Introduction

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

Note: The purpose of this assignment is to practice programming the Haskell language, not to practice searching the Haskell libraries. Therefore, if there is a function from the Prelude or a standard library that “solves” a particular problem, then you are asked to nonetheless define the function “from scratch” and without looking at the definition of the function from the Prelude or a standard library.

Download Homework01.hs.

Add all functions for this assignment to the file named Homework01.hs.

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.

List Functions

1. (5pts) Define a function listSwizzle
   • listSwizzle :: [a] -> [a] -> [a]

   that takes two arguments (both lists) and returns one result (a list). The function listSwizzle should return a list, with the first element of the first input list as the first element of the result, the first element of the second input list as the second element of the result, the second element of the first input list as the third element of the result, the second element of the second input list as the fourth element of the result, and so on.

   Example:
   • listSwizzle [1, 2, 3] [4, 5, 6] ↦ [1,4,2,5,3,6]

   If the lists are of unequal lengths, listSwizzle retains the excess elements from the tail of the longer one.

   Example:
   • listSwizzle [1, 2, 3, 4] [5, 6] ↦ [1,5,2,6,3,4]

2. (5pts) Define a function listHasElem
   • listHasElem :: Eq a => [a] -> a -> Bool

   that takes two arguments (a list and an element) and returns one result (a boolean). The function listHasElem should return True if the argument list contains the argument element (use == from the Eq type class to compare elements) and returns False if the argument list does not contain the argument element.

   Examples:
   • listHasElem [1, 2, 3] 2 ↦ True
   • listHasElem [1, 2, 3] 4 ↦ False
   • listHasElem ['A', 'B', 'C'] 'B' ↦ True
   • listHasElem ['A', 'B', 'C'] 'D' ↦ False

   (The function listHasElem is equivalent to the Prelude function elem.)
3. (5pts) Define a function listHasDuplicates

- listHasDuplicates :: Eq a => [a] -> Bool

that takes one argument (a list) and returns one result (a boolean). The function listHasDuplicates should return True if the argument list contains duplicate elements (use == from the Eq type class to compare elements) and returns False if the argument list does not contain duplicate elements.

Examples:
- listHasDuplicates [1, 2, 3] \sim False
- listHasDuplicates [1, 2, 1] \sim True
- listHasDuplicates ['A', 'B', 'C'] \sim False
- listHasDuplicates ['A', 'B', 'A'] \sim True

4. (5pts) Define a function listDiff

- listDiff :: Eq a => [a] -> [a] -> [a]

that takes two arguments (both lists) and returns one result (a list). The function listDiff should return the “list difference” of the argument lists (the list that results from removing each of the elements of the second list from the first list; use == from the Eq type class to compare elements).

Examples:
- listDiff [1, 2, 3] [3, 2, 1] \sim []
- listDiff [1, 2, 3] [3, 4, 5] \sim [1, 2]
- listDiff [1, 1, 2, 2, 3, 3] [3, 2, 1] \sim [1, 2, 3]
- listDiff ['A', 'B', 'C'] ['B', 'B'] \sim ['A', 'C']
- listDiff ['A', 'A', 'B', 'B', 'C', 'C'] ['B', 'B'] \sim ['A', 'A', 'C', 'C']

(The function listDiff is equivalent to the Prelude function \.)

Hint: You may need to define a helper function.

5. (5pts) Define a function listSnip

- listSnip :: Eq a => [a] -> a -> a -> [a]

that takes three arguments (a list and two elements) and returns one result (a list). The function listSnip should return the list that results from removing all of the elements of the argument list from the first occurrence of the first argument element to the first following occurrence of the second argument element (use == from the Eq type class to compare elements). If the first argument element does not appear in the argument list, then listSnip returns the argument list; if the second argument element does not appear in the argument list after the first occurrence of the first argument element, then listSnip returns the elements up to the first occurrence of the first argument element.

Examples:
- listSnip [1, 2, 3, 4, 5, 6] 2 4 \sim [1, 5, 6]
- listSnip [1, 2, 3, 4, 5, 6] 7 9 \sim [1, 2, 3, 4, 5, 6]
- listSnip [1, 2, 3, 4, 5, 6] 4 9 \sim [1, 2, 3]
- listSnip [1, 2, 3, 4, 5, 6] 1 6 \sim []
- listSnip ['A', 'B', 'A', 'B', 'A', 'B'] 'B' 'B' \sim ['A', 'A', 'B']

Hint: You may need to define a helper function.
6. (5pts) Define a function `listHasSublist`

- `listHasSublist :: Eq a => [a] -> [a] -> Bool`

that takes two arguments (both lists) and returns on result (a boolean). The function `listHasSublist` should return `True` if the first argument list contains the second argument list as a (contiguous) sublist and returns `False` otherwise.

