CPSC 449: Haskell exercises 2

Winter 2021 (Revised January 4, 2021)

Due: Friday, February 5th (2021) at 11:59 PM midnight

For this exercise you are expected to develop *at least 6* of the programs below. The tutorials will work through (some of) the solutions. You are expected to hand in a documented Haskell script containing your solutions ... I encourage you to discuss these programs with your classmates: the aim is to complete these exercises (somehow!) so that you get used to thinking in Haskell syntax and using high-order functions. You should comment your code indicating, in particular, how you arrived at a solution. Recall that it is important that you understand the solutions as this comprehension will be tested in the tutorial written tests.

Recall, you are required to document the source of your code. You should not expect credit for code which you do not understand or to which you have not contributed. In particular, while it may help to use the functions from the Haskell's prelude (or libraries) – especially to get you going – in the end you have to fully understand the code you submit ... so in the end, to avoid losing marks (particularly for these beginning exercises) it is better to avoid prelude and library functions and to develop your own code.

Please name your functions according to what is prescribed below. If there are name conflicts with names defined in **`Prelude`**, then (a) explicitly import **`Prelude`**, and (b) use the **`hiding`** clause to hide the conflicting names when importing (see the grey box on page 53 of **[Thompson]**).

1. A logical formula in two variable is a function of the form $\texttt{f} :: (\texttt{Bool},\texttt{Bool}) -> \texttt{Bool}$. Write a function

   ```
   twoTautology:: ((Bool,Bool) -> Bool) -> Bool
   ```

   which determines whether a logical formula `f` is always true (i.e. a tautology): to be a tautology it must be true for all possible arguments of `f`)! Now write a function which determines whether two logical functions of two variables are equivalent:

   ```
   twoEquiv::((Bool,Bool)->Bool)->((Bool,Bool)->Bool)->Bool
   ```

   This should be true when the functions agree on all inputs.

2. Write a function

   ```
   badFermat :: Integer
   ```

that shows Fermat's conjecture (all numbers of the form $2^{2^n}+1$ are prime) is false by finding a number for which it is not true. What happens if the conjecture is true?

3. The function

   ```
   collatz :: Int -> Int
   ```

   is defined by $\mathsf{collatz}(n) = n/2$ if n is even and $\mathsf{collatz}(n) = 3n+1$ if n is odd. It is believed that if `collatz` is applied repeatedly, then eventually one will see the number 1. For example, `collatz(5) = 16`, and `collatz(16) = 8`, and `collatz(8) = 4`, and `collatz(4) = 2`, and `collatz(2) = 1`. The Collatz index of a number is the minimum number of times that `collatz` needs to be applied to obtain 1. For example, the Collatz index of 5 is 5, and the Collatz index of 7 is 16 (the sequence is $[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]$. Write a Haskell function

   ```
   collatzIndex :: Int -> SF [Int]
   ```

   which computes the Collatz "index" of a number by calculating the sequence ending in 1. What happens if a number has no Collatz index?

4. The bisection method is a neat way to find (approximate) roots to continuous functions – it has advantages to Newton's method because it only requires the function to be continuous instead of differentiable. Fix e to be a really small number (sometimes called machine epsilon) by making it a constant

   ```
       e :: Double
       e = exp (-150)
   ```

   We will regard any x with $|x| < e$ as being effectively zero. Let $f : \mathcal{R} \to \mathcal{R}$ be a continuous function, and a and b are numbers for which $f(a)$ and $f(b)$ have opposite signs. Then f has a zero (crosses the axis) in the interval $[a, b]$. The bisection method computes the midpoint, m, of a and b, and then compares the sign of $f(a)$, $f(b)$, $f(m)$. One of the pairs $(f(a), f(m))$ or $(f(m), f(b))$ have opposite signed numbers, or m is a root $|f(m)| < e$. Use this information to either return the root, or continue searching on a smaller interval. Write a Haskell program

   ```
   bisection::(Double->Double)->(Double,Double)->Maybe Double
   ```

   which given a function f and a pair of initial values (a, b), returns, when $f(a)$ and $f(b)$ are of opposite sign or one is effectively zero, the value (up to e!) of a zero crossing for the function f.

5. Implement bubble sort, quicksort, and merge sort in Haskell:

   ```
   bsort::(a -> a -> Bool) -> [a] -> [a]
   qsort::(a -> a -> Bool) -> [a] -> [a]
   msort::(a -> a -> Bool) -> [a] -> [a]
   ```

6. A matrix in Haskell can be represented by the type

   ```
     type Matrix a = [[a]]
     type DoubleMatrix = Matrix Double
   ```

   Write functions

   ```
   transpose:: Matrix a -> (Maybe (Matrix a))
   addMat :: DoubleMatrix -> DoubleMatrix -> (Maybe DoubleMatrix)
   multMat :: DoubleMatrix -> DoubleMatrix -> (Maybe DoubleMatrix)
   ```

   Which returns a matrix when these functions are well-defined.

7. Implement list reversal in at least two different ways from naive reverse, fast reverse, a higher-order reverse (using a fold left):

   ```
   nreverse:: [a]  -> [a]
   freverse:: [a]  -> [a]
   hreverse:: [a]  -> [a]
   ```

8. An AVL tree is a binary search tree:

   ```
   data STree a = Node (STree a) a (STree a)
                | Leaf
   ```

   consisting of a tree of items which are ordered satisfying two conditions

   (a) The value of an item at a node is larger than any item in its left tree and smaller than any item in its right tree

   (b) The tree is balanced: the difference in height between the left tree and right tree at any node is at most 1.

   Write a function to determine whether a tree is an AVL tree:

   ```
   isAVL:: (Ord a) => STree a -> Bool
   ```

9. Calculate the factorial of 1,891 or explain six different ways of programming factorial (see `http://www.willamette.edu/ fruehr/haskell/evolution.html`).

10. Given a Rose tree (`data Rose a = RS a [Rose a]`) calculate the width of the tree (that is, in the undirected graph of the tree, the length of the longest path):

    ```
    widthRose: Rose a -> Int
    ```

    Can you do this using a fold?