CS 357: Declarative Programming
Homework 6 (Spring '14)

1. The function *bits2num* takes a string of 0's and 1's and returns the binary number represented by that string. For example,

```
*Main> bits2num "1011000"
88
```

2. The function *num2bits* takes an integer and returns a string representing that number in binary. For example,

```
*Main> num2bits 87783
"010101011011100111"
```

3. The variance of a list of numbers of length n is the average squared difference between each number and the numbers' mean: $\sum_{i=1}^{n}(x_i - \bar{x})^2/n$ where $\bar{x}$ is the numbers' mean: $\sum_{i=1}^{n} x_i/n$. Without using explicit recursion, give a definition of a function, *variance*, which works as follows:

```
*Main> variance [1..10]
8.25
```

4. Define a function *difference* which, when given two sets *xs* and *ys* (represented as lists) returns the set consisting of all elements of *xs* not found in *ys*. Note: This definition is asymmetrical. For example,

```
*Main> difference "ABCD" "AD"
"BC"
```

5. Recall that the function *combinations* takes a list of elements of typeclass *Ord* and an integer k as its arguments and returns a list of $\binom{n}{k}$ length k lists representing all possible subsets of size k. The function splits is similar, except that, given a list of elements of length n, it returns a list of $\sum_{k=1}^{n-1}\binom{n}{k}$ pairs of lists. The first component of the pair represents a combination of length k. The second component represents the complementary combination of length $n-k$. Two combinations are complementary when their union is equal to the original list. For example,

```
*Main> :t splits
splits :: (Ord a) => [a] -> [([a], [a])]
*Main> splits "abc"
[("c","ab"),("b","ac"),("bc","a"),("a","bc"),("ac","b"),("ab","c")]
```

Write *splits*.

6. The function *argmin* takes a function f and a list *xs* as arguments and returns the element of the list *x* such that f applied to *x* has minimum value. For example,

```
*Main> :t argmin
argmin :: (Ord a) => (t -> a) -> [t] -> t
*Main> argmin length ["ABC","EF","GHIJ","K"]
"K"
*Main>
```

   Write *argmin*.

7. The function *bogus* takes a list of pairs of source alphabet probabilities and values and returns a coding tree which can be used with the functions, *encode* and *decode*, for encoding and decoding Huffman coding trees defined in class. The bogus coding algorithm is very simple: it splits its list argument into two subsets where the sum of the probabilities in each subset are as nearly equal as possible. It then uses the first subset to recursively build the left half of the bogus coding tree and the second subset to recursively build the right half of the bogus coding tree. For example,

```
*Main> :t bogus
bogus :: (Ord t) => [(Double, t)] -> Htree t
*Main> let xs = [(0.30,'e'), (0.14,'h'), (0.1,'l'), (0.16,'o'),
(0.05,'p'), (0.23,'t'), (0.02,'w')]
*Main> concatMap (encode (bogus xs)) "hello"
"10100100100011"
*Main> decode (bogus xs) "10100100100011"
"hello"
```

   Hint: I found the functions *splits* and *argmin* to be very helpful. I also found that the function *merge* (defined in class) is just as good at combining bogus coding trees as it is Huffman coding trees. Indeed, given all of the above, my definition of *bogus* is just two short lines of code.

8. The function *church* takes an integer *n* as its argument and returns a function which composes any unary function *n* times. For example,

```
*Main> :t church
church :: Int -> (c -> c) -> c -> c
*Main> (church 4) tail "ABCDEFGH"
"EFGH"
```

   Write *church* using $foldr$.

9. The function *gop* (short for *generalized-outer-product*) takes a length *m* list of length *n* lists (all of the same type) and returns a length *nm* list of length *m* lists representing the *m*-fold cartesian product of elements from the lists. For example,

   ```
   *Main> :t gop
   gop :: [[a]] -> [[a]]
   *Main> gop ["AB"]
   ["A","B"]
   *Main> gop ["ABC","DEF","GHI"]
   ["ADG","ADH","ADI","AEG","AEH","AEI","AFG","AFH","AFI","BDG",
   "BDH","BDI","BEG","BEH","BEI","BFG","BFH","BFI","CDG","CDH",
   "CDI","CEG","CEH","CEI","CFG","CFH","CFI"]
   *Main>
   ```

   Write *gop*. Hint: Use a recursive nested-map.

10. Using the definition of *BTree* given in class, write a function *trees* which takes a list of leaf values as its argument and returns a list of all binary trees with the given leaves. For example,

    ```
    *Main> :t trees
    trees :: (Ord t) => [t] -> [Btree t]
    *Main> (trees "ABCDE") !! 114
    Fork (Leaf E) (Fork (Fork (Leaf A) (Fork (Leaf C)
    (Leaf B))) (Leaf D))
    *Main> length (trees [0..4])
    1680
    ```

    Hint: Define *trees* using a list-comprehension, recursion, and the function *splits*.

11. A DNA molecule is a sequence of four bases which are conventionally represented using the characters 'A', 'G', 'C', and 'T'. Genomes represented by DNA molecules are subject to four different types of point mutations:

    insertions - A base is inserted between two adjacent points in a genome.

    deletions - A point is deleted from a genome.

    substitutions - A base at a point is replaced with another base.

    transpositions - The bases at two adjacent points are exchanged.

    Give definitions for Haskell functions *insertions*, *deletions*, *substitutions* and *transpositions* which take a genome represented as a string and return a list of all genomes produced by single point point mutations of the specifed kind. For example.

```
⋆Main> insertions "GC"
insertions "GC"
["AGC","GAC","GCA","GGC","GGC","GCG","CGC","GCC","GCC",
"TGC","GTC","GCT"]
⋆Main> deletions "AGCT"
["GCT","ACT","AGT","AGC"]
⋆Main> substitutions "ACT"
["ACT","AAT","ACA","GCT","AGT","ACG","CCT","ACT","ACC",
"TCT","ATT","ACT"]
⋆Main> transpositions "GATC"
["AGTC","GTAC","GACT"]
```