Introduction

In this programming assignment, you will define a number of functions in Haskell.

Download `Homework02.hs`.

Add all functions for this assignment to the file named `Homework02.hs`.

Read and follow the course Haskell Style Guide (https://www.cs.rit.edu/~mtf/teaching/20191/psfp/style.html). Especially, look for opportunities to eliminate unnecessary complexity (use anonymous functions, use partial applications, use sectioning, use functionals).

This (and future) assignments will make use of some Haskell libraries for testing. This assignment makes use of the `Test.HUnit` module, which is installed on the CS Department Linux machines, is automatically installed when using the full installer of the Haskell Platform, and can be manually installed when using the core installer of the Haskell Platform by executing `cabal update ; cabal v2-install --lib HUnit`.

Function Functions

1. (5pts) Define a function
   
   ```haskell
   fnRepeat :: Integer -> (a -> a) -> (a -> a)
   The application fnRepeat n f should return a function that, when applied to an argument (call it x), returns f applied n times to x. The idea of fnRepeat n f is that it returns (f (f (f ... (f x)...))), with n copies of f.
   
   Examples:
   ```
   ```haskell
   • fnRepeat 10 (+ 1) 10 ∼ 20
   • fnRepeat 5 (+ 1) 10 ∼ 15
   • fnRepeat 0 (+ 1) 10 ∼ 10
   • fnRepeat 4 (∪ x → x + x) 2 ∼ 32
   • fnRepeat 0 (∪ x → x + x) 2 ∼ 2
   • fnRepeat 3 (∪ x → x ++ x) "*" ∼ "********"
   • fnRepeat 0 (∪ x → x ++ x) "*" ∼ "*
   ```

   Note that a function f applied 0 times to x is equivalent to x.
List Functions

2. (5pts) Define a function
   • listIntersperse :: a -> [a] -> [a]

   The function listIntersperse should return a list with the input element “interspersed” between all elements of the input list.

   Examples:
   • listIntersperse 0 [] \rightarrow []
   • listIntersperse 0 [1] \rightarrow [1]
   • listIntersperse 0 [1,2] \rightarrow [1,0,2]
   • listIntersperse 0 [1,2,3,4] \rightarrow [1,0,2,0,3,0,4]
   • listIntersperse ',', "" \rightarrow ""
   • listIntersperse ',', "a" \rightarrow "a"
   • listIntersperse ',', "abcd" \rightarrow "a,b,c,d"

   Note: For full credit, listIntersperse must be implemented with either the provided listFoldl function or the provided listFoldr function and must not be recursive.

   (The function listIntersperse is equivalent to the function Data.List.intersperse.)

3. (5pts) Define a function
   • listConcatMap :: (a -> [b]) -> [a] -> [b]

   The function listConcatMap should return a list that is the concatenation of the lists obtained by applying the input function to the elements of the input list.

   Examples:
   • listConcatMap (\ n -> listReplicate n '*') [] \rightarrow ""
   • listConcatMap (\ n -> listReplicate n '*') [1] \rightarrow "*"
   • listConcatMap (\ n -> listReplicate n '*') [1,2,3] \rightarrow "*****"
   • listConcatMap id [[1],[2,3],[4,5,6]] \rightarrow [1,2,3,4,5,6]

   Note: For full credit, listConcatMap must be implemented with either the provided listFoldl function or the provided listFoldr function and must not be recursive.

   (The function listConcatMap is equivalent to the Prelude function concatMap.)
4. (5pts) Define a function

   • listIsPrefix :: Eq a => [a] -> [a] -> Bool

The application listIsPrefix xs ys should return True if ys is a prefix of xs and should return False otherwise.

Examples:

   • listIsPrefix [1,2,3] [] ~ True
   • listIsPrefix [1,2,3] [1] ~ True
   • listIsPrefix [1,2,3] [1,2] ~ True
   • listIsPrefix [1,2,3] [2] ~ False
   • listIsPrefix [1,2,3] [1,2,3] ~ True
   • listIsPrefix [1,2,3] [1,2,3,4] ~ False
   • listIsPrefix [] [1,2,3] ~ False

Note: listIsPrefix must be implemented with either the provided listFoldl function or the provided listFoldr function and must not be recursive.

Warning: listIsPrefix is tricky. For partial credit, write a comment describing how you want to solve the problem.

