

Eric Olsen

Assignment:

The assignment for this project was to take and enhance a C++ program provided in the week 5 reading. The given program accepted an infix input from the console, assigned values to its variables, and returned the result of the given equation. For example, the input below would display an answer of 11:

$(x + (y + 3)), x = 2, y = 6;$

The assignment was to take this program and expand its operations to include subtraction, multiplication, and division. The assignment also required implementing a negation and ternary expression. Additionally, the assignment required that only integer values be calculated and that all expressions be read from file using a single expression per line. The updated language grammar in BNF form is below. This assignment allowed us to assume that all input to this program was syntactically correct.

BNF Grammar:

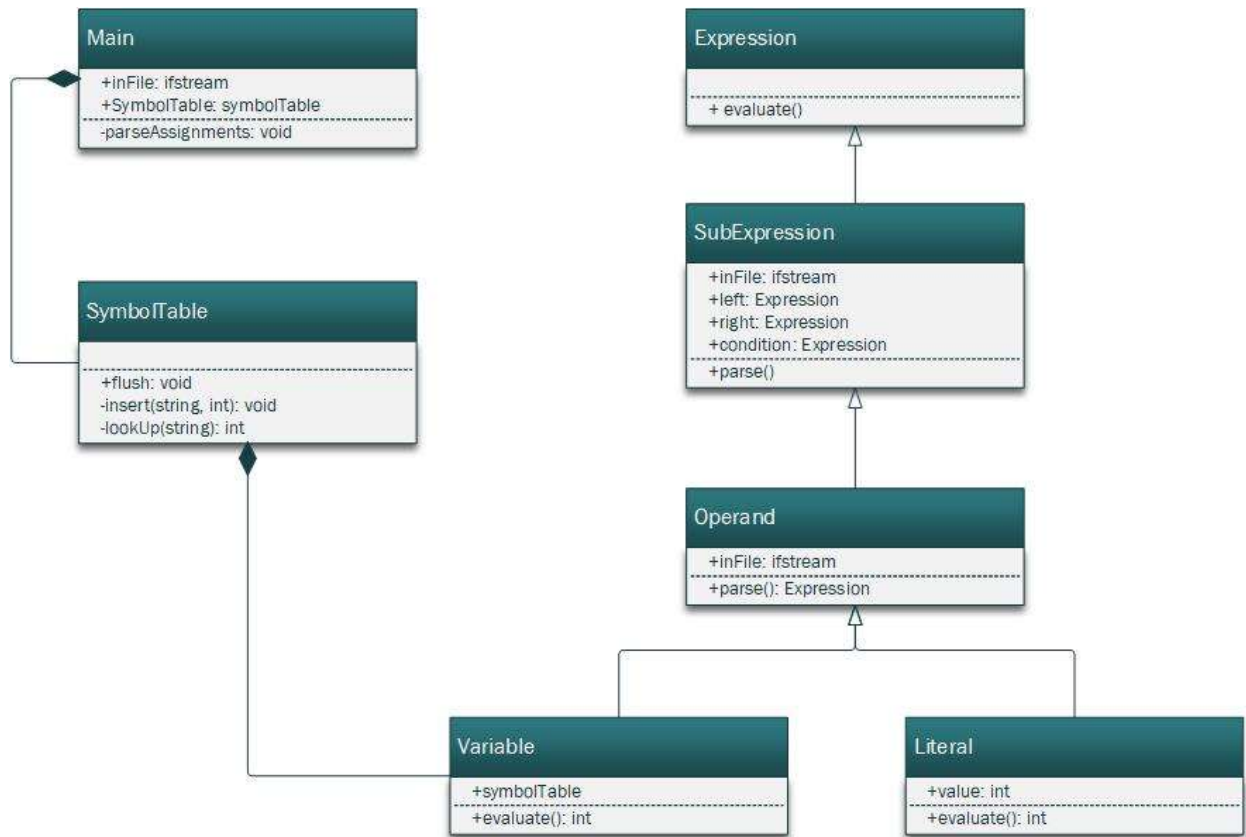
(text in red represents the added features to this program)

```
<program> → <exp> , <assigns> ;
<exp> → '(' <operand> <op> <operand> ')' | '(' <operand> ':' <operand>
      '?' <operand> ')' | '(' <operand> '!' ')'
<operand> → <literal> | <variable> | <exp>
<assigns> → <assigns> , <assign> | <assign>
<assign> → <variable> = <literal>

<op>      → '+' | '-' | '*' | '/' | '>' | '<' | '=' | '&' | '|'
<variable> → [a-zA-Z][a-zA-Z0-9]*
<literal>  → [0-9]+
```

