

Andrew Eissen

UMUC CMSC 330

8/8/18

Project 2: Expression Interpreter in C++

Section I: Project description

The second CMCS 330 Advanced Programming Language project requires that the author improve upon the expression interpreter written in C++ provided in the Module 3 case study and extend it to accept a user-defined text file. It must parse the included arithmetic, relational, logical, and conditional operations within the included expressions properly. The proper answer to each included expression is determined and displayed to the user in the console. The program operates by assembling an arithmetic expression tree and displaying the various operators and operations as associated tree nodes.

Section II: Deliverables

Included in the Project 2 packet is this documentation file in `.pdf` form, a plaintext `.txt` file including the test cases discussed herein, and a set of `.h` and `.cpp` files related to the program, twenty four (24) in total. Additionally, included in a separate folder are associated screenshot images featured in this document for ease of viewing. The author understands that images imbedded in Microsoft Word documents can appear grainy if not sufficiently large and has thus elected to include the screenshots themselves along with the other deliverables.

Due to the high number of files, the author has not attached or included any ancillary folders related to the running of the program in an IDE. Though the author made minor use of the C/C++ plugin for Netbeans and the Cygwin compiler obtained from the Department of Computer Science at Virginia

Tech, much of the program was written, extended, and documented in the author's copy of the Sublime Text 3 text editor by hand before being parsed, compiled, and tested in NetBeans. As such, only the relevant files are included in this project.

Section III: Development and approach

The process of developing the program into a workable entity began first with the proper installation of all components required for production. As the author has never programmed in C++ before and thus had no idea as to the specific compilers and build tools to employ, a significant amount of research was undertaken to understand this part of the process. The author eventually decided to go with a Cygwin-driven approach, downloading the necessary `gcc-core`, `gcc-g++`, `gdb`: GNU Debugger, and `'make'` tools from the Virginia Tech Department of Computer Science's Cygwin mirror and installing all relevant software. Furthermore, as the author decided the use of an IDE would assist in overcoming the initial difficulties of learning and writing in a new language, additional research was undertaken to install the best plugins essential to the creation of a C++ project series in the Netbeans IDE. This took a fair bit of effort, as difficulties were encountered in the course of building test projects that took time to overcome. However, the issues were resolved satisfactorily after several "Hello World" projects, allowing the author to begin copy/pasting the relevant Module 3 code into the assorted files.

From this point, the author began the first phase of the project, assembling the final three operator files of the ten total classes. This was done making use of inspiration provided by the extant files. The author tested the complete ten file program several times via the use of several test methods contained in the body of `module3.cpp`, though additional testing was waived until the completion of the entire program. Once this was completed satisfactorily, the author commenced the expansion part of the project, extending its functionality to include elements related to logical, relational, and conditional operators and the evaluation of multiple expressions from an input file.

As far as readability was concerned, the author went to the effort of investigating the proper manner in which inline documentation is to be added to the various files of the program and the way in which code styling is to be undertaken. Though the Google C++ styleguide suggested the standard use of `//` comments as the backbone of class documentation, other sources, such as the official Ccdoc guidelines, suggested a more Javadoc-like approach which was embraced by the author. As such, the Ccdoc of the program follows conventions similar to those previously evidenced in the author's Project 1 submission. As far as styling and indentation was concerned, the author elected to follow the Google styleguide, indenting with two spaces in most cases and keeping line widths at 80 characters, among other such restrictions. The author encourages readers to peruse the styleguide for related details.

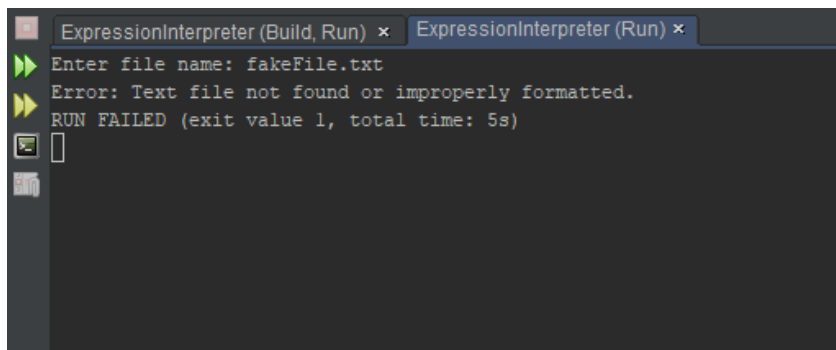
Content-wise, documentation was divided into two types depending on the nature of the file in which it was located. Documentation related to the purpose and workings of the methods was included in the header files for a "snapshot" style quick-look approach, while documentation in the `.cpp` files was more related to the changes made to those method implementations in comparison to the originals as they existed in the initial pre-extension Module 3 codebase.