5. (5pts) Define a function

   • listParar :: (a -> [a] -> b -> b) -> b -> [a] -> b

The function listParar is a variant of listFoldr; in the cons case, listParar provides access to the head of the list (like listFoldr), the tail of the list (unlike listFoldr), and the result of folding over the tail of the list (like listFoldr).

Examples:

   • listParar (\ x xs xss -> (x:xs):xss) [] [] ~ ([] :: [[Integer]])
   • listParar (\ x xs xss -> (x:xs):xss) [] [1] ~ [[1]]
   • listParar (\ x xs xss -> (x:xs):xss) [] [1,2] ~ [[1,2], [2]]
   • listParar (\ x xs xss -> (x:xs):xss) [] [1,2,3,4,5] ~ [[1,2,3,4,5], [2,3,4,5], [3,4,5], [4,5], [5]]
   • listParar (\ x xs xss -> (x:xs) ++ xss) [] [1,2,3,4,5] ~ [1,2,3,4,5,2,3,4,5,3,4,5,4,5,5]
   • listParar (\ x xs c -> if listIsPrefix (x:xs) "oo" then c + 1 else c) 0 "foobar" ~ 1
   • listParar (\ x xs c -> if listIsPrefix (x:xs) "oo" then c + 1 else c) 0 "foooobarfooo" ~ 5

Note: For full credit, listParar must be implemented with either the provided provided listFoldl function or the provided listFoldr function and must not be recursive.

6. (5pts) Define a function

   • listTabulate :: (Int -> a) -> Int -> [a]

The application listTabulate f n should return a list of length n equal to [f 0, f 1, ..., f (n-1)].

Examples:

   • listTabulate id 0 ~ ([] :: [Int])
   • listTabulate id 1 ~ ([0] :: [Int])
   • listTabulate id 5 ~ ([0,1,2,3,4] :: [Int])
   • listTabulate (\ i -> listReplicate i '*') 10 ~ ["\*" ,"**", "***", "****", "*****", "******", "*******", "********", "*********"]

Note: For full credit, listTabulate must be implemented with either the provided provided listUnfoldl function or the provided listUnfoldr function and must not be recursive.
7. (5pts) In *Programming in Haskell*, Graham Hutton defines an unfold for lists as follows:

\[
\text{unfold} :: (b -> \text{Bool}) \rightarrow (b \rightarrow a) \rightarrow (b \rightarrow b) \rightarrow b \rightarrow [a] \\
\text{unfold} \ p \ h \ t \ x \mid p \ x \quad = \quad [] \\
\mid \text{otherwise} = \ h \ x : \text{unfold} \ p \ h \ t \ (t \ x)
\]

Define a function

* huttonListUnfold :: (b -> \text{Bool}) \rightarrow (b \rightarrow a) \rightarrow (b \rightarrow b) \rightarrow b \rightarrow [a] 

that is equivalent to (behaves the same as) the function `unfold` given above, but must be implemented with either the provided `listUnfoldl` function or the provided `listUnfoldr` function and must not be recursive.
Tree Functions

For the following problems, use the following polymorphic algebraic data type for binary trees:

\[
\text{data Tree } a = \text{Leaf} \mid \text{Node } (\text{Tree } a) \, a \, (\text{Tree } a) \quad \text{deriving } (\text{Eq}, \text{Show})
\]

8. (5pts) Define a function

- \text{treeToList :: Tree } a \rightarrow [a]

The function \text{treeToList} should return the list of the elements of the input tree according to an in-order traversal.

Examples:

- \text{treeToList } (\text{Node } (\text{Node Leaf 7 Leaf}) \, 8 \, (\text{Node Leaf 9 Leaf})) \sim [7,8,9]
- \text{treeToList } (\text{Node Leaf 8 Leaf}) \sim [8]
- \text{treeToList Leaf} \sim []

Note: For full credit, \text{treeToList} must be implemented with the provided \text{treeFold} function and must not be recursive.

9. (5pts) Define a function

- \text{treeTakeWhile :: (a -> Bool) -> Tree } a \rightarrow Tree a

The application \text{treeTakeWhile } p \, t returns the largest (possibly empty) tree that is a prefix of \( t \) and where all elements of the result tree satisfy \( p \).

