

**Project 2: Extending the C++ for Statements of Expression Language**

**Victoria Lee**

UMGC: University of Maryland Global Campus

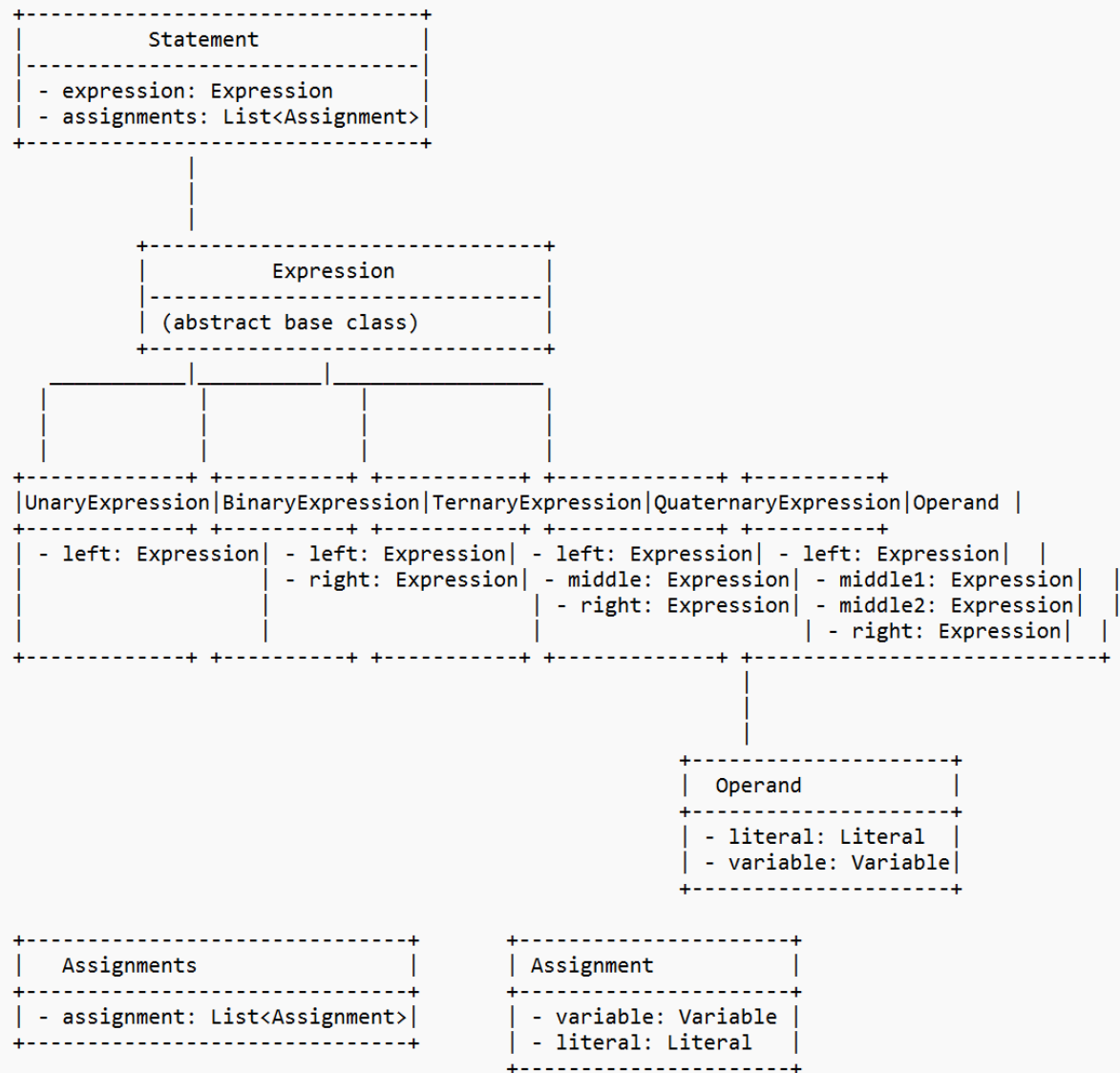
CMSC 330: Advanced Programming Languages

10/6/2024

### UML Diagrams:

Note: The Main Method is in Project2 Class. This allows it to run the program smoothly.

