FP Exercise Sheet 3

October 10, 2022

Recursion

1. Write a function `isAscending :: [Int] -> Bool` which returns `True` if the given list contains elements in ascending order.
2. Write a function `myTake :: Int -> [a] -> [a]` which returns the given number of elements from a list.

   If the list is shorter than the given number of elements, return as many elements as you can. What other design decisions could you make? What are the advantages and disadvantages of each?
3. Write a function `dropOdds :: [a] -> [a]` which returns a list *without* the elements occurring in odd positions (assuming a 0-indexed list). For example, `dropOdds [1,2,3,4,5] = [1, 3, 5]`.
4. Write a function `myIntersperse :: [a] -> a -> [a]` which returns a list with the given element inserted at every other position. For example, `myIntersperse [1,2,3] 10 = [1, 10, 2, 10, 3, 10]`.
5. Write a function `myReverseRec :: [a] -> [a]` which reverses the given list. Write this using recursion.
6. Write a function `myReverseFold :: [a] -> [a]` which reverses the given list. Write this using a fold.

Higher-order Functions

1. Write a function `myMap :: (a -> b) -> [a] -> [b]` which applies the given function to each element of a list.
2. Write a function `myFoldr :: (a -> b -> b) -> b -> [a] -> b` which is a right-associative fold over the input list.
3. Write a function `myFoldl :: (b -> a -> b) -> b -> [a] -> b` which is a left-associative fold over the input list.
4. Write a function `myFilter :: (a -> Bool) -> [a] -> [a]` which retains the elements of the given input list where the predicate function returns `True`.

   For example, `myFilter (\x -> x `mod` 2 == 0) [1,2,3,4,5,6] = [2,4,6]`.

Algebraic Datatypes

1. Suppose we have the following data type:

```
data ArithExpr =
      Add ArithExpr ArithExpr
    | Sub ArithExpr ArithExpr
    | Mul ArithExpr ArithExpr
```

```
  | Div ArithExpr ArithExpr
  | Value Int
```

(a) Write a function `evalExpr :: ArithExpr -> Int` which evaluates an expression.

(b) Write a function `showExpr :: ArithExpr -> String` which prints an expression as a string (so `showExpr Add (Mul (Value 2) (Value 4)) (Value 5) = "(2 * 4) + 5`.

2. Recall that the `Maybe a` datatype is defined as `data Maybe a = Just a | Nothing`.

   (a) Write a function `safeHead :: [a] -> Maybe a`, which returns the head of the list if the list is non-empty, and `Nothing` otherwise.

   (b) Write a function `safeDiv :: Int -> Int -> Maybe Int` which returns the first argument divided by the second if the second argument is not 0, and `Nothing` otherwise.

   (c) Write a function `addSafeDiv :: (Int, Int) -> (Int, Int) -> Maybe Int` which applies `safeDiv` to each pair of input numbers, and returns the results added together.

3. Suppose we have the following data type, defining n-ary trees:

   ```
   data Tree a = Leaf | Node a [Tree a]
   ```

   (a) Write a function `sumTree :: Tree Int -> Int` which, given a tree carrying values of type `Int`, returns the sum of all integers in the tree.

   (b) Write a function `reverseTree :: Tree a -> Tree a` which reverses the children at each node.

4. Suppose we have the following data type, defining *binary* trees:

   ```
   data BinaryTree a = BLeaf | BNode a (BinaryTree a) (BinaryTree a)
   ```

   (a) Write a function `toTree :: BinaryTree a -> Tree a` which transforms a binary tree into an n-ary tree.

   (b) Write a function `fromTree :: Tree a -> Maybe (BinaryTree a)` which transforms an n-ary tree into a binary tree. Return `Nothing` if any node has more than 2 children.

Further Exercises

1. Write a function `mergeSort :: (Ord a) => [a] -> [a]`, which implements the merge sort algorithm. (Note that the `Ord a` annotation ensures you are given lists whose elements can be compared).

2. Let us define a *connected word* as a word which we can write using only letters which are the same or adjacent on a QWERTY keyboard. For example, "dessert" and "pool" are connected words (e.g., 'p' is adjacent to 'o', 'o' is the same letter, and 'o' is adjacent to 'l'), but "trench" is not (as 'n' is not adjacent to 'e').

   Using the provided `connectionMap` definition which maps each character to all adjacent characters, and the included `words` file, write:

   (a) A function `isConnected :: String -> Bool` which returns `True` if a word is connected.

   (b) A function `connectedWords :: String -> IO [String]` which takes the name of a file containing a list of words, and returns a list of connected words.

   (c) A function `printStats :: IO ()` which prints the longest 5 connected words, and the number of connected words.

   Note: you can use the `sortBy :: (a -> a -> Ordering) -> [a] -> [a]` function to sort the list by length.

   See `https://hackage.haskell.org/package/base-4.17.0.0/docs/Data-Ord.html#t:Ordering` for more details on the `Ordering` data type.