Lab 9
Calculator

3 November 2015

The drill for this lab is the **Chapter 6 drill**. Come to lab on Tuesday either with it completed or with a specific written question in your notebook identifying which drill step you got to and what about it you're stuck on.

Notes and adjustments:

- The file it refers to is `calculator02buggy.cpp` in the shared directory. You can copy it to your own directory using

  ```
  cp /home/shared/160-3/calculator02buggy.cpp .
  ```

  (as before, don't forget the dot at the end!)

I'll be circulating around the lab to answer questions. If you're stuck on some part of the drill, ask me about that (and while you're waiting for me to get to you, look at the next section about vim help). If you're not stuck but haven't finished the drill, work on that now. If you're done with the drill, go on to the next section.

Vim FOTD: help!

Edit a file (such as your stats source code) using `vim` and type

```
:help
```

and hit enter. The session subdivides into an extra window, which has some helpful text in it. Now type

```
:help dd
```

In general, any command-mode command can be explained in this fashion. Likewise colon-mode commands ("`:help :w`") and command-line options ("`:help -o`") and even settable configuration options that you put in your vimrc ("`:help 'incsearch'`").

To close the extra help window inside of Vim, use `:q` just like you normally do to close a file—it will return the file you were editing to filling the entire PuTTY window.

It turns out that Vim is a pretty vast program, with an immense number of features. You'll keep discovering them, and any time you hear about a feature or suspect its existence, there's a good chance you can find out more about it using the `:help` system.

Comprehending the code

In the course of debugging and doing the other drill steps, you hopefully started to develop a sense of what was going on in the different parts of the program. For the rest of the lab, you'll flesh that out further, and demonstrate understanding with a mix of documentation, test cases, and added features. At this point, at least skim through to the end, and then feel free to take the parts in any order (e.g. doing the ones you find easy first!). Note that you can work on these even if you haven't found all three logic bugs yet (and working on them may even help you find the bugs).

Replace this `'8'` business

Let us recall the author's excellent advice from p. 96: Avoid magic constants! The other `char` values used in the `kind` field are not 'magic' constants, because they stand for themselves: the plus sign stands for plus signs, for instance. But the `'8'` is used in the `kind` field to stand generally for all numbers. This is reasonable, but somewhat jarring to see in the code. (To be fair to the author, he's not entirely breaking his own rule—he plans to fix this in a later chapter. We're just getting to it sooner.)

Use a `constexpr` declaration near the top of the file to give a descriptive name to this use of `'8'`, and then throughout the file, replace (this use of) `'8'` with that defined name. This shouldn't change the behaviour of the program in any way, but it is a form of self-documentation and should make the code easier to read.

Document the expression types

The program, and its underlying grammar, works by subdividing possible expressions into "expressions", "terms", and "primaries". The three functions that handle this part of the program each have brief comments (the one for `term()` says "*deal with *, /, and %*"), but these aren't as illustrative as they could be. Add comments above each of those three functions with representative examples of the different sorts of expressions that that particular function would be able to handle.

Test cases

Build a pair of files with input and corresponding expected output. Since this program loops and keeps reading as long as there's input, we don't need zillions of separate files, and can include multiple distinct test cases in the same file.

Your test cases should collectively verify all the different varieties of whole expressions that the program can handle. In addition to handling each of the operators, they should also effectively verify that order of operations is handled correctly.

Implement exponents

Although the `^` operator in C++ is used for something else, it's common in informal use (and some formal uses, e.g. some calculators) to use it to represent exponentiation, so that `4^3` would compute 4^3, for instance.

Remembering that processing of exponents comes after parentheses but before multiplication or division—at a separate level of the grammar—implement the `^` operator to raise its left argument to the power of its right argument.

Note that you'll need to add recognition of the `^` character to the `get` function. You can make use of the builtin function `pow` to actually perform the computation: for instance, `pow(4,3)` would compute 4^3.

Handing in

As usual, use the `handin` program. Designate this as `lab9`. Hand it in by 4pm on Monday, 9 November.

Rubric (tentative)

RUBRIC

1 Attendance at lab with drill done or question written down
1 Readme with suitable information
Drill
1 Compiles (5 compiler bugs found, fixed)
1 Logic bug #1 found, fixed
1 Logic bug #2 found, fixed
1 Logic bug #3 found, fixed
½ Drill items 2 and 3
½ Drill items 4 and 5
Other
½ Replace '8' marker
½ Examples of primary, term, expression
1 Test cases are comprehensive
1 Implement exponent