Section IV: Test cases

In order to test the program's reactions to properly formatted input, the author undertook a series of test cases over the program's lifetime. As with the program development, the author attempted unit-testing over a period of several major phases as the program grew and expanded. The author tested first the additional classes considered part of the initial standard console input implementation included in the Module 3 codebase, namely `Minus.h`, `Times.h`, and `Divide.h`. Once these tests passed successfully, the author devised a test input text file containing algebraic expressions, logical comparison expressions, and negation and ternary operations. Once the project was completed, this file's test cases were employed to ensure that the program arrived at the same answers

as the author. As the rubric states that test cases and input files may be assumed to be properly formatted and arranged in accordance with the project BNF grammar, the author did not include much in the way of error-checking related to specific entries, and included only syntactically correct test cases in the file. Future updates would have likely focused on handling improper input.

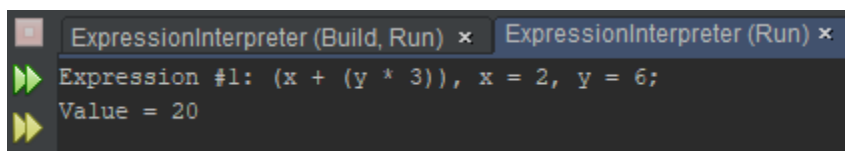
Test Case 1: Improperly named input file



```
ExpressionInterpreter (Build, Run) x ExpressionInterpreter (Run) x
>> Enter file name: fakeFile.txt
>> Error: Text file not found or improperly formatted.
>> RUN FAILED (exit value 1, total time: 5s)
```

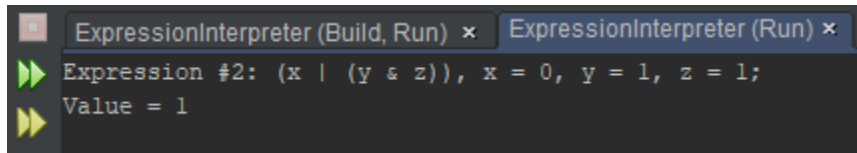
As the program gives the user the option to include the name of a file included in the folder with the program files, it is required to handle console input filenames that do not correspond to any extant files. If encountered, the program throws a warning in the console and exits, requiring the user to input a properly formatted file name.

Test Case 2: Module 3 example



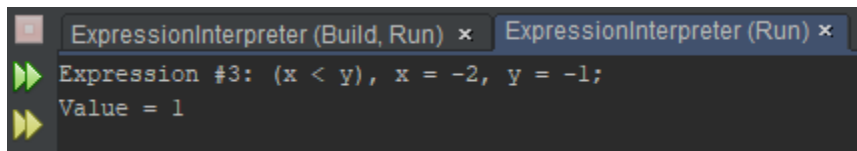
```
ExpressionInterpreter (Build, Run) x ExpressionInterpreter (Run) x
>> Expression #1: (x + (y * 3)), x = 2, y = 6;
>> Value = 20
```

The basic, default example included in the original Module 3 case study reading, namely $(x + (y * 3)), x = 2, y = 6;$, was used as the second test case, as it illustrates an example of a properly formatted algebraic expression making use of multiple different types of operator along with multiple variables. The proper answer, 20, is displayed as expected.