My own UML Diagram for the grammar given:

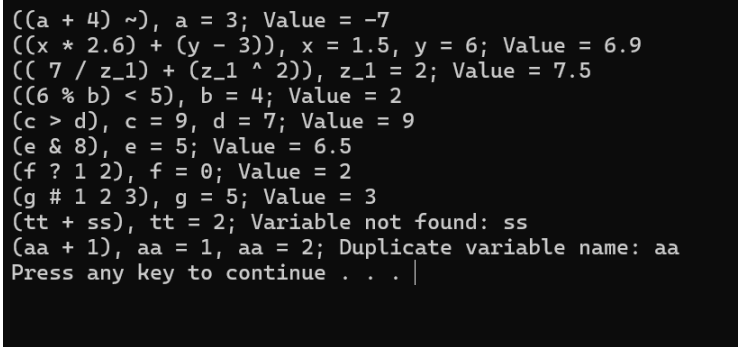


### Developer's Guide, Test Cases, and Lessons Learned:

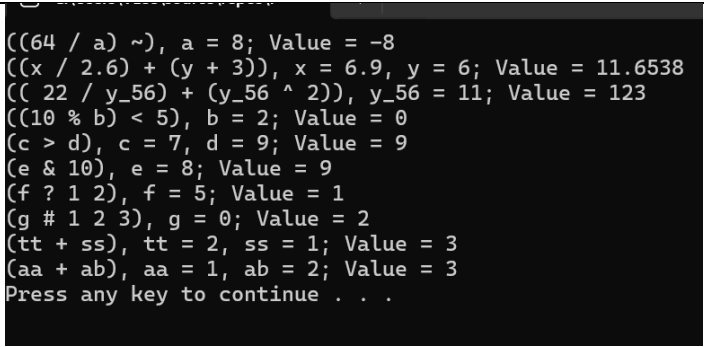
You can import the files using the new project and import all of the classes. Make sure to have a project called "Project2\_Skeleton1" and all the ".h" and ".cpp" are in the "src" file. An alternative way is to create the project and import all the files included from the ".ZIP" file. You can compile the file and execute the program by going to the "project2" class, right-clicking on the class, and selecting the run option. To import and read each file, go to the "project2" class and scroll for the code where it has the input file name. Change the name to the files that I uploaded. The names could be: "input.txt", "input2.txt", "input3.txt", "input4.txt", "input5.txt", and "input6.txt" It should pop up the expressions, equations, calculations, or errors from that file.

**Table 1 (below):** *Developer's guide describing compiling and executing the program.*

*Documentation includes Lessons learned at the end.*

Test #	Description	Screenshot	PASS / FAIL Flag
1.	<p>This is Testing the Given "input.txt" file.</p> <p>This should test all the functions and make sure that if the variable is not found let the user know. It would also notify if there is a duplicate variable name. The unary expression should be negative after computing. It should be able to calculate multiple variables (more</p>	 <pre> ((a + 4) ~), a = 3; Value = -7 ((x * 2.6) + (y - 3)), x = 1.5, y = 6; Value = 6.9 (( 7 / z_1) + (z_1 ^ 2)), z_1 = 2; Value = 7.5 ((6 % b) &lt; 5), b = 4; Value = 2 (c &gt; d), c = 9, d = 7; Value = 9 (e &amp; 8), e = 5; Value = 6.5 (f ? 1 2), f = 0; Value = 2 (g # 1 2 3), g = 5; Value = 3 (tt + ss), tt = 2; Variable not found: ss (aa + 1), aa = 1, aa = 2; Duplicate variable name: aa Press any key to continue . . .   </pre>	PASS