Program Design:

Below is a UML Diagram of the overall program:



UML Observations:

The use of header files made this program seem much larger when viewed from an IDE or file-explorer; the UML diagram above depicts only seven classes. All operands can be Variables, Literals, and Subexpressions. This design feature of SubExpression preserved the recursive functionality that defined this overall language.

Code Observations:

The trend throughout this entire project was my personal discomfort programming in a new language. This was my first attempt at using the C++ language which is what defines this assignment's language under the hood. Taking the code from the reading and implementing it in my IDE was simple once I learned how to provide function declarations and proper preprocessor directives.

Duplicating the code functionality into the Minus, Times, and Divide header files was also very straightforward. The first real problem I encountered was switching the infix input away from the console and into an input-stream object. After some troubleshooting I realized that all programs referencing this stream needed to include the external reference to the input-stream object in the Main.cpp file. After declaring this variable, all instances of the 'cin' could be safely replaced with the input-stream object 'inFile.' To iterate the entire read file, it also became necessary to clear the symbol table and remove

stale data. I consequently added a 'flush' method to the symbolTable.h and .cpp files. It functioned by calling on the vector.clear() method.

Expanding the expression definition to include a ternary and negation expression was the largest task for this project. I will begin first with the negation expression. I encountered some frustrating exceptions when attempting to use an alternate constructor for a single operand. Consequently, I ended up defining the negation expression with a left operand and duplicated, unused right operand. Were this project to expand any further, I would need to address this potential point of confusion and waste of memory. When the left operand evaluated to zero, it was considered false and returned a 1 which we consider to be true. If it did not evaluate to 0, it was considered false and returned a 1. Detecting the '!' token in postfix rather than prefix was simple and even convenient to implement.

Defining the ternary expression required adding an additional constructor to the SubExpression class. This was not difficult, but it was unfamiliar to me. The ternary expression was defined by 3 internal expressions, left, right, and condition. This language differed from C-style languages and it was important to note that the left expression was the true value, right was the false value, and the third expression was the evaluated condition. When the condition's evaluate method was called, it checked to see if the values did not equal zero (was true). If so, it returned the left expression's evaluate method. If the condition equaled zero (was false), it returned the right expression's evaluate method.

The only additional note for this project was changing the numerical operations to return integer values. This was a little tedious since the program spans about twenty different files. However, this became simple once I changed the return value in the virtual function. The compiler errors guided me to every subsequent misnamed method.

Test Data:

The pages below outline the test data provided to this program. My high-level notes follow:

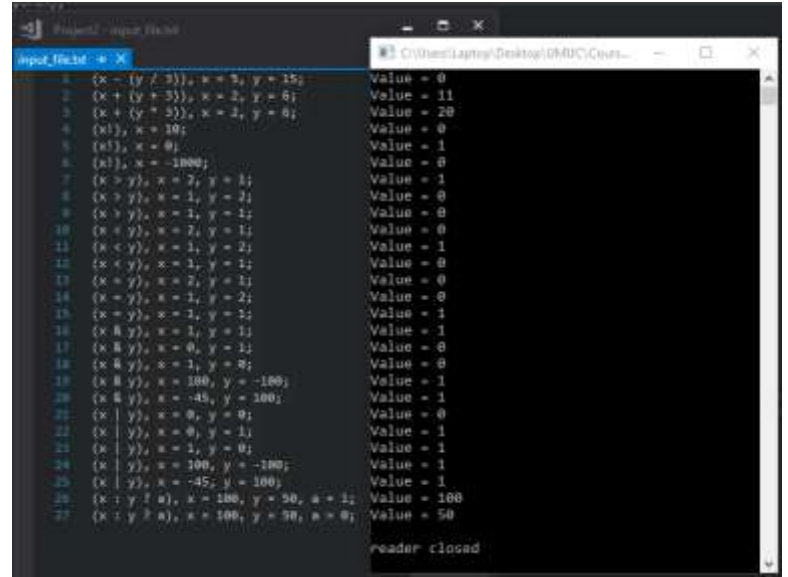
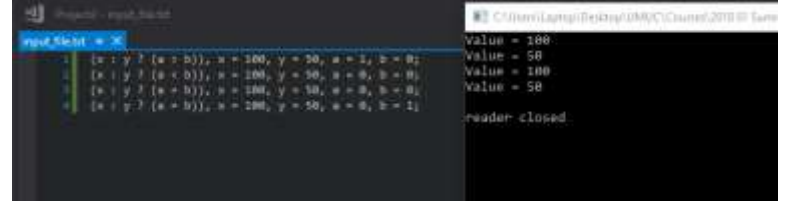
I discovered during testing that the code does not support using the same variable in the infix expression. In the tested example, I attempted to execute the following code:

```
(x : (x!) ? (a | b)), x = 100, a = 1, b = 0;
```

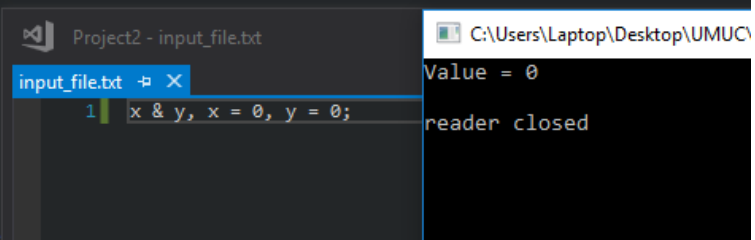
The double reference generated a null pointer exception since x was already declared. I believe that fixing this issue may be beyond the scope of this project and have left it as it is. Reference test case 3 for further details.

Additionally, in test cases 5 through 7, I tested the program response to improper input. In all cases I expected to encounter some form of an endless loop. In all three cases the code exited the loop and displayed an answer; two out of three were even correct. Consequently, if we were not able to assume proper input into this program, there would be a great deal of work to complete before publishing this program.