Test Case 3: "And" and "or"

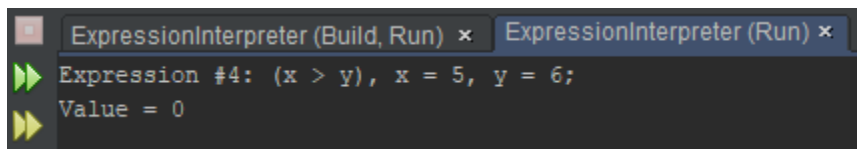
```
ExpressionInterpreter (Build, Run) × ExpressionInterpreter (Run) ×
>> Expression #2: (x | (y & z)), x = 0, y = 1, z = 1;
>> Value = 1
```

The third test cases tests the `and` and `or` operators, newly added as per the Project 2 rubric requirements. In the case of the example, `(x | (y & z)), x = 0, y = 1, z = 1;`, we can see that the inner `and` comparison between `y` and `z` will result in a value of 1 due to `1 & 1` being true. The subsequent `or` operation between `x`'s value of 0 and the parenthesized block's result of 1 will result in a final value of 1, due to at least one of the operands being true.

Test Case 4: Less than

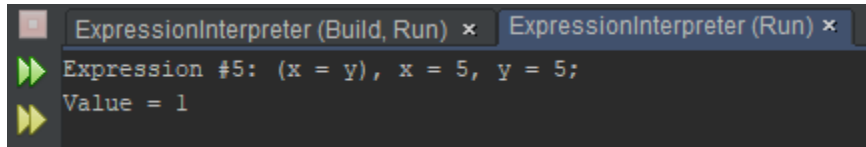
```
ExpressionInterpreter (Build, Run) × ExpressionInterpreter (Run) ×
>> Expression #3: (x < y), x = -2, y = -1;
>> Value = 1
```

The fourth test case demonstrates the result of comparison operations involving the use of the “less than” operator. In the case of the example, `(x < y), x = -2, y = -1;`, the result will be a value of 1, as it is true that -2 is less than -1.

Test Case 5: Greater than

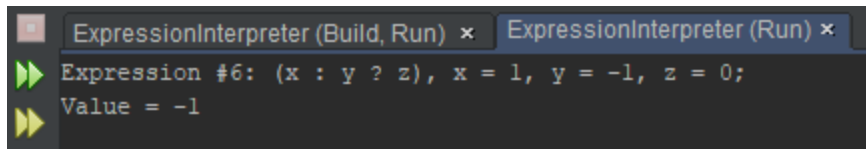
```
ExpressionInterpreter (Build, Run) × ExpressionInterpreter (Run) ×
>> Expression #4: (x > y), x = 5, y = 6;
>> Value = 0
```

The fifth test case demonstrates the result of comparison operations involving the use of the “greater than” operator. In the case of the fifth test case example, `(x > y), x = 5, y = 6;`, the result will be 0, as it is false that 5 is greater than 6.

Test Case 6: Equality

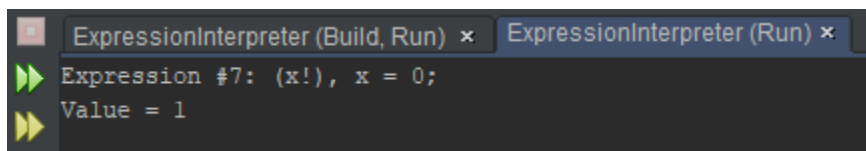
```
ExpressionInterpreter (Build, Run) x ExpressionInterpreter (Run) x
>> Expression #5: (x = y), x = 5, y = 5;
>> Value = 1
```

The sixth test cases tests the program's use of the "Equals" class to undertake equality comparison operations. The example, $(x = y), x = 5, y = 5;$, results in a true value of 1 due to the fact that $5 == 5$ is true.

Test Case 7: Ternary operation

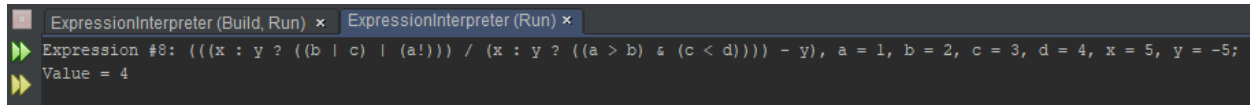
```
ExpressionInterpreter (Build, Run) x ExpressionInterpreter (Run) x
>> Expression #6: (x : y ? z), x = 1, y = -1, z = 0;
>> Value = -1
```

As notated in the Project 2 guidelines rubric, the ternary operation is undertaken slightly differently in the project grammar than is conventionally seen in the contexts of modern programming languages. In this case, the first operand is the value used if the condition is true, the second if false, and the third is the condition itself. In the case of the example, $(x : y ? z), x = 1, y = -1, z = 0;$, the condition is false, as $z = 0$, meaning that the value printed to the console is y 's value of -1.

