Assignment 4: Evaluating Infix Expressions

Due date: November 16, 11:59PM EST.

1 Summary

In this assignment you will get practice with stacks, input text processing, and sorting. Your program will be given the name of a file on the command line and will process that file as described below, writing its output to the standard output stream. If the program is not given a file name on the command line, it will instead read its input from the standard input stream. This exactly how UNIX filters such as *cat*, *sort*, and *grep* work. It is convenient because they can be chained together in pipelines this way, or used as standalone programs.

The input file is a plain text file whose lines contain ordinary infix expressions with positive numbers as operands, and operators `+`, `-`, `*`, `/`, and `^` as well as parentheses, one expression per line. For example, a valid input line might look like

```
( 32.0*32.0 + (12.5 - 6.2)^2 + (15.625 - 8.375) ^ 2 ) ^ (1.0 / 2.0)
```

Notice in the above that the amount of white space between operators and operands can vary, and that the exponentiation operator can have non-integer exponents. Your program has to evaluate each expression and write a line of output of the form

```
33.41 = ( 32.0*32.0 + (12.5 - 6.2)^2 + (15.625 - 8.375) ^ 2 ) ^ (1.0 / 2.0)
```

for each input line. The problem is complicated just a little bit because the output must be sorted in order of increasing numeric value of the evaluated expression. In other words, if there are two expressions whose values are 10 and 20 respectively, the one whose value is 10 has to be written before the one whose value is 20.

You have a fair amount of freedom in choosing the algorithm to evaluate these expressions. *The constraint is that you must not use recursion.* One choice is to convert the expression to postfix and then evaluate the postfix. You could similarly convert to prefix and then evaluate the prefix. A third choice is an algorithm that evaluates the infix expression directly using a two-stack algorithm. Such an algorithm is described in Exercise 8 in the *Programming Problems* section of Chapter 6 of the *Walls and Mirrors* textbook. (Both the fifth and sixth editions have this exercise.)

Regardless of which algorithm you use, you may either implement your own stack or use the stack template class from the STL.

For sorting, you have the choice to implement your own sort, or use an existing sort from a library. The STL has a sorting algorithm, and the C standard library has a function called `qsort` that you may use if you like. (See the man page for `qsort` or google it online.)

2 Detailed Requirements

2.1 Input File Format

Each expression will be terminated by a newline character. There will be no unary operators, which implies that no number will be preceded by a minus sign. Negative numbers can be entered by writing `(0-x)` where `x` is a positive number. The input expressions will not contain characters other than fixed decimal numbers, the operators +, -, *, /, ^, and parentheses, and white space characters (tabs or spaces). Fixed decimal numbers may be integers or have fractional parts.

The expression may not be a proper infix expression. It may have syntax mistakes such as missing or unmatched or superfluous parentheses, missing or extra operators, and so on. Your program must catch such malformed expressions and skip over them without terminating. It does not have to identify the problem; it is enough to just skip over them.

It is convention that the exponentiation operator is right-associative, meaning that the expression `x^y^z` is interpreted as `x^(y^z)`. For this program it will be left associative, simply because it makes your job easier.

2.2 Output Format

The program must write to standard output (i.e., `cout` in C++). Each well-formed infix expression `str` should result in a line of output of the form

```
result = str
```

where `result` is the numeric value of the result and `str` is an ***identical*** copy of the infix expression. It is sufficient to use 3 digits of decimal precision for fractional values of the result. Malformed expressions should not be copied to standard output and should not cause any line to appear in standard output. Instead, these lines should be sent to the standard error stream (`cerr` in C++), with no message; the original input line should be copied to standard error. (You can separate standard error and standard output by redirecting standard error on the command line to a file using

```
command 2>errorfile
```

The output lines must be sorted in increasing order of their numeric results. Under no circumstances should they be made to fit the screen by inserting newlines. If they are too wide, they will be wrapped by the terminal.