Examples:

- `listHasSublist [1, 2, 3, 4, 5] [2, 3] ~ True`
- `listHasSublist [1, 2, 3, 4, 5] [2, 4] ~ False`
- `listHasSublist [1, 2, 3] [1, 2, 3] ~ True`
- `listHasSublist [1, 2, 3] [] ~ True`
- `listHasSublist ['A', 'B', 'C', 'D', 'E'] ['B', 'C'] ~ True`
- `listHasSublist ['A', 'B', 'C', 'D', 'E'] ['B', 'D'] ~ False`
- `listHasSublist ['A', 'B', 'C'] ['A', 'B', 'C'] ~ True`
- `listHasSublist ['A', 'B', 'C'] [] ~ True`

Note that a list contains itself and the empty list as sublists.

(The function `listHasSublist` is equivalent to the function `Data.List.isInfixOf`.)

Hint: You may need to define a helper function.

7. (5pts) Define function `listRotations`

- `listRotations :: [a] -> [[a]]`

that takes one argument (a list) and returns one result (a list of lists). The function `listRotations` should return a list of all rotations of the argument list.

Examples:

- `listRotations [1, 2, 3] ~ [[1,2,3],[2,3,1],[3,1,2]]`
- `listRotations [] ~ [[]]`
- `listRotations ['A', 'B', 'C'] ~ ["ABC","BCA","CAB"]`

Note that the order in which the rotations appear in the result list does not matter.

Also note that the Haskell type `String` is a synonym for `[Char]` (i.e., a list of characters); hence, `ghci` will display a list of characters as a string constant.

8. (5pts) Define a function `listSublists`

- `listSublists :: [a] -> [[a]]`

that takes one argument (a list) and returns one result (a list of lists). The function `listSublists` should return a list containing all sublists of elements of the argument list.

Examples:

- `listSublists [1, 2, 3] ~ [[],[1],[2],[3],[1,2],[2,3],[1,2,3]]`
- `listSublists [] ~ [[]]`
- `listSublists ['A', 'B', 'C'] ~ ["","A","B","C","AB","BC","ABC"]`

Note that the order in which the sublists appear in the result list does not matter.

Also note that the Haskell type `String` is a synonym for `[Char]` (i.e., a list of characters); hence, `ghci` will display a list of characters as a string constant.
9. (10pts) Define a function \texttt{listTranspose}

- \texttt{listTranspose :: [[a]] \rightarrow [[a]]}

that takes one argument (a list of lists) and returns one result (a list of lists). The function \texttt{listTranspose} should return a list that transposes the rows and columns of the argument list; the number of rows in the result list is equal to the length of the shortest row in the argument list (and extra elements of longer rows in the argument list are ignored).

Examples:

- \texttt{listTranspose [[1, 2, 3], [4, 5, 6]] \rightarrow [[1, 4], [2, 5], [3, 6]]}
- \texttt{listTranspose [[1, 2], [3, 4, 5]] \rightarrow [[1, 3], [2, 4]]}
- \texttt{listTranspose [[1, 2, 3], [4, 5]] \rightarrow [[1, 4], [2, 5]]}
- \texttt{listTranspose [[1, 2], [], [4, 5]] \rightarrow []}
- \texttt{listTranspose [[]] \rightarrow []}
- \texttt{listTranspose [] \rightarrow []}

\textit{Warning: listTranspose is tricky. For partial credit, write a comment describing how you want to solve the problem.}
Tree Functions

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

```haskell
data Tree a = Leaf | Node (Tree a) a (Tree a) deriving (Eq, Show)
```

10. (5pts) Define a function `treeSum`
   - `treeSum :: Num a => Tree a -> a`
     that takes one argument (a tree of numbers) and returns on result (a number). The function `treeSum` should return the sum of the elements of the argument tree (use `+` from the `Num` type class to add elements).
     Examples:
     - `treeSum (Node (Node Leaf 7 Leaf) 8 (Node Leaf 9 Leaf)) ≈ 24`
     - `treeSum Leaf ≈ 0`
     - `treeSum (Node (Node Leaf 7.5 Leaf) 8.5 (Node Leaf 9.5 Leaf)) ≈ 25.5`

11. (5pts) Define a function `treeProduct`
   - `treeProd :: Num a => Tree a -> a`
     that takes one argument (a tree of numbers) and returns on result (a number). The function `treeProduct` should return the product of the elements of the argument tree (use `*` from the `Num` type class to multiply elements).
     Examples:
     - `treeProduct (Node (Node Leaf 7 Leaf) 8 (Node Leaf 9 Leaf)) ≈ 504`
     - `treeProduct Leaf ≈ 1`
     - `treeProduct (Node (Node Leaf 7.5 Leaf) 8.5 (Node Leaf 9.5 Leaf)) ≈ 605.625`

