Homework 2

Due Thursday, 9/18/14 at midnight.

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

1. (10 points) **Parsing**

   Please do problem 4.1 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) **Programming the lambda calculus**

   Because Grace has anonymous functions, we can translate the lambda calculus terms for booleans, numbers, operations, etc. into Grace. For example, $\underline{\text{true}} = \lambda u.\lambda v.u$ translates to

   ```
   def lamTrue = {u -> {v -> u}}
   ```

   Notice that in the above, `lamTrue` is just a name for the anonymous function in Grace that takes two arguments and returns the first.

   Here are terms representing, false, cond, 0, 1, and some of the operations we defined in class.

   ```
   def lamFalse = {u -> {v -> v}}
   def lamCond = {b -> {tVal -> {fVal -> b.apply(tVal).apply(fVal)}}}
   def lamZero = {s -> {z -> z}}
   def lamOne = {s -> {z -> s.apply(z)}}

   def lamSucc = {n -> {s -> {z -> s.apply(n.apply(s).apply(z))}}}
   def lamPlus = {n -> {m -> {s -> {z -> m.apply(s).apply(n.apply(s).apply(z))}}}}
   def lamMult = {n -> {m -> m.apply(lamPlus.apply(n)).apply(lamZero)}}
   def lamIsZero = {n -> n.apply{x -> lamFalse}.apply(lamTrue)}
   ```

   Because these terms are built using blocks in Grace, function application takes place using the `apply` method. Thus `lamSucc.apply(lamOne)` will return the representation of the successor of one (which we could call `lamTwo`)

   Of course we can't easily print out the answers to these computations because, for example, all of the numbers will be anonymous functions taking two arguments. However, I have written functions that can used to convert lambda encodings of numbers and booleans to actual numbers and booleans.

```
def succ = {n -> n+1}
method lamToNumber(n) -> Number {
     n.apply(succ).apply(0)
}

method lamToBool(b) -> Boolean {
      b.apply(true).apply(false)
}
```

All that `lamToNumber` does is to apply the lambda calculus representation of the number to the successor function and to zero in order to give you the number it represents. Thus you won't be surprised to know that `lamToNumber(lamSucc.apply(lamOne))` will return the number 2.

Play with these in order to make sure you understand how they work. Then write a lambda calculus term representing exponentiation. *Hint: its definition looks very similar to that for multiplication.* Translate your into Grace and show that it gives the right answers to computations such as 3^4.

Your function should be named `lamExp` and take arguments n and m that are lambda expressions corresponding to numbers such that `lamExp.apply(n).apply(m)` returns a lambda expression corresponding to n to the mth power.

5. (10 points) **Lambda Reduction with Sugar**

   Here is a "sugared" lambda-expression using `let` declarations:

$$\begin{array}{l}\texttt{let } \mathit{compose} = \lambda f.\, \lambda g.\, \lambda x.\, f(g\, x) \texttt{ in} \\ \quad \texttt{let } h = \lambda x.\, x + x \texttt{ in} \\ \quad\quad ((\mathit{compose}\, h)\, h)\, 3\end{array}$$

   The "de-sugared" lambda-expression, obtained by replacing each `let` $z = U$ `in` V by $(\lambda z.\, V)\, U$ is

$$\begin{array}{l}(\lambda \mathit{compose}. \\ \quad (\lambda h.\, ((\mathit{compose}\ h)\ h)\ 3)\ \ (\lambda x.\, x + x)) \\ \quad (\lambda f.\, \lambda g.\, \lambda x.\, f(g\ x))\end{array}$$

   This is written using the same variable names as the `let`-form in order to make it easier to compare the expressions.

   Simplify the desugared lambda expression using reduction. Write one or two sentences explaining why the simplified expression is the answer you expected.

6. (20 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. (You may use the functions defined in class in defining these new functions.)

   (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. You may assume that you are given the function Pred defined (but not explained) in class that computes the predecessor of a number (where the predecessor of 0 is 0, predecessor of 1 is 0, 2 is 1, etc.). That is, your answer may include the term Pred without providing a definition of it.

(b) (5 points)

Define a function LessThan such that LessThan m n = true iff m < n.

You might want to test your answer using the Grace code given in problem 4.

(c) (5 points)

Define the recursive fibonacci function fib such that fib 0 = 1, fib 1 = 1, and fib n = fib (n-1) + fib (n-2) for n > 1. *This is tricky as you are not allowed to name the function and use the name in the definition. You must use the Y-combinator and emulate what we did in class defining factorial.*

(d) (5 points)

Use your definition of fib from above to calculate fib 2. Do it step by step, showing all of your work. You may assume that fib 1 = 1, fib 0 = 1 and that Plus 1 1 = 2 without showing all of the reduction steps. All other steps in the reduction should be shown.

7. (10 points) **Fixed Points**

We showed in class that every function f definable in the lambda calculus has a fixed point, i.e., there is a term a such that f(a) = a. In class we defined the function Succ as the successor function. I.e., Succ n = n+1 for all encodings of integers, n. Needless to say, we don't expect the successor function to have a fixed point, yet we proved that every function has a fixed point!

Compute the fixed point of Succ. What can you say about it? In particular, why doesn't this contradict our expectations that the successor function on the natural numbers does not have a fixed point. *Yes, this is supposed to be tricky and puzzling, but you need to deal with this type of seemingly contradictory information. A hint is that you should think about what all encodings of numbers look like.*