There are six problems labeled A–F. The first three problems are intended to help you practice basic skills with CFGs. If you feel that you need more practice, then do more examples from the textbook.

A. Exercise 2.4b (about starting and ending with the same symbol).

B. Exercise 2.4c (about odd-length strings).

C. Exercise 2.6b (about the complement of our canonical non-regular language — be careful to capture all of the strings that you need).

When CFGs are used to describe the syntax of programming languages, they are often written in a different style from our textbook CFGs. Variables are named using angle brackets, as in the start variable `<expr>` below. Rules are written using `::=` instead of $\rightarrow$. But these differences are superficial; programming-style CFGs are no more or less "powerful" than textbook CFGs. Anyway, here's a CFG for fully parenthesized algebraic expressions in a hypothetical programming language:

```
<expr> ::=  <var> | <num> | (<expr><op><expr>)
<var>  ::=  <char> | <char><var>
<char> ::=  a | ... | z | A | ... | Z | _
<num>  ::=  <dig> | <dig><num>
<dig>  ::=  0 | ... | 9
<op>   ::=  + | * | ^ | - | /
```

D. Based on the CFG above, draw the parse tree for the expression

```
((3 * (4 / temp)) - ((x + 50) * (r ^ pow)))
```

In doing so, mimic the textbook's examples in Section 2.1, and our parse tree example from today's class, with variables labeling branch nodes and terminals labeling leaf nodes. For example, the root node of your parse tree will be labeled `<expr>`. Do not take shortcuts; draw the entire tree. (For example, just the subtree corresponding to `pow` has nine nodes.)

E. Prove that every regular language is context-free, by showing that any regular expression can be converted into a context-free grammar that describes the same language. Hint: Use structural induction, as we did in converting regular expressions to NFAs.

For the final problem, let $\Sigma = \{\texttt{a}, \texttt{b}\}$. In class we developed a CFG for the language $A \subseteq \Sigma^*$ consisting of strings w that contain twice as many `a`s as `b`s. Then we converted that CFG into

a PDA. The PDA seemed to have three states, but that's because a bunch of "pushing states" were left hidden.

F. Exactly how many states did that PDA actually have? Don't draw them all out, but do give your reader an idea of how you arrived at your total.