Project Testing

Test Case #	Input	Description	Expected Output	Actual Output	Pass / Fail	Screen shots
1	$(x - (y / 3)), x = 5, y = 15;$ $(x + (y + 3)), x = 2, y = 6;$ $(x + (y * 3)), x = 2, y = 6;$ $(x!), x = 10;$ $(x!), x = 0;$ $(x!), x = -1000;$ $(x > y), x = 2, y = 1;$ $(x > y), x = 1, y = 2;$ $(x > y), x = 1, y = 1;$ $(x < y), x = 2, y = 1;$ $(x < y), x = 1, y = 2;$ $(x < y), x = 1, y = 1;$ $(x = y), x = 2, y = 1;$ $(x = y), x = 1, y = 2;$ $(x = y), x = 1, y = 1;$ $(x \& y), x = 1, y = 1;$ $(x \& y), x = 0, y = 1;$ $(x \& y), x = 1, y = 0;$ $(x \& y), x = 100, y = -100;$ $(x \& y), x = -45, y = 100;$ $(x y), x = 0, y = 0;$ $(x y), x = 0, y = 1;$ $(x y), x = 1, y = 0;$ $(x y), x = 100, y = -100;$ $(x y), x = -45, y = 100;$ $(x : y ? a), x = 100, y = 50, a = 1;$ $(x : y ? a), x = 100, y = 50, a = 0;$	Testing the basic functionality of every operation with syntactically correct inputs. This also checks that the file read operation is functioning.	Value = 0 Value = 11 Value = 20 Value = 0 Value = 1 Value = 0 Value = 0 Value = 1 Value = 1 Value = 0 Value = 0 Value = 0 Value = 0 Value = 0 Value = 1 Value = 0 Value = 0 Value = 0 Value = 1 Value = 0 Value = 0 Value = 0 Value = 1 Value = 1 Value = 1 Value = 0 Value = 0 Value = 0 Value = 1 Value = 1 Value = 1 Value = 1 Value = 1 Value = 1 Value = 100 Value = 50	Value = 0 Value = 11 Value = 20 Value = 0 Value = 1 Value = 0 Value = 0 Value = 1 Value = 1 Value = 0 Value = 0 Value = 0 Value = 0 Value = 1 Value = 0 Value = 0 Value = 0 Value = 1 Value = 0 Value = 0 Value = 0 Value = 1 Value = 1 Value = 1 Value = 0 Value = 0 Value = 0 Value = 1 Value = 1 Value = 1 Value = 1 Value = 1 Value = 100 Value = 50	Pass	
2	$(x : y ? (a > b)), x = 100, y = 50, a = 1, b = 0;$ $(x : y ? (a < b)), x = 100, y = 50, a = 0, b = 0;$ $(x : y ? (a = b)), x = 100, y = 50, a = 0, b = 0;$	Testing condition as a nested expression	Value = 100 (true) Value = 50 (false) Value = 100 (true)	Value = 100 Value = 50 Value = 50	Pass	

	$(x : y ? (a = b)), x = 100, y = 50, a = 0, b = 1;$		Value = 50 (false)			
3	$x : (x!) ? (a \mid b)), x = 100, a = 1, b = 0;$	Negate a variable following result of ternary operation. Same infix variable declaration.	Value = 100 (true)	Null Pointer Exception. This program does not support duplicate variables in the same infix expression.	Fail	
4	$(x : (y!) ? (a \mid b)), x = 100, y = 50, a = 1, b = 0;$ $(x : (y!) ? (a \& b)), x = 100, y = 50, a = 1, b = 0;$ $((x*y) : (y!) ? (a = b)), x = 100, y = 50, a = 1, b = 1;$	Negate an operation as result of ternary operation, different variable declaration	Value = 100 (true) Value = 0 (false condition negated) Value = 5000 (true condition)	Value = 100 Value = 0 Value = 5000	Pass	
5	$(x : (y!) ? (a \mid b)), x = 100, y = 50, a = 1, b = 0$ $(x : (y!) ? (a \& b)), x = 100, y = 50, a = 1, b = 0;$ $((x*y) : (y!) ? (a = b)), x = 100, y = 50, a = 1, b = 1;$	Improper input, missing semi-colon	Endless loop	Value = 100 loop terminates (first answer correct)	N/A	
6	$(x \&\& y), x = 0, y = 0;$	Improper input, undefined operation	Endless loop	Value = 1 (correct output, very misleading)	N/A	

7	x & y, x = 0, y = 0;	Improper input, missing parentheses	Endless loop	Value = 0 (<i>incorrect output, but compiles</i>)	N/A	
---	----------------------	-------------------------------------	--------------	--	-----	--

Lessons Learned:

The obvious lesson learned from this project was the syntax of C++. Coming from a Java background I now have an improved awareness for how much differently these languages behave. Since I was under a few time constraints with my other responsibilities, I had to learn most of the C++ functionalities rather quickly on the fly as opposed to a slow and methodical introductory pace. Now that this project is complete, I am excited to visit some of C++'s introductory concepts since it's compatibility with C makes it a very powerful language.

Additionally, my previous conception of object-oriented programming revolved around Java's concept encapsulating every object into an extended language object. Since this is not the case as in C++, external references and function declarations become essential. Stumbling through this concept has broadened my understanding about how different languages accomplish this feature of object-oriented programming.