Examples:

- \text{treeTakeWhile } (\leq 2) \, t \sim \text{Leaf}
- \text{treeTakeWhile } (\leq 3) \, t \sim \text{Node Leaf 3 Leaf}
- \text{treeTakeWhile } (\leq 4) \, t \sim \text{Node (Node Leaf 4 (Node Leaf 1 Leaf)) 3 Leaf}
- \text{treeTakeWhile } (\leq 5) \, t
  \sim \text{Node (Node (Node Leaf 5 Leaf) 4 (Node Leaf 1 Leaf)) 3 Leaf}
- \text{treeTakeWhile } (\leq 6) \, t
  \sim \text{Node (Node (Node Leaf 5 Leaf) 4 (Node Leaf 1 Leaf)) 3 Leaf}
- \text{treeTakeWhile } (\leq 7) \, t
  \sim \text{Node (Node Leaf 5 Leaf) 4 (Node Leaf 1 Leaf)) 3 Leaf}
- \text{treeTakeWhile } (\leq 8) \, t
  \sim \text{Node (Node (Node Leaf 5 Leaf) 4 (Node Leaf 1 Leaf)) 3 (Node (Node Leaf 2 Leaf) 8 (Node Leaf 3 Leaf))}

where

- \( t = \text{Node (Node (Node Leaf 5 Leaf) 4 (Node Leaf 1 Leaf)) 3 (Node (Node Leaf 2 Leaf) 8 (Node Leaf 3 Leaf))} \)

Note: For full credit, \text{treeTakeWhile} must be implemented with the provided \text{treeFold} function and must not be recursive.
10. (5pts) Define functions

- `treeFoldr :: (a -> b -> b) -> b -> Tree a -> b`
- `treeFoldl :: (b -> a -> b) -> b -> Tree a -> b`

The application `treeFoldr op id t` behaves like `listFoldr op id (treeToList t)`; that is, it combines the elements of the tree in right-to-left order with the operation `op` and identity `id`.

- `treeFoldr 0 (+) (Node (Node Leaf 7 Leaf) 8 (Node Leaf 9 Leaf)) ~ 24`
- `treeFoldr 0 (+) (Node Leaf 8 Leaf) ~ 8`
- `treeFoldr 0 (+) Leaf ~ 0`
- `treeFoldr [] (:) (Node (Node Leaf 'A' Leaf) 'B' (Node Leaf 'C' Leaf)) ~ "ABC"`
- `treeFoldr [] (:) (Node Leaf 'B' Leaf) ~ "B"`
- `treeFoldr [] (:) Leaf ~ ""`

The application `treeFoldl op id t` behaves like `listFoldl op id (treeToList t)`; that is, it combines the elements of the tree in left-to-right order with the operation `op` and identity `id`.

- `treeFoldl 0 (+) (Node (Node Leaf 7 Leaf) 8 (Node Leaf 9 Leaf)) ~ 24`
- `treeFoldl 0 (+) (Node Leaf 8 Leaf) ~ 8`
- `treeFoldl 0 (+) Leaf ~ 0`
- `treeFoldl [] (:) (Node (Node Leaf 'A' Leaf) 'B' (Node Leaf 'C' Leaf)) ~ "CBA"`
- `treeFoldl [] (:) (Node Leaf 'B' Leaf) ~ "B"`
- `treeFoldl [] (:) Leaf ~ ""`

Note: `treeFoldr` and `treeFoldl` must be implemented with the provided `treeFold` function and must not be recursive.

Warning: `treeFoldr` and `treeFoldl` are tricky. For partial credit, write a comment describing how you want to solve the problem.

11. (5pts) Define a function

- `treeBuild :: Integer -> Tree Integer`

The application `treeBuild h` should return a complete binary tree of height `h` (and, therefore, of size \(2^h - 1\)) such that the elements of the tree, according to an in-order traversal, are the integers \([0..2^h-2]\).

Examples:

- `treeBuild 0 ~ Leaf`
- `treeBuild 1 ~ Node Leaf 0 Leaf`
- `treeBuild 2 ~ Node (Node Leaf 0 Leaf) 1 (Node Leaf 2 Leaf)`
- `treeBuild 3 ~ Node (Node (Node Leaf 0 Leaf) 1 (Node Leaf 2 Leaf)) 3 (Node (Node Leaf 4 Leaf) 5 (Node Leaf 6 Leaf))`

Note: `treeBuild` must use the provided `treeUnfold` function and must not be recursive.

