Homework 6

Due Thursday, 10/17/13, 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 statements. 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 Lecture 8 (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. (Note that the "eval" construct discussed in this problem is in Javascript as well.)

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

   Please do problem 7.11 from Mitchell, page 198.

5. (10 points) **Type Classes**

   Type classes can be used to help write functions that are agnostic as to the implementation of data types. For example, we can write the following type class:

```
class IsStack stack where
   push:: a -> stack a -> stack a
   pop:: stack a -> stack a
   top:: stack a -> a
   isEmpty:: stack a -> Bool
```

This says that a type constructor `stack` can be in class `IsStack` if it supports polymorphic functions `push`, `pop`, `top`, and `isEmpty`.

Please define two instances of `IsStack`, one using regular Haskell lists to form the stack, and the second using the following data type definition:

```
data MyStack a = EmptyStack | MkStack a (MyStack a) deriving Show
```

For example, the code to make lists into an instance of `IsStack` begins:

```
instance isStack [] where ...
```

To test out the type class, write a Haskell function to evaluate postfix arithmetic expressions that uses the same code for any implementation of stacks. The type of the function should be:

```
eval::(IsStack stack) => [Token] -> stack Int -> Int
```

where

```
data Token = NUM Int | Plus | Minus | Mult | Div | Neg deriving Show
```

Background: Recall that an arithmetic expression is in postfix form if the operator follows the two operands. Thus (3+5)*7 in infix would be written as 3 5 + 7 * in postfix form, while 6 + (4 - 2) would be written 6 4 2 - +.

Postfix expressions can be evaluated quite simply on a stack using the following rules.

(a) A number is always pushed onto the run-time stack.

(b) When a binary operator is encountered, pop off the top two numbers on the stack and replace them by the result of applying the operator to them (but be careful to get the order correct!!), and then pushing the result back onto the stack.

(c) When a unary operator is encountered, pop off the top item on the stack, apply the operator, and then push the result back onto the stack.

(d) When the expression has been processed then there should be a single number left on the stack, and it is the answer.

Please write the arithmetic expression as a list of tokens in postfix order. Sample representations of the terms above would be `[NUM 3, NUM 5, Plus, NUM 7, Mult]` and `[NUM 6, NUM 4, NUM 2, Minus, Plus]`.

Include sample code that shows that the `eval` function works with both instances of the `IsStack` class.

6. (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. That is, the above let expression is tranlated into the syntax tree for

   ```
   ((fn vble => body) term)
   ```

   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 3 – 5 of Lecture 12.

   (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`.