	<p>than 1). It should calculate the expression with variables such as the ones with underscore. It should be able to use the modulo or remainder function to determine the outcome while comparing the minimum number. It should do the same when comparing with the maximum. It should compute the average of two numbers using the &amp; subexpression. For the ternary expression, it should pop up the second number if it is equal to 0. For the quaternary expression, it should pop up the third number if the number is greater than 0.</p> <p>The text file has:</p> <p><math>((a + 4) \sim), a = 3;</math></p> <p><math>((x * 2.6) + (y - 3)), x = 1.5, y = 6;</math></p> <p><math>((7 / z\_1) + (z\_1^2)), z\_1 = 2;</math></p> <p><math>((6 \% b) &lt; 5), b = 4;</math></p>		
--	--	--	--

	<p>(c &gt; d), c = 9, d = 7;</p> <p>(e &amp; 8), e = 5;</p> <p>(f ? 1 2), f = 0;</p> <p>(g # 1 2 3), g = 5;</p> <p>(tt + ss), tt = 2;</p> <p>(aa + 1), aa = 1, aa = 2;</p>		
2.	<p>This is Testing the Given "input2.txt" file.</p> <p>This should test all the functions. The unary expression should put a negative after the computing. It should be able to calculate multiple variables (more than 1). It should be able to calculate the expression even with underscored variables changed like y_56. It should be able to use the modulo or remainder function to determine the outcome while comparing the minimum number. It should do the same when comparing with the maximum. It</p>	 <pre> ((64 / a) ~), a = 8; Value = -8 ((x / 2.6) + (y + 3)), x = 6.9, y = 6; Value = 11.6538 (( 22 / y_56) + (y_56 ^ 2)), y_56 = 11; Value = 123 ((10 % b) &lt; 5), b = 2; Value = 0 (c &gt; d), c = 7, d = 9; Value = 9 (e &amp; 10), e = 8; Value = 9 (f ? 1 2), f = 5; Value = 1 (g # 1 2 3), g = 0; Value = 2 (tt + ss), tt = 2, ss = 1; Value = 3 (aa + ab), aa = 1, ab = 2; Value = 3 Press any key to continue . . . </pre>	PASS

	<p>should compute the average of two numbers using the &amp; subexpression. For the ternary expression, it should pop up the first number if it is not equal to 0. For the quaternary expression, it should pop up the second number if the number is equal to 0. It should calculate the expression even though one variable contains similar letters. Ex: variable 1 has aa and variable 2 is ab. Despite their similarity with the letter a, since ab is together and not the same as aa it should not claim it is a duplicate variable.</p> <p>The text file has:</p> <p><math>((64 / a) \sim), a = 8;</math></p> <p><math>((x / 2.6) + (y + 3)), x = 6.9, y = 6;</math></p> <p><math>((22 / y_{56}) + (y_{56}^2)), y_{56} = 11;</math></p> <p><math>((10 \% b) &lt; 5), b = 2;</math></p>		
--	--	--	--

	<p>(c &gt; d), c = 7, d = 9;</p> <p>(e &amp; 10), e = 8;</p> <p>(f ? 1 2), f = 5;</p> <p>(g # 1 2 3), g = 0;</p> <p>(tt + ss), tt = 2, ss = 1;</p> <p>(aa + ab), aa = 1, ab = 2;</p>		
3.	<p>This is Testing the Given “input3.txt” file.</p> <p>This should test all the functions. The unary expression should be negative after computing from the modulo. It should be able to calculate multiple variables (more than 1). It should be able to calculate the expression even with underscored variables changed like z_1 and z_2. It should be able to use the modulo or remainder function to determine the outcome while comparing the minimum number. It should do the same when comparing with</p>	<pre>((a % 4) ~), a = 3; Value = -3 ((x / 2.6) + (y * 3)), x = 1.5, y = 6; Value = 18.5769 (( 7 / z_2) + (z_1 ^ 2)), z_1 = 2, z_2 = 7; Value = 5 ((6 % b) &lt; 4), b = 4; Value = 2 (c &gt; d), c = 80, d = 70; Value = 80 (e &amp; 10), e = 5; Value = 7.5 (f ? 1 2), f = 0; Value = 2 (g # 1 2 3), g = -3; Value = 1 (tt - z1), tt = 2, z1 = 5; Value = -3 ((aa + 1) + (ba + 1)), aa = 1, ba = 2; Value = 5 Press any key to continue . . .  </pre>	PASS

