Book Assignment #1: Introduction

Issued: Thursday, September 18
Due: Tuesday, October 14

Purpose

This assignment asks you to begin thinking about programming languages and programming-language translation. These exercises are not actually in our textbook, but they ask you about material from our textbook's Chapters 1 and 2.

1. Many programming languages have string-literal tokens. Typically, a string-literal is sequence of adjacent characters surrounded/delimited by a matching pair of some kind of quotation mark. We'll use double quotes. For example: `"Hello world!"` has 12 characters.

   Of course, we need to specify which characters can appear between the delimiters, thereby composing the literal. In addition to the quote character, we need a way to include any of the 256 eight-bit characters, many of which don't exist on many keyboards. A common method is to choose an "escape" character, which precedes the character that actually belongs in the literal.

   We'll use a backslash. For example: `"Say \"hello\" to the world."` has 25 characters. A backslash can also precede a *description* of the character that belongs in the literal. Some descriptions are single characters. For example, `"Hello world!\n"` has a newline character after the exclamation mark. See the `man` page for `ASCII`, for others.

   Finally, a backslash description can be an integer between 0 and 255, specifying the eight-bit character, in one of several radixes (e.g., hexadecimal, decimal, octal, or binary). For this exercise, we'll just allow octal. For example: `"Hello\040world!"` is equivalent to our first example.

   Give a regular definition, as seen in lecture, for this language of delimited string literals. Don't worry about multibyte encoding (e.g., Unicode).

2. In lecture, we saw the following context-free grammar (CFG), for simple arithmetic expressions:

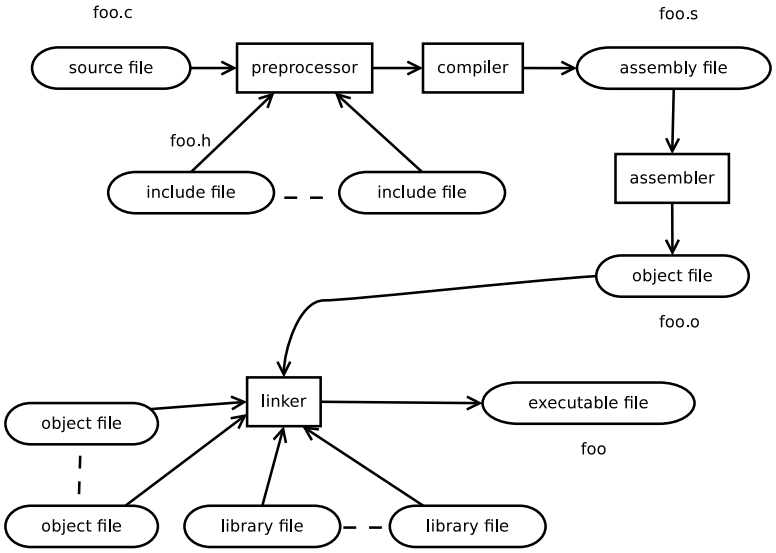
```
1  expr     : expr add_op term
2           | term

3  term     : term mult_op factor
4           | factor

5  factor   : id
6           | number
7           | '-' factor
8           | '(' expr ')'

9  add_op   : '+'
10          | '-'

11 mult_op  : '*'
12          | '/'
```

   As in lecture, draw a parse tree, and give a (textual) derivation, for each of the strings: `-x*2` and `-(x*-2)`.

3. Suppose we want to augment the arithmetic-expression CFG, from above, with arithmetic-shift operators, like those found in Java, C, and C++ (viz., `<<` and `>>`. They are binary infix operators, with low precedence. For example, `1+2<<3` evaluates to 24.

   Give two augmented CFGs, with the new shift operators. One should enforce left associativity; the other, right. Then, draw two parse trees, for a single string, demonstrating that the associativity direction for these operators is significant.

4. In lecture, we discussed a typical "toolchain" used to build C/C++ applications for Unix/Linux (e.g., a compiler, assembler, and linker):



The command lines to invoke these tools can be complex and esoteric, so they are typically recorded in some sort of specification file, for eventual execution by a tool based in some way on the original such tool: Make.

(a) What would be the symptom(s) of executing these command lines in the wrong order?

(b) How does Make determine the order in which these tool executions should occur?

(c) How could Make's algorithm be undermined, thereby causing those symptom(s)?