Your name:

**Due:** Wednesday, September 25

**Instructions:** Review the course honor policy for written homeworks.

This cover sheet must be the front page of what you hand in. Fill out the left column in the table to the right after we go over each problem in class, according to the rubric below.

This rubric is also on the website, in more detail, under "Other Stuff" → "Grading Rubrics".

**Make sure all problems are submitted IN ORDER.**

- **5:** Solution is completely correct, concisely presented, and neatly written.
- **4:** The solution is mostly correct, but one or two minor details were missed, or the presentation could be better.
- **3:** The main idea is correct, but there are some significant mistakes. The presentation is somewhat sloppy or confused.
- **2:** A complete effort was made, but the result is mostly incorrect. There may be some basic misunderstandings of the topic or the problem.
- **1:** The beginning of an attempt was made, but the work is clearly incomplete.
- **0:** Not submitted.

**Comments or suggestions for the instructor:**

What other students did you work with?

**Citations** (be specific about websites):
1 Homographs and Synonyms

Pick a pair of two programming languages that you know, and come up with an example of each of the following. As always, you can work together, but everyone must turn in unique examples.

a) A **homograph** is a code fragment that is the same *syntactically* between the two languages, but has different *semantics* in each.

b) A **synonym** is a code fragment that is the same *semantically* between the two languages, but has different *syntax*.

2 Scanner DFA

C++ and Java support a few different kinds of numerical constants, or “literals”. The most basic are regular ints that you know and love like 15, 256, or 32. There are also floating-point numbers like 3.7 or .0684.

These can be described with the following regular expressions:

- **INT**: \[0 -9\]+  
- **FLOAT**: \[0 -9\]*"."\[0 -9\]*

(Remember, a * means “one or more of the previous thing”, and a * means “zero or more of the previous thing”. Square brackets [[]] indicate a range of characters (or multiple ranges), and anything in quotes "" means that literal character.)

Draw the DFA for the scanner that accepts FLOAT and INT tokens. Be sure to label each accepting state with the type of token, and put characters or character ranges on each transition.)
Besides int and float literals, C++ and Java also support two kinds of numbers that you may not know about are different bases: to start an octal (base-8) number, you put a leading zero, so 034 and 0702 are octal numbers. And to write a number in hexadecimal (base 16), you start with “0x”, like 0x3472 or 0xf3a.

Modify the DFA from the previous problem to incorporate these two new types of tokens. So all in all you should support:

Draw the DFA for the scanner that accepts the following tokens (as specified by regular expressions). Label accepting states with the name of the token.

OCTAL: 0[0-7]*
HEX: 0x[0-9a-f]+  
INT: [0-9]+  
FLOAT: [0-9]*"." [0-9]*

Observe that every octal literal like 034 could also count as an INT according to this specification. Give octal literals higher priority in your scanner, so that whenever there is a conflict, the token type will be OCTAL.
4 Modifying a Grammar

The following grammar is ambiguous (in terms of parsing), but it still defines a language:

\[
\begin{align*}
S & \rightarrow \text{exp} \\
\text{exp} & \rightarrow \text{exp} \ \text{OPA} \ \text{exp} \\
\text{exp} & \rightarrow \text{NUM}
\end{align*}
\]

Write a new grammar for the same language which is unambiguous and would work well for top-down parsing. (In particular, your new grammar should be LL(1).)

5 Top-Down Parsing

Using your modified grammar from the previous problem, draw the parse tree that would be generated by a top-down parser for the following string:

53 + 89 - 103
6 PREDICT and FOLLOW

a) Give the PREDICT and FOLLOW sets for your modified grammar from two problems ago.

b) What part(s) of the sets above would be used to create a recursive-descent LL(1) parser?