	<p>the maximum. It should compute the average of two numbers using the &amp; subexpression. For the ternary expression, it should pop up the second number if it is equal to 0. For the quaternary expression, it should pop up the first number if the number is less than 0. It should calculate the expression including negative results even though one variable contains similar letters. IT should compute multiple variables where one is all in letters and the other is a letter and a number. Ex: variable 1 is tt and variable 2 is z1. Or Variable 1 is aa and variable 2 is ba. Since these are different and may be similar, since they are not the same, it should not count as duplicate variables.</p> <p>The text file has:</p> <p>((a % 4) ~), a = 3;</p>		
--	---	--	--



	$((x / 2.6) + (y * 3)), x = 1.5, y = 6;$  $((7 / z\_2) + (z\_1^2)), z\_1 = 2, z\_2 = 7;$  $((6 \% b) < 4), b = 4;$  $(c > d), c = 80, d = 70;$  $(e \& 10), e = 5;$  $(f ? 1\ 2), f = 0;$  $(g \# 1\ 2\ 3), g = -3;$  $(tt - z1), tt = 2, z1 = 5;$  $((aa + 1) + (ba + 1)), aa = 1, ba = 2;$		
4.	<p>This is Testing the Given “input4.txt” file.</p> <p>This should test all the functions. The unary expression should be negative after computing from the modulo. It should be able to calculate multiple variables (more than 1). It should be able to calculate the expression even with underscored variables changed like z_1 and z_2. IT should be able to calculate the</p>	<pre> ((a - 4) ~), a = 6; Value = -2 ((x / 2.6) + (y1 * 6)), x = 1.5, y1 = 6; Value = 36.5769 ((7 / z_2) + (z_1 ^ 2)), z_1 = 2, z_2 = 7; Value = 5 ((6 % x) &lt; 4), x = 4; Value = 2 (c &gt; d), c = 60, d = 20; Value = 60 (e &amp; 10), e = -20; Value = -5 (f ? 1 2), f = 22; Value = 1 (g # 1 2 3), g = 33; Value = 3 (tt - a), tt = 2, a = 5; Value = -3 ((aa + 1) + (ba + 1)); Uninitialized variable or variable not found: aa Press any key to continue . . . </pre>	PASS

	<p>expression even though the variable letter was the same. This should erase the previous variable and put it in a new number when called by the same variable. For instance, <math>x = 1.5</math> is called in the first expression, then when <math>x = 4</math> is called in a second expression, erase the value from the first and then use the value for the second with that variable letter. It should be able to compute even though it's the same variable condition. It should be able to use the modulo or remainder function to determine the outcome while comparing the minimum number. It should do the same when comparing with the maximum. It should compute the average of two numbers using the &amp; subexpression and output a negative number if it starts off with a negative</p>		
--	---	--	--

	<p>number. For instance, -20. For the ternary expression, it should pop up the first number if it is not equal to 0. For the quaternary expression, it should pop up the third number if the number is greater than 0. It should calculate the expression including negative results even though one variable contains similar letters. It should compute multiple variables where the variable is called again repeatedly. For instance, in expression 1 <math>a=6</math> and in expression 2 <math>a = 5</math>. It should be able to not use the value from expression 1 when <math>a</math> is called from expression 2 as previously mentioned. This allows that you can use the same variable name for different expressions, but not in the expression itself. For example, you cannot use the <math>x = 1</math> and <math>x = 3</math> for the</p>		
--	---	--	--

