Project 1: Calculator

Due: 3 February 2012

This project will have you writing a compiler for a very small arithmetic language that translates programs in that language into a simplified subset of Perl. The source language has been designed to reduce or remove the need for a lot of compiler techniques that we'll cover later; the main purpose of this is to get you used to the compiler pipeline.

1 The language

This is the grammar for the language you're translating:

expr	::=	*term* `+` *term*	Addition
	|	*term* `-` *term*	Subtraction
	|	*term* *term*	Multiplication
	|	`sqrt` *term*	Square Root
	|	*term* `sq`	Square
term	::=	*number*	
	|	`(` *expr* `)`	

A *number* is a sequence of one or more base-ten digits, and is interpreted as a base-ten integer. A complete program in the source language is one or more lines, each of which contains exactly one *expr*.

All tokens are separated by whitespace.

There are a few possibly-unexpected consequences of setting up the grammar as I have; for instance, the following are all *not* valid examples of *expr*:

```
2 + 3 + 4
2
( 2 ) + ( 3 )
sqrt ( 4 )
2 + (3 + 4)
```

These, however, are valid *expr*s:

```
2 ( 4 + 5 )
( 2 + 3 ) ( 4 - 5 )
```

2 The output

The output will be a valid Perl program that performs the computations and prints out the result of each computation, one per output line. Here is everything you need to know about Perl to accomplish this:

The first line of the file should be `#!/usr/bin/perl` .

There is a `printf` function that works just as in C, C++, and Java.

There are operators `+`, `-`, and `*` that work just as in C, C++, and Java.

There is a `sqrt` function that works just as in C, C++, and Java.

There is an operator `**` that does exponentiation, e.g. `2**7` is 2^7 or 128.

3 Your program

Your program will perform all the major stages of compiling (translation), most of them fairly straightforwardly.

Lexical analysis. Since every token is separated by whitespace, you can use the default behaviour of the C++ stream input operator (`>>`) for this; since there are no identifiers in the language you needn't worry about a symbol table.

Parsing. Your parser will be a fairly simple example of a "recursive-descent" parser, and can be fairly ad-hoc; every time you go to read an expression, either it starts with `sqrt`, or a left paren, or a number. If it's a paren, you call the "parse expression" code recursively, and it will return the representation (tree) of that whole expression. If it's a number, you have a leaf node. And so on. The grammar is simple enough that you don't need any of the advanced parser techniques from later (and you don't need to write separate functions to parse *expr* and *term*).

Semantic analysis. You'll skip this; no type checking or other verification, all expressions are just numbers.

Code generation. After you have a parse tree, you'll traverse the tree and have the expression and each of its sub-expressions generate the code that would compute the corresponding expression in Perl.

4 Other notes

Your executable should accept a filename on the command line, and you should read your input program from that file and write your output to a similarly-named file that ends in `.pl` (which is the extension for Perl). Make sure to call `chmod` the output file so it is executable (`man 2 chmod` for details).

You can assume that every input file is valid. Don't worry (for this project) about *checking* the syntax; you're just parsing it to set up the syntax-driven translation.

You probably want to have at least two concrete classes, one for the internal nodes (operators) and one for the leaves, plus a superclass that they both inherit from; it'll make the code walk easier. If you're comfortable with OO design you might want separate classes for the binary operators (plus, minus, times) and the unary operators (square, square root).

5 Handing in

On torvalds, you can type

```
handin cmsc445 proj1   file1 file2 file3
```

or better, from a parent directory,

```
handin cmsc445 proj1   dirname
```

(replacing file1 and dirname with the actual locations of your stuff).

Make sure to include test cases and a readme file with information on how to compile, run, and test your code. Your test cases should actually verify that your code works; if you have failing test cases, note this in the readme (otherwise you'll lose points for the broken thing *and* for not testing your code!).