2.3 Error Handling Requirements

The program should report if the file named on the command line does not exist or if it cannot be opened for reading. All errors in the syntax of input expressions should be handled by throwing exceptions in an appropriate way.

2.4 Design Considerations

The logic that parses the infix expression and evaluates it should be entirely contained in a *calculator* class. This class should be responsible for receiving the string that contains the expression, evaluating it, and returning its value to the calling program.

Only your main program is allowed to read input and write output. It may rely on a subsidiary class or function to do so, but *the calculator is not allowed to perform output.* The main program should read input, call the calculator, and get the result from the calculator. For each input line, it should display the line and have it evaluated by the calculator, and after all calculations have been performed, it should display the calculated values in sorted order.

I suggest that you use an `istrstream` object inside the calculator so that you can use the extraction operator to read numbers and characters easily. If you want to get really snazzy, give the class an overloaded `operator()` for the calculation, something like:

```
float operator( ) (const string & expression);
```

that evaluates the expression. Then you could write something like

```
result = calculator(inputLine);
```

in your main program. This is not a requirement, but it is certainly a nice way to code this.

Lastly, you should think about the implications of the fact that the output must be sorted and that the expressions may come from standard input. What does that imply about the way that the program will work?

3 Grading Rubric

The program will be graded based on the following rubric out of 100 points.

- A program that cannot run because it fails to compile or link on a `cslab` host loses 80%. The remaining 20% will be assessed using the rest of the rubric below.
- Meeting the requirements of the assignment: 50%
- Performance 10%
- Design (modularity and organization) 15%
- Documentation: 20%
- Style and proper naming: 5%

This implies that a program that does not compile on a lab computer cannot receive more than 20 points. Some implementations might be more efficient in terms of running time than others. There are certain programming decisions that can lead to very poor performance or to acceptable performance. The performance component of 5% will be given if the program is reasonably efficient.

4 Submitting the Assignment

This assignment is due by the end of the day (i.e. 11:59PM, EST) on November 16, 2017. There is a directory in the CSci Department network whose full path name is

`/data/biocs/b/student.accounts/cs235_sw/projects/project3`.

That is where your submission will go. In order to put it there you must follow these steps. First you will create a zip file containing your source code and nothing but your source code. To do this, create a directory named `proj3_`*`username`* where *username* is replaced by your login name, and put all of your files into that directory. Do not place anything else into this directory. ***You will lose 1 point for each file that does not belong there***. With all files in your directory, change directory so that `proj3_`*`username`* is within the current directory and run the command

```
zip -r proj3_username.zip ./proj3_username
```

You must run this command and not any other. This will compress the directory and all of the files within that directory into the file named `proj3_`*`username`*`.zip`. Then you will use the program `submit235project` to deposit the zip file into the submission directory. The program requires two arguments: the number of the assignment and the pathname of your zip file. For example, if your username on our system is `Bugs.Bunny` and your zip file is named `proj3_Bugs.Bunny.zip` and it is in your current working directory (e.g., your home directory) then you would type

```
/data/biocs/b/student.accounts/cs235_sw/bin/submit235project 3 proj3_Bugs.Bunny.zip
```

The program will create the file

`/data/biocs/b/student.accounts/cs235_sw/projects/project3/proj3_Bugs.Bunny.zip`.

You will not be able to read this file, nor will anyone else except for me. But you can verify that the command succeeded by typing the command

```
ls -l /data/biocs/b/student.accounts/cs235_sw/projects/project3
```

and making sure you see your file and that its size is the same as the size of the original `proj3_Bugs.Bunny.zip.`

If you decide to make any changes and resubmit, just run the command again and it will replace the old file with the new one. Do not try to put your file into this directory in any other way - you will be unable to do this.

If you put a solution there and then decide to change it before the deadline, just replace it by the new version. Once the deadline has passed, you cannot do this. I will grade whatever version is there at the end of the day on the due date. You cannot resubmit the program after the due date.