	<p>expression, but it can be called in other expression list. It should also print out an error that the variable is uninitialized or not found if the user inputs only the equation but no variables.</p> <p>The text file has:</p> <p><math>((a - 4) \sim), a = 6;</math></p> <p><math>((x / 2.6) + (y1 * 6)), x = 1.5, y1 = 6;</math></p> <p><math>((7 / z\_2) + (z\_1 ^ 2)), z\_1 = 2, z\_2 = 7;</math></p> <p><math>((6 \% x) &lt; 4), x = 4;</math></p> <p><math>(c &gt; d), c = 60, d = 20;</math></p> <p><math>(e \&amp; 10), e = -20;</math></p> <p><math>(f ? 1\ 2), f = 22;</math></p> <p><math>(g \# 1\ 2\ 3), g = 33;</math></p> <p><math>(tt - a), tt = 2, a = 5;</math></p> <p><math>((aa + 1) + (ba + 1));</math></p>		
--	---	--	--

5.	<p>This is Testing the Given “input5.txt” file.</p> <p>This should test all the functions except ternary and quaternary expressions. IT should compute all binary expressions, addition, minus, division, in multiple ways, multiplication, modulo with unary, and average with other binary operations. It should be able to compute the calculation and then compare it with the min and max. It should also be able to compute the exponents. This is written with added spacing and non-spacing to make sure that no spacing interferes with the parsing and symbols.</p> <p>The text file has:</p> <p>(x+y), x = 10, y= 2;</p> <p>(x-y), x=0, y=0;</p> <p>((x+y)/(x-y)), x=8, y=6;</p>	<pre>(x+y), x = 10, y= 2; Value = 12 (x-y), x=0, y=0; Value = 0 ((x+y)/(x-y)), x=8, y=6; Value = 7 ((x/y)/(x-y)), x=14, y=4; Value = 0.35 (x/y), x=35, y=5; Value = 7 (x*y), x=12, y=11; Value = 132 ((x*y)~), x = 3, y = 4; Value = -3 ((x-y)&amp;(6+y)), x = 1, y = 2; Value = 3.5 ((x-3) &lt; (y+1)), x = 2, y = 5; Value = -1 ((x-3) &gt; (y+1)), x = 2, y = 5; Value = 6 ((x*2) ^ 2), x = 2; Value = 16 Press any key to continue . . .</pre>	PASS
----	---	--	------

	$((x/y)/(x-y))$ , $x=14, y=4$ ;  $(x/y)$ , $x=35, y=5$ ;  $(x*y)$ , $x=12$ , $y=11$ ;  $((x\%y)\sim)$ , $x = 3, y = 4$ ;  $((x-y)\&(6+y))$ , $x = 1, y = 2$ ;  $((x-3) < (y+1))$ , $x = 2, y = 5$ ;  $((x-3) > (y+1))$ , $x = 2, y = 5$ ;  $((x*2) ^ 2)$ , $x = 2$ ;		
6.	<p>This is Testing the Given “input6.txt” file.</p> <p>This should test all the unary, ternary, and quaternary expressions. It should compute the binary expression for the unary negation. For the unary, it should put a negation after the calculation. Since it is 3.5 it should be -3.5 For the ternary expression, it should pop up the first number if it is not equal to 0. For instance, <math>x = -1</math> and <math>x = 1</math> are</p>	<pre> (((x-y)&amp;(6+y))~), x = 1, y = 2; Value = -3.5 (x ? 100 200), x = -1; Value = 100 (x ? 100 200), x = 1; Value = 100 (x ? 100 200), x = 0; Value = 200 (y # 100 200 300), y = -1; Value = 100 (y # 100 200 300), y = 0; Value = 200 (y # 100 200 300), y = 1; Value = 300 Press any key to continue . . .   </pre>	PASS

