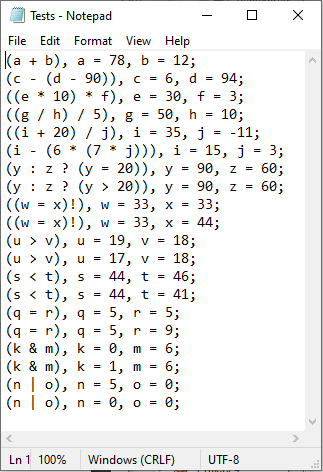
The smaller lessons I learned were just trial-and-error phases I went through and figuring out how to write C++ syntax. It quickly became normal to have so many header files and put the member functions of classes in their own .cpp files, which was something very unknown to me as I had only programmed in Java and Python before. This was also my first time using the Protected scope (as opposed to Public and Private), as well as my first time categorizing each file’s contents into their own scopes. In Java, you just label something as public or private, but in C++, there are Public and Private GROUPINGS that you have to put the file contents into.

This was one of the things I grew to like about C++ during this project: the organization of the code in comparison to Python and Java. Those two languages can be written in many different orders as the programmer would like, but C++ forces you to organize your contents into many categorized files, scopes, and classes. It’s almost refreshing to be guided like this, rather than having to organize my Java code on my own. Especially in this program where Inheritance Hierarchy is at the core of the function flow, you need a built-in structure like C++ where you can look at the code and clearly see the paths the inheritance hierarchy takes. I felt much more comfortable writing inheritance code in C++ than in other languages.

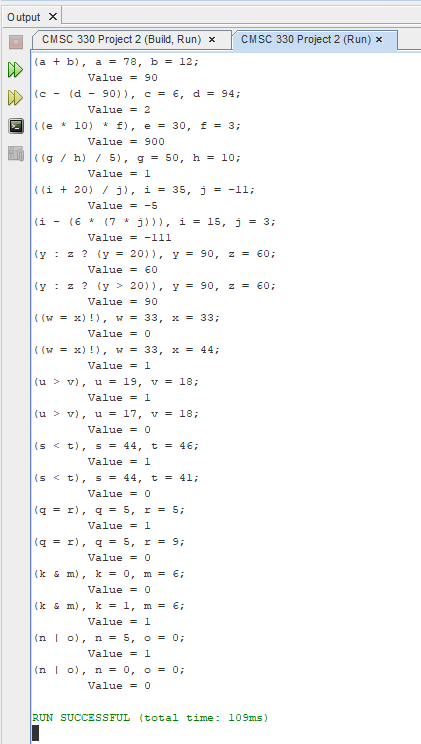
Overall, this program was a blast to extend and finish. I enjoyed adding on the different operators to the expression evaluation and getting it to read from a file. I will value this first C++ experience as it showed me the unique organization and CONVENIENCE of that language.

**Screenshots + Test Cases**:

Input file name: *Tests.txt*



Compilation with generated Results/Values in Netbeans Output:



Summary of results: SUCCESS.

* Input file was successfully read
* Each expression (one on each line) was interpreted/evaluated correctly
* Each expression followed by their Value were printed out in the Output, good readability
* All results are of type Integer
* All required operators (arithmetic, relational, conditional, logical) work correctly
* Variables can be used more than once among different expressions
* Below is a **Table of Test Cases for each Expression in Tests.txt:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test Case #** | **Aspect Tested** | **Input** | **Expected Output** | **Actual Output** | **Pass**  **?** |
| 1 | Addition Operator (+) | (a + b), a = 78, b = 12; | Value = 90 | Value = 90 | YES |
| 2 | Subtraction Operator (-), subexpression | (c - (d - 90)), c = 6, d = 94; | Value = 2 | Value = 2 | YES |
| 3 | Multiplication Operator (\*), subexpression | ((e \* 10) \* f), e = 30, f = 3; | Value = 900 | Value = 900 | YES |
| 4 | Division Operator (/), subexpression | ((g / h) / 5), g = 50, h = 10; | Value = 1 | Value = 1 | YES |
| 5 | Subexpression with different operators, negative integer | ((i + 20) / j), i = 35, j = -11; | Value =  -5 | Value =  -5 | YES |
| 6 | Subexpressions with different operators, negative integer | (i - (6 \* (7 \* j))), i = 15, j = 3; | Value =  -111 | Value = -111 | YES |
| 7 | Conditional Operator (? :), returning False (second operand) | (y : z ? (y = 20)), y = 90, z = 60; | Value = 60 | Value = 60 | YES |
| 8 | Conditional Operator (? :), returning True (first operand) | (y : z ? (y > 20)), y = 90, z = 60; | Value = 90 | Value = 90 | YES |
| 9 | Negation Operator (!), returning False (0) | ((w = x)!), w = 33, x = 33; | Value = 0 | Value = 0 | YES |
| 10 | Negation Operator (!), returning True (1) | ((w = x)!), w = 33, x = 44; | Value = 1 | Value = 1 | YES |
| 11 | Greater Than Operator (>), returning True (1) | (u > v), u = 19, v = 18; | Value = 1 | Value = 1 | YES |
| 12 | Greater Than Operator (>), returning False (0) | (u > v), u = 17, v = 18; | Value = 0 | Value = 0 | YES |
| 13 | Less Than Operator (<), returning True (1) | (s < t), s = 44, t = 46; | Value = 1 | Value = 1 | YES |
| 14 | Less Than Operator (<), returning False (0) | (s < t), s = 44, t = 41; | Value = 0 | Value = 0 | YES |
| 15 | Equality Operator (=), returning True (1) | (q = r), q = 5, r = 5; | Value = 1 | Value = 1 | YES |
| 16 | Equality Operator (=), returning False (0) | (q = r), q = 5, r = 9; | Value = 0 | Value = 0 | YES |
| 17 | And Operator (&), returning False (0) | (k & m), k = 0, m = 6; | Value = 0 | Value = 0 | YES |
| 18 | And Operator (&), returning True (1) | (k & m), k = 1, m = 6; | Value = 1 | Value = 1 | YES |
| 19 | Or Operator (|), returning True (1) | (n | o), n = 5, o = 0; | Value = 1 | Value = 1 | YES |
| 20 | Or Operator (|), returning False (0) | (n | o), n = 0, o = 0; | Value = 0 | Value = 0 | YES |