1 Educational Goals

- Design your own complex program
- Separate the user interface of a program from the reusable core mathematical functionality
- Discover how programming languages parse and evaluate arithmetic expressions
- Gain experience working with stacks and queues
- Learn team programming

2 Homework

Choose a partner and complete both the homework and the programming portion together. You may either agree on interfaces and divide the workload in half, or pair-program the entire project together. There will be one more team project this semester, for which you will be required to pair-program and will have to choose a different partner.

1. Evaluate the following postfix expressions by hand:
   
   (a) \[7 3 1 + -\]
   (b) \[5 100 * 4 /\]

2. Design your program. This is the first time that you’re given a functional specification without an API specification. You must choose the state, methods, and data structures on your own. Your design should include the state (member variables) of the calculator class and the signatures and contracts of the major public and helper methods, but it should provide no implementation of methods. Ignore the I/O processing and focus solely on the calculator portion in the design.

3. Why is the minus symbol always used for binary subtraction instead of unary negation in postfix?

4. I intentionally left some naturally desirable features out of the specification to make it easier to implement. Tell me what serious mathematical issues are underspecified or obvious features are lacking as a result and why implementing them would be tricky.

3 Specification

1. Input and output:
   
   (a) Print the following on program startup as a single line: username, Your Full Name, email@williams.edu where username, Your Full Name, and email are the appropriate fields for you.
   (b) Print one line of any prompt that you wish (a line ends at a newline, not where it wraps in your terminal).
   (c) Read infix, postfix, or quit, followed by a set of tokens and a newline
   (d) If the first token was quit, then terminate without printing anything
   (e) Otherwise, print Result: followed by a space and the result token, or Error: and an error message on a single line
   (f) Return to state (c)

2. Implement support for operations on number tokens using the following binary operators: addition (+), subtraction (-), floating-point multiplication (*), floating-point division (/)
3. Implement parenthesized expressions

4. Design your program as a `main.cpp` file with a `main` function that handles all I/O using the `Scanner` class provided on the course webpage, and a separate calculator class that performs no I/O.

5. On bad input, throw an exception inside of your calculator which you then catch in the I/O routines.

6. Submit your solution as described at `http://cs.williams.edu/~morgan/cs136/docs.html`

### 4 Advice

Implement this lab as two separate steps:

1. infix to postfix; and
2. postfix evaluation.

Implement these in either order, but test each in isolation before connecting them. You can find information about both on the web, as well as algorithms for them. Learning how to locate and use external resources on your own is part of the point of this lab; also, beware: don’t assume that the first resource that you find is the best one, or is even correct! For infix to postfix, use* Dijkstra’s Shunting-yard algorithm [1].

Don’t return a non-zero return code from your program: that means that an error occurred, and when grading, you don’t want me to think that an error occurred!

Abstract logical blocks of code into methods. Methods are verbs for key actions, such as “pop a number off the stack” and “is this token an operator?” If your methods are longer than 10 statements, then you probably need more helpers. If the same code appears twice (or nearly so) in your program, then you probably need more helpers.

```cpp
assert program state is consistent and throw exceptions for badly formed input, which you catch print messages for (i.e., don’t assert for input and don’t throw on internal errors).
```

Create a map of the properties of each operator and populate it during construction.

```cpp
std::queue is the standard library queue. See the following methods in particular: front(), push(), pop(), empty().
```

```cpp
std::stack is the standard library stack. See: back(), push(), pop(), empty().
```

```cpp
std::unordered_map is the standard library map. See: at(), count(). This requires that your class defines constructor of no arguments, even if you don’t use that constructor.
```

I have no `new`, `delete`, `vector`, or pointer type in my solution to this assignment. You can implement the calculator entirely with variables on the stack, references, stacks, and queues.

Even if you don’t plan to implement the bonus boolean operations, think about how you would extend your program to them and design it to make that extension easier. I have a `stack<token>`, not a `stack<double>` in my implementation.

In the interest of completing the lab quickly, you don’t have to get bogged down in detailed error reporting on bad input. That is, it is ok to throw and then print something like "stack is empty" instead of an error like "addition requires two arguments". Throwing a completely uninformative exception such as "ERROR!" is not acceptable, however.

It is much easier to handle the expected cases than the error ones in this program. Be paranoid and assume that the input is wrong at every stage.

Test your program on large amounts of input by redirecting a file into it at the command line, such as:

```
icompile --run < simple-tests.txt
```

You can use the provided test utility script to verify that the output matches the desired one.

I provided only the most basic tests to get you started and help clarify the specification. I’m going to test against a much larger battery that I specifically designed to trigger common implementation bugs. You should make lots of tests on your own to ensure that you find (and fix) bugs in your code before I do!

*I cite primary sources for algorithms. The primary source is not always the best place to find an intuitive explanation, and often a textbook like Knuth’s, course notes for other schools, or even a Wikipedia article provides a more clear one...just beware that the farther from a primary or peer-reviewed source, the more likely that what you’re reading has errors!*
5 Bonus

If you perform bonus work and your partner does not wish to, make clear in your readme.txt file who did the work.

1. Add support for infix negation (-). Tip: this can only arise in specific situations that are easy to recognize programmatically
2. Add support for boolean tokens
3. Add support for the following operators on booleans and numbers: logical AND (&&), logical OR (||), and comparisons (<, <=, >, >=, ==, !=)
4. Add support for unary logical negation (!)

References