	<p>values that do not equal to 0 then, it should give out 100. For the ternary expression, it should pop up the second number if it is equal to 0. For instance, <math>x = 0</math> pops up the number 200.</p> <p>For the quaternary expression, it should pop up the third number if the number is less than 0. For instance, <math>y = -1</math> pops up 100.</p> <p>For the quaternary expression, it should pop up the third number if the number is equal to 0. For instance, <math>y = 0</math> pops up 200.</p> <p>For the quaternary expression, it should pop up the third number if the number is greater than 0. For instance, <math>y = 1</math> pops up 300.</p> <p>The text file has:</p> <p><math>((x-y) \&amp; (6+y)) \sim</math>, <math>x = 1, y = 2</math>;</p>		
--	--	--	--

	<p>(x ? 100 200), x = -1;</p> <p>(x ? 100 200), x = 1;</p> <p>(x ? 100 200), x = 0;</p> <p>(y # 100 200 300), y = -1;</p> <p>(y # 100 200 300), y = 0;</p> <p>(y # 100 200 300), y = 1;</p>		
7.	<p>This is Testing the Given “input7.txt” file.</p> <p>This should test all the unary, ternary, and quaternary expressions. It should compute the binary expression for the unary negation. For the unary, it should put a negation after the calculation. For the ternary expression, it should pop up the first number if it is not equal to 0. For instance, x = -10 and x =10 are values that do not equal to 0 then, it should give out 100. For the ternary</p>	<pre>((x+y)&amp;(6-y))~, x = 1, y = 2; Value = -3.5 (x ? 100 200), x = -10; Value = 100 (x ? 100 200), x = 10; Value = 100 (x ? 100 200), x = 0; Value = 200 (y # 100 200 300), y = -10; Value = 100 (y # 100 200 300), 1 = 0; Uninitialized variable or variable not found: y (y # 100 200 300), \$ = 10; Uninitialized variable or variable not found: y Press any key to continue . . .  </pre>	PASS



	<p>expression, it should pop up the second number if it is equal to 0. For instance, <math>x = 0</math> pops up the number 200.</p> <p>For the quaternary expression, it should pop up the third number if the number is less than 0. For instance, <math>y = -10</math> pops up 100.</p> <p>For the quaternary expression, it should pop up an error that a variable must have a starting letter, not a number. It throws a message that it should be the letter y, not 1.</p> <p>For the quaternary expression, it should pop up an error that a variable must have a starting letter, not a symbol. It throws a message that it should be the letter y, not \$.</p> <p>The text file has:</p> <p><math>((x+y)\&amp;(6-y))\sim</math>, <math>x = 1, y = 2</math>;</p>		
--	---	--	--

<pre> (x ? 100 200), x = -10;  (x ? 100 200), x = 10;  (x ? 100 200), x = 0;  (y # 100 200 300), y = -10;  (y # 100 200 300), 1 = 0;  (y # 100 200 300), \$ = 10; </pre>		
--	--	--

### Lessons Learned (paragraphs):

To achieve my project goals, I learned how to create a project, classes, immutable classes, try/catch, throw exceptions, encapsulation, inheritance, information hiding, polymorphism, methods, and functions for the Calculation Expressions program. I learned that the Calculation Expressions program allowed me to comprehend the significant principles of tokens, parsing, symbols, and software development. A class defines all the attributes an object can have and methods that define the object's functionality. A subclass inherits the properties and behaviors of another class. So, the main lesson learned from the Project1 class is that it is responsible for creating the Calculation Expressions program and calculating the expressions based on the "input.txt" file. Moreover, the additional classes I had to create and implement are Average, Maximum, Minimum, Multiply, Divide, Exponent, Modulo, UnaryExpression, TernaryExpression, and QuaternaryExpression. The main goals are to Expand the C++ program to parse and evaluate additional expression types, modify variable and literal tokens to allow underscores and floating-point numbers and Improve error handling for uninitialized and