Warning: `treeBuild` is tricky. For partial credit, write a comment describing how you want to solve the problem.
Difference Lists

A difference list is an alternative implementation of lists that uses first-class functions and provides an \(O(1)\) append operation; it is useful for algorithms that construct large lists in an irregular fashion (e.g., logging, pretty printing, web templating) and need only convert to a (standard) list at the end.

A difference list is represented by a function, which, when applied to a (standard) list, returns the “contents” of the difference list prepended to the argument (standard) list. In essence, a difference list is a data structure that makes “the list to be prepended onto” a parameter. For example, the list

```haskell
list :: [Int]
list = 1 : 2 : 3 : []
```

could be represented by the difference list (i.e., by the function) that is ready to append `list` to an argument list

```haskell
dlist :: DList Int
dlist = DList (\ ys -> list1 ++ ys)
```

which is equivalent to the difference list that replaces the `[]` at the end of `list` with a parameter

```haskell
dlist :: DList Int
dlist = DList (\ ys -> 1 : 2 : 3 : ys)
```

Making “the list to be prepended onto” a parameter provides flexibility. Calling the function with the empty list returns a list; but the function can also be called with another (non-empty) list to prepend \([1,2,3]\) onto it.

The \texttt{DList a} type is defined as follows:

```haskell
data DList a = DList ([a] -> [a])
```

That is, the \texttt{DList} constructor carries a first-class function of type \([a] \rightarrow [a]\). Converting from a \texttt{DList a} to a \([a]\) simply applies the function to the empty list:

```haskell
dlistToList :: DList a -> [a]
dlistToList (DList f) = f []
```

Converting from a \([a]\) to a \texttt{DList a} simply returns the function that is ready to append the argument list to another list:

```haskell
dlistFromList :: [a] -> DList a
dlistFromList xs = DList (xs ++)
```

Other functions that construct difference lists could be implemented by converting to and from lists, but direct implementations are more efficient.

12. (2.5pts) Define the value
   
   \(\texttt{dlistEmpty} :: \texttt{DList a}\)
   
   that is the difference list analogue of \([]\).

13. (2.5pts) Define the function
   
   \(\texttt{dlistSingleton} :: \texttt{a} \rightarrow \texttt{DList a}\)
   
   that is the difference list analogue of \((::[])\).

14. (2.5pts) Define the function
   
   \(\texttt{dlistCons} :: \texttt{a} \rightarrow \texttt{DList a} \rightarrow \texttt{DList a}\)
   
   that is the difference list analogue of \((::)\).

15. (2.5pts) Define the value
   
   \(\texttt{dlistAppend} :: \texttt{DList a} \rightarrow \texttt{DList a} \rightarrow \texttt{DList a}\)
   
   that is the difference list analogue of \((++)(\).

Note: each of the above functions must not be implemented using \texttt{dlistToList} or \texttt{dlistFromList}.
Luhn Algorithm

Note: For the problem in this section, feel free to use the Prelude list functions (map, filter, foldr, foldl,...), rather than our hand-rolled versions (listMap, listFilter, listFoldr, listFoldl,...).


16. (5pts) Define a function

   • luhnCheck :: String -> Bool

that returns True if the input string is a non-empty string of decimal digits that are valid according to the Luhn algorithm. Note that luhnCheck should return False if the input string is empty or contains any non-digit characters.

Examples:

   • luhnCheck "1784" ~> True
   • luhnCheck "4783" ~> False
   • luhnCheck "79927398713" ~> True
   • luhnCheck "79927398712" ~> False
   • luhnCheck "79972398713" ~> False
   • luhnCheck "Acct# 79927398713" ~> False
   • luhnCheck "7992739871x" ~> False
   • luhnCheck "" ~> False

Note: The useful isDigit :: Char -> Bool and ord :: Char -> Int functions have been imported from the Data.Char module.

Strive for a concise solution.

Requirements and Submission

Your submission must :load into ghci without errors; submissions that have parse errors or type errors will receive no credit. Submissions that violate code style guidelines will lose up to 25%.

Submit Homework02.hs to the Homework02 Assignment on MyCourses by the due date.