Homework 6

Due Thursday, 10/13/11, at midnight.

Please turn in your homework solutions online at `http://www.dci.pomona.edu/tools-bin/cs131upload.php` before the beginning of class on the due date.

1. (20 points) **Table-driven parsing**

   Early in the semester we discussed the fact that `if-then-else` expressions in C, C++, Java, and Pascal are ambiguous. Suppose we use the following grammar for a very simple language with conditional statements:

   ```
       <stmt> ::= if (bool) <stmt> <possElse> | assn
   <possElse> ::= else <stmt> | e
   ```

   where `e` stands for the empty string.

   In this grammar, "`if`", "`bool`", "`assn`", and "`else`" are tokens, and hence terminals of the grammar, while `<stmt>` and `<possElse>` represent non-terminals. (In a more realistic language, "`bool`" would be a non-terminal that would generate Boolean valued expressions, while "`assn`" would generate assignment statement. However, it makes the problem easier if we just let them be terminals.

   (a) Show that the string "`if (bool) if (bool) assn else assn`" has two distinct parse trees.

   (b) Compute the **first** and **follow** sets for the grammar, and build a parse table like that in Lectures 14 & 15 (see online notes).

   (c) Explain how the parse table suggests that the grammar will be problematic to parse using predictive parsing like that explained in class.

   (d) How could we manually modify the parse table so that, if used as the basis for a recursive descent or stack-based top down parser, it always chooses a parse according to the C/C++/Java/Pascal rules (i.e., always associate an "`else`" with the nearest "`if`")?

2. (10 points) **Static and Dynamic Scope**

   Please do problem 7.8 from Mitchell, page 196.

3. (10 points) **Eval and Scope**

   Please do problem 7.10 from Mitchell, page 197. You only need to do parts b and c, but please read part a so that the rest makes sense.

4. (18 points) **Lambda Calculus and Scope**

   Please do problem 7.11 from Mitchell, page 198.

5. (20 points) **Interpreters**

   The parser for PCF accepts let expressions of the form `let vble = term in body end`, but transforms them into syntax trees for function applications. In this problem I would like you to

start with a modified parser for PCF that takes let expressions like that above and parses it into an abstract syntax tree of the form `AST_LET(vble,term,body`.

In this problem you will extend the environment interpreter for PCF to interpret this term correctly.

(a) Begin by writing the computation rule using environments for let expressions. This should be similar to the rules expressed in slides 5 – 7 of Lecture 17.

(b) Extend the environment interpreter in PCFEnvinterpreter.hs (available on the web page with "Programs from Lecture") to correctly interpret terms of the form `AST_LET(vble,term,body)`. A parser that generates these `AST_LET` terms is available on that same web page. Test your program with let expressions like `let x = 2 in succ x end`.