multiple variable assignments. I learned that a deep understanding of the grammar rules is crucial for accurately parsing expressions. I noticed and learned that recursive descent parsing is a suitable approach for parsing expressions defined by context-free grammar. I had to find a way to implement robust error-handling mechanisms are essential to catch invalid expressions and provide meaningful error messages. I also had to learn adhering to the correct operator precedence is vital for accurate evaluation. I utilized short-circuit evaluation to improve performance for certain expressions. It took me a long time to comprehend the expressions of unary, ternary, and quaternary operators are necessary for correct evaluation. I had to ensure the symbol table is initialized before each statement evaluation to prevent unintended variable reuse. Then I went back and implemented checks for uninitialized and multiply assigned variables. I had to learn to organize code into separate classes and use inline functions for small, frequently used functions. This includes creating test trials for verifying the correctness of the implementation. As for errors, I provide the information after compiling and running it. In real life, users can utilize this application as a calculator and understand Parser, Symbols, Expressions, and Tokens. It will also allow the user to have more hands-on for building a calculator. I think that Wolfram Alpha probably uses a similar way when implementing the math calculator online to utilize. I learned that the two restrictions are that the grammar cannot have any left recursive productions and must not require more than one token look ahead. A recursive descent parser is a top-down parser that parses any given string using recursive functions or procedures (GeeksforGeeks, 2023, June 9). A top-down parser builds the parse tree from the top and then down which is the start non-terminal (GeeksforGeeks, 2023, June 9). Through this parser, it removes the left recursion and left factoring from it. The resulting grammar is a grammar that is parsed by a recursive descent parser. According to Ben-Ari (2006)

and Jarc (2018), I learned that C++ has a call by value, pointers, and references used for passing by reference, constant mode, and C++ allows both pointers and the values pointed to be constant. The benefits of C++ are that I can decide when to copy or share data, and pointers avoid unnecessary duplication. The disadvantages are null pointers, memory leaks, and the need to free the memory manually. For C++, it offers both "pass-by-value" and "pass-by-reference" explicitly, allowing developers to choose the most suitable method based on the situation. For C++ it allows separation, but it is not the default because the header files contain the specification, and the source files contain the implementation. Interfaces can be defined in C++ by creating a class with all pure virtual functions and no data. C++ also allows stack types. Multiple inheritance in C++ introduces complexities such as the diamond problem, where a class inherits from two classes that both inherit from a common base class, potentially leading to ambiguity and redundancy. C++ deals with these issues using virtual inheritance, which ensures that the base class is only included once, preventing multiple copies of it in the derived class hierarchy. I learned to comprehend operator precedence and associativity rules are crucial for correct expression evaluation. I noticed that the side effects in expressions can affect their evaluation order and results. I learned that pure expressions are expressions without side effects. I noticed in my lessons in module 3 that different languages have variations in arithmetic, relational, logical, and bitwise operators. The syntax of control statements varies among languages, including fully bracketed syntax, selection statements, and iterative statements. Selection statements include if statements and case/switch statements. Iterative statements include for loops and while/do-while loops. Structured programming emphasizes the use of nested statements and avoids unstructured constructs like goto statements. I noticed that subprograms are essential for modularizing code and promoting code reusability. The parameter

