The Art of Recursion: Problem Set 7
Due Tuesday, 30 October 2012

This problem set will allow you to explore the Haskell programming language, and use it in turn to explore some aspects of recursion.

See http://www.cis.upenn.edu/~cis194/resources.html for links to install a Haskell coding environment on your computer as well as many other Haskell-related resources. Some resources you may find particularly helpful in completing this assignment include:

- These lecture notes on Haskell basics: http://www.cis.upenn.edu/~cis194/lectures/2012-01-12.html
- Hoogle is a search engine for Haskell libraries, useful for looking up documentation on a particular function, or even search for a function by its type signature: http://www.haskell.org/hoogle/
- This cheat sheet: http://cheatsheet.codeslower.com/.

See also the Penn emacs club’s instructions on getting up and running with a Haskell coding environment in emacs (though you are in no way required to use emacs): http://emacsclub.github.com/html/haskell.html.

Throughout the rest of the assignment, you will find marginal notes hinting at library functions or Haskell syntax that you may find useful in completing the given exercise.

You should turn in a .hs or .lhs file containing your solutions via email. This file itself is a literate Haskell document, which you can load into ghci.

Recursive functions

1. Recall from the first midterm the recursive function

   \[ h(0) = 1 \]
   \[ h(2k) = h(k - 1) + h(k) \]
   \[ h(2k + 1) = h(k) \]

   Implement \( h \) directly as a Haskell function. Empirically, what is the performance of \( h \) like?

2. [Optional] What is the big-O performance of \( h \)? (The answer may surprise you!) To gain some intuition for this exercise you may find it helpful to change \( h \) to use “open recursion style” and add a “logger” as shown in last week’s class.
3. **Hyperbinary numbers** are like the usual binary numbers, except that the digit 2 allowed in addition to the usual 0 and 1. For example, the hyperbinary number 211020 represents \(2 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 2 \cdot 2^1 + 0 \cdot 2^0 = 2 \cdot 32 + 16 + 8 + 2 = 92\).

Write a Haskell function \(\text{allHB} :: \text{Integer} \rightarrow \left[\left[\text{Int}\right]\right]\) which generates all the distinct hyperbinary representations of a given number. (Assume that negative numbers have no hyperbinary representations.) It's easiest if we have the least significant hyperbinary digits come first, so that the lists are “backwards”. For example, the hyperbinary number 211020 from above (representing 92) would be stored as the list \([0,2,0,1,1,2]\).

As an example of \(\text{allHB}\) in action,

\[
\text{allHB } 4 = \left[\left[0,0,1\right], \left[0,2\right], \left[2,1\right]\right]
\]

(though it’s OK if your implementation of \(\text{allHB}\) returns these three representations in a different order).

4. **Optional** Prove that for all \(n \geq 0\), \(\text{length} \left(\text{allHB} \ n\right) = h \ n\). You may assume that \(\text{length} \left(\text{xs} ++ \text{ys}\right) = \text{length} \ \text{xs} + \text{length} \ \text{ys}\) and that \(\text{length} \left(\text{map} \ \text{f} \ \text{xs}\right) = \text{length} \ \text{xs} \text{ for all functions f and lists xs, ys (though proving these is not hard, given the definitions of length, ++ and map).}

**Structural recursion and folds**

Consider the following algebraic data type:

```haskell
data BTree a where
  Leaf :: a -> BTree a
  Branch :: BTree a -> BTree a -> BTree a
```

5. Implement a fold for \(\text{BTree}\).

6. Use your fold to implement each of the following functions:

   (a) \(\text{leaves} :: \text{BTree} \ a \rightarrow \text{Int}\), which counts the number of leaves in a tree.

   (b) \(\text{depth} :: \text{BTree} \ a \rightarrow \text{Int}\), which computes the length of the longest path from the root to any leaf. Note that \(\text{depth} \ (\text{Leaf} \ x) = 0\).

   (c) \(\text{treeMap} :: (a \rightarrow b) \rightarrow \text{BTree} \ a \rightarrow \text{BTree} \ b\), which transforms a tree full of \(a\)'s into a tree full of \(b\)'s by applying the given function in every \(\text{Leaf}\).

   (d) \(\text{treeWidth} :: \text{BTree} \ a \rightarrow \text{Int}\), which computes the width of a tree, defined as the length of the longest path between any two leaves.