Homework 2
Due Friday, 04/15/11

Please turn in your written solutions along with printouts and test runs of any programs in class. Also turn in any programs online via Blackboard. Both are due before the beginning of class on the due date.

1. (10 points) Parsing
   Please do problem 4.2 from Mitchell, page 83.

2. (5 points) Lambda Calculus Reduction
   Please do problem 4.3 from Mitchell, page 83.

3. (10 points) Symbolic Reduction
   Please do problem 4.4 from Mitchell, page 83.

4. (10 points) Lazy Evaluation and Parallelism
   Please do problem 4.11 from Mitchell, page 87.

5. (5 points) Algol 60 Pass-By-Name
   Please do problem 5.2 from Mitchell, page 122.

6. (15 points) Defining Terms in Lambda Calculus
   In class we defined Church numerals and booleans, and showed how to define more complex functions in the pure lambda calculus. Please show how to define the following functions in the pure lambda calculus.

   (a) (5 points)
   Define a function Minus such that Minus m n = m - n if m > n and 0 otherwise. Do not use recursion, but instead define it directly.

   (b) (10 points)
   Define a function LessThan such that LessThan m n = true iff m < n.

7. (10 points) Fixed Points
   Write the recursive function choose in Haskell that computes binomial coefficients using the recursion:
   \[ \text{choose} \ n \ k = \text{choose} \ (n-1) \ k + \text{choose} \ (n-1) \ (k-1) \]
   with base cases \( \text{choose} \ k \ k = 1 \) and \( \text{choose} \ n \ 0 = 1 \). You should define it by writing a \text{makeChoose} function and calling \text{fix} on it, as we did to create \text{fact}. Demonstrate that the function works correctly. Also, trace the computation that Haskell does on \text{choose} 2 1.

8. (15 points) Stack Operations
   Certain programming languages (and HP calculators) evaluate expressions using a stack. As some of you may know, PostScript is a programming language of this ilk for describing images when sending them to a printer. We are going to implement a simple evaluator for such a language. Computation is expressed as a sequence of operations, which are drawn from the following data type:
data OpCode = Push Float | Add | Mult | Sub | Div | Swap deriving Show

The operations have the following effect on the operand stack. (The top of the stack is shown on the left.)

<table>
<thead>
<tr>
<th>OpCode</th>
<th>Initial Stack</th>
<th>Resulting Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push(r)</td>
<td>...</td>
<td>r ...</td>
</tr>
<tr>
<td>Add</td>
<td>a b ...</td>
<td>(b + a) ...</td>
</tr>
<tr>
<td>Mult</td>
<td>a b ...</td>
<td>(b * a) ...</td>
</tr>
<tr>
<td>Sub</td>
<td>a b ...</td>
<td>(b - a) ...</td>
</tr>
<tr>
<td>Div</td>
<td>a b ...</td>
<td>(b / a) ...</td>
</tr>
<tr>
<td>Swap</td>
<td>a b ...</td>
<td>b a ...</td>
</tr>
</tbody>
</table>

The stack may be represented using a list for this example, although we could also define a stack data type for it.

type Stack = [Float]

Write a recursive evaluation function with the signature

`eval :: ([OpCode], Stack) -> Float`

It takes a list of operations and a stack. The function should perform each operation in order and return what is left in the top of the stack when no operations are left. For example,

`eval([Push 2.0, Push 1.0, Sub],[])` returns 1.0. The `eval` function will have the following basic form:

```haskell
eval ([], a:rest) = --
eval ((Push n):ops, rest) = --
--
eval (_, _) = error "Invalid computation";
```

You need to fill in the blanks and add cases for the other opcodes.

The last rule handles illegal cases by matching any operation list and stack not handled by the cases you write. These illegal cases include ending with an empty stack, performing addition when fewer than two elements are on the stack, and so on. You may ignore divide-by-zero errors for now.