passing mechanisms vary among languages, including pass-by-value, pass-by-reference, and call-by-value-result. And the return values allow subprograms to provide results to the calling context. Expressions are parsed and evaluated following the rules of operator precedence and associativity. Expressions can contain literal values, variables, and nested expressions. The SubExpression class represents binary arithmetic operations. The Operand class represents literal values and variables. Variables are assigned values using the assignment operator (=). The parseAssignments function parses and inserts assignments into the symbol table. The main function reads the input expression, parses it into an expression tree, evaluates the expression, and prints the result. The SymbolTable class stores variables and their corresponding values. The parseName function parses variable names from the input. The evaluation methods in the Expression subclasses calculate the values of expressions.(Ben-Ari, 2006; Jarc, 2018). Some of the improvements that could be made are I should have debugged more, to make sure everything is working fine. I should have explored techniques like compiler optimizations to improve performance for complex expressions, implemented more comprehensive syntax error handling to provide better feedback to users, designed the code to be easily extensible to support additional expression types or features in the future, and adhered to consistent coding standards and formatting conventions to enhance readability and maintainability. I could have also improved the performance by creating an efficient data structure for the symbol table, such as a hash table, but I chose not to do it. Overall, I learned to apply it to Project 2 with the lessons about classes, subclasses, packages, headers, importing libraries, constructors, encapsulation, inheritance, interfaces, information hiding, and polymorphism.

My design approach was to create all the required classes before implementing the additional classes I was to include. I started with a Bottom-Up Design when building the code,

but then debugged the code through a Top-Down Design. I followed the instructions on what is asked for in Project 2. I utilized the lessons to apply them to the other classes that were implemented. Once it was finished, I went back into the other classes to create the expressions and binary subexpressions according to the rubric, and then the user inputs would be passed and it should pass back through the project2 class. I had to learn new coding in C++ through module 3 of week 5 and had to enhance my knowledge of this concept. The project was not difficult to code, but it was much harder to interpret it before coding it which is why it took me longer than I thought it would. The main class creates a Project2 program allowing the “input.txt” file to provide the expressions, calculate it, and evaluate it based on the input on that file. If it is an invalid input, it will tell the user in the error section of the console. To debug the additional classes, I looked at the lessons, my old codes, and online concepts about this chapter. I then modified the classes. Then, I checked back to see if the output was correct through the classes.

## References

- Ben-Ari, M. (2006). *Understanding Programming Languages*. John Wiley & Sons, Chichester and Weizmann Institute of Science. <https://leocontent.umgc.edu/content/dam/course-content/tus/cmssc/cmssc-330/document/upl.pdf?ou=1277895>
- GeeksforGeeks. (2023, April 6). *Passing By Pointer vs Passing By Reference in C++*. GeeksforGeeks. [https://www.geeksforgeeks.org/passing-by-pointer-vs-passing-by-reference-in-cpp/?ref=oin\\_asr2](https://www.geeksforgeeks.org/passing-by-pointer-vs-passing-by-reference-in-cpp/?ref=oin_asr2)
- GeeksforGeeks. (2023, June 9). *Recursive Descent Parser*. GeeksforGeeks. <https://www.geeksforgeeks.org/recursive-descent-parser/>
- Jarc, D. J. (2018). *CMSC 330: (Week 2) [PowerPoint slides]*. University of Maryland Global Campus. <https://leocontent.umgc.edu/content/dam/permalink/flc30adc-2624-4d1d-90fe-bf7a35045fa2.html?ou=1277895>
- Jarc, D. J. (2018). *CMSC 330: (Week 6) [PowerPoint slides]*. University of Maryland Global Campus, <https://leocontent.umgc.edu/content/dam/permalink/79b48b14-ef37-42bd-8934-8a175e61e216.html?ou=1277895>
- Nievergelt, J. (n.d.). *Algorithms and data structures*. University of Maryland Global Campus. [https://leocontent.umgc.edu/content/dam/course-content/tus/cmssc/cmssc-330/document/Nievergelt\\_Algorithms%20and%20Data%20Structures08%20%281%29.pdf?ou=1277895](https://leocontent.umgc.edu/content/dam/course-content/tus/cmssc/cmssc-330/document/Nievergelt_Algorithms%20and%20Data%20Structures08%20%281%29.pdf?ou=1277895)
- UMGC. (n.d.). *Module 1: Formal Syntax and Semantics*. UMGc Leo Content. <https://leocontent.umgc.edu/content/umuc/tus/cmssc/cmssc330/2245/modules/m1-module-1/s3-commentary.html?ou=1277895>