Test Case 8: Negation

```
ExpressionInterpreter (Build, Run) x ExpressionInterpreter (Run) x
>> Expression #7: (x!), x = 0;
>> Value = 1
```

The eighth test case makes use of the negation operation to return the opposite negated value of the supplied variable. In the case of the test case example, $(x!), x = 0;$, the value printed is 1, the negation of 0. As with the ternary operation, negation is notated differently than in other languages.

Test Case 9: Supercase


The screenshot shows a window titled 'ExpressionInterpreter (Run)'. It displays the following text:

Expression #8: $((x : y ? ((b \mid c) \mid (a!))) / (x : y ? ((a > b) \& (c < d))) - y), a = 1, b = 2, c = 3, d = 4, x = 5, y = -5;$

Value = 4

The final test case, $((x : y ? ((b \mid c) \mid (a!))) / (x : y ? ((a > b) \& (c < d)))) - y, a = 1, b = 2, c = 3, d = 4, x = 5, y = -5;$, is a supercase making use of many reused variables and parenthesized subexpressions with diverse operations contained within. Beginning with the left hand side of the inner main multiplication operation, we can see that the condition of the ternary operation evaluates to true, as the first inner `or` comparison returns a 1 due to both `b` and `c` being non-zero and the outer `or` comparison evaluates to true since at least one of the operands is true (the negation of `a`'s value of 1 to 0 is irrelevant). Thus, the left hand operand of the multiplication operation is `x`, which equals 5.

On the right hand side, the condition of the ternary operation evaluates to false, as `a`'s value of 1 is not greater than `b`'s 2, regardless of whether or not `c`'s 3 is legitimately less than `d`'s 4. Both must evaluate to true in order for the `and` condition to itself evaluate to true, resulting in the selection of `y` and its value of -5 for the right hand side of the multiplication operation. Thus, as $5 / -5$ equals -1, this is the expected answer for the inner multiplication operation. Subtracting `y`'s value of -5 from this in the final operation results in a final value of 4 as $-1 - (-5) = 4$.

The test case makes sole use of single letter variables alone, with no hardcoded integers included alongside such fields. However, this was done solely to make the test case expression equation readable and presentable in the console; in other cases, the author could have constructed expressions containing multi-letter variables (provided these fields began with a legitimate letter) placed alongside any number of in-expression integers. The author's choice of format was simply a matter of preference and convenience, allowing the reader to follow along with the expression without much difficulty.

Section V: Lessons learned and potential improvements

The author has always been the sort of person to learn best by doing rather than by poring through textbooks and academic material. In the course of implementing the desired functionality, the author has come to a deepened and heightened understanding of C++ that he did not possess before, despite all the CMSC 330 class studies in the language and its distinctive approach to popular programming paradigms. However, due in large part to his unfamiliarity with the language, its syntax, and its features, the author stuck close to familiar shores in the course of expanding the program, following the lead set by the Module 3 extracts and not venturing particularly far from the examples provided therein.

If the author had possessed a significantly longer period of time in which to practice and perfect the inner and outer workings of the program, he would have liked to have studied conventional class-creation methods in C++ to create more optimized classes possessing enhanced readability. Under present circumstances, due to his general ignorance of the language, he remains unsure of whether or not his classes are of a decent, readable quality. Further, certain methods feel somewhat “spaghetti code”-ish due to this general feeling of author ignorance, possessing a lesser quality than that evidenced in the author’s Project 1 submission.

Furthermore, provided more time in which to experiment and research the language and its libraries and extensions, the author would have liked to have improved upon the user experience of the program, perhaps by making use of some sort of GUI generator similar to Java’s Swing. This would have allowed for the implementation of a user GUI including a set of buttons for file input and selection and a status log displayed the results of properly formatted expressions, as well as relevant error messages. This would have been in large part inspired by the author’s Project 1 Java submission, which made use of such a system.