12. (5pts) Define a function `treeHasElem`
   - `treeHasElem :: Eq a => Tree a -> a -> Bool`
     that takes two arguments (a tree and an element) and returns one result (a boolean). The function `treeHasElem` should return `True` if the argument tree contains the argument element (use `==` from the `Eq` type class to compare elements) and returns `False` if the argument tree does not contain the argument element.
     Examples:
     - `treeHasElem (Node (Node Leaf 7 Leaf) 8 (Node Leaf 9 Leaf)) 8 ≈ True`
     - `treeHasElem (Node (Node Leaf 7 Leaf) 8 (Node Leaf 9 Leaf)) 9 ≈ True`
     - `treeHasElem (Node (Node Leaf 7 Leaf) 8 (Node Leaf 9 Leaf)) 6 ≈ False`
     - `treeHasElem (Node (Node Leaf 'A' Leaf) 'B' (Node Leaf 'C' Leaf)) 'B' ≈ True`
     - `treeHasElem (Node (Node Leaf 'A' Leaf) 'B' (Node Leaf 'C' Leaf)) 'A' ≈ True`
     - `treeHasElem (Node (Node Leaf 'A' Leaf) 'B' (Node Leaf 'C' Leaf)) 'Z' ≈ False`
13. (5pts) Define a function \texttt{treeHasSubtree}

- \texttt{treeHasSubtree :: Eq a => Tree a -> Tree a -> Bool}

that takes two arguments (both trees) and returns on result (a boolean). The function \texttt{treeHasSubtree} should return \texttt{True} if the first argument tree contains the second argument tree as a (contiguous) subtree and returns \texttt{False} otherwise.

Examples:

- \texttt{treeHasSubtree t Leaf} \sim\texttt{True}
- \texttt{treeHasSubtree t (Node Leaf 8 Leaf) \sim\texttt{True}}
- \texttt{treeHasSubtree t (Node Leaf 9 Leaf) \sim\texttt{True}}
- \texttt{treeHasSubtree t (Node Leaf 6 Leaf) \sim\texttt{False}}
- \texttt{treeHasSubtree t (Node (Node Leaf 7 Leaf) 8 Leaf) \sim\texttt{True}}
- \texttt{treeHasSubtree t (Node (Node Leaf 7 Leaf) 8 (Node Leaf 9 Leaf)) \sim\texttt{True}}
- \texttt{treeHasSubtree t (Node Leaf 8 (Node Leaf 9 Leaf)) \sim\texttt{True}}
- \texttt{treeHasSubtree t (Node (Node Leaf 7 Leaf) 88 (Node Leaf 9 Leaf)) \sim\texttt{False}}

where

- \texttt{t = Node (Node Leaf 7 Leaf) 8 (Node Leaf 9 Leaf)}

Note that a tree contains itself and the empty (Leaf) tree as a subtree.

Hint: You may need to define a helper function.

14. (10pts) Define a function \texttt{treeReverse}

- \texttt{treeReverse :: Tree a -> Tree a}

that takes one argument (a tree) and returns one result (a tree). The function \texttt{treeReverse} returns a tree with the same shape as the argument tree and with an in-order traversal of elements the reverse of the in-order traversal of elements of the argument tree.

Examples:

- \texttt{treeReverse Leaf} \sim\texttt{Leaf}
- \texttt{treeReverse (Node Leaf 8 Leaf)} \sim\texttt{Node Leaf 8 Leaf}
- \texttt{treeReverse (Node (Node Leaf 7 Leaf) 8 Leaf)} \sim\texttt{Node (Node Leaf 8 Leaf) 7 Leaf}
- \texttt{treeReverse t \sim\texttt{tr}}

where

- \texttt{t = Node (Node Leaf 1 (Node (Node Leaf 2 Leaf) 3 Leaf)) 4 (Node Leaf 5 (Node (Node Leaf 6 Leaf) 7 (Node (Node Leaf 8 Leaf) 9 Leaf)))}
- \texttt{tr = Node (Node Leaf 9 (Node (Node Leaf 8 Leaf) 7 Leaf)) 6 (Node Leaf 5 (Node (Node Leaf 4 Leaf) 3 (Node (Node Leaf 2 Leaf) 1 Leaf)))}

Warning: \texttt{treeReverse} is tricky. For partial credit, write a comment describing how you want to solve the problem.

Requirements and Submission

Your submission must :\texttt{load} into \texttt{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 \texttt{Homework01.hs} to the \texttt{Homework01 Assignment} on MyCourses by the due date.