Programming Exercise 3

Due Apr 17 by 11:59pm **Points** 50 **Submitting** a file upload

Programming Exercise 3

Due Friday, April 17

Be sure to comment your code and include your name at the top of the file.

For many of the functions below, you are to welcome use helper functions. Your goal is to make the coding logic as simple as you can.

Part 1: Haskell Functions

1. Create the function `interleave3` that takes three lists. It returns a list where the elements are interleaved in the pattern: *list1, list2, list3, list1, list2, list3, ...*

```
*Main> interleave3 [1,2,3,4] [10,20] [100,200,300,400,500]
[1,10,100,2,20,200,3,300,4,400,500]
```

2. Create the function `removeLast` takes a list and removes the last element of the list. You may assume the list has at least one element.

```
*Main> removeLast [1,2,3,4,5]
[1,2,3,4]
```

3. Create the function `insertInOrder` that takes a list of elements that is in sorted order and a value, and it places the value into the proper place in the list.

```
*Main> insertInorder 6 [1,2,4,7,8]
[1,2,4,6,7,8]
```

Part 2: Haskell Types

4. Using the `BinaryTree` type created in class, write the four AVL-tree functions `rotateleft`, `rotateright`, `rotateleftright` and `rotaterightleft`. You may assume the inputs to the functions have all the necessary children. (Note: these functions just do the rotations from the input node, you are not to search the tree for imbalances.)

```
*Main> rotateleft (InnerNode 5 (Leaf 3) (InnerNode 10 (InnerNode 7 (Leaf 6) (Leaf 8)) (InnerNode 20 (Leaf 15) (Leaf 25))))
InnerNode 10 (InnerNode 5 (Leaf 3) (InnerNode 7 (Leaf 6) (Leaf 8))) (InnerNode 20 (Leaf 15) (Leaf 25))
*Main> rotaterightleft (InnerNode 5 (Leaf 3) (InnerNode 10 (InnerNode 7 (Leaf 6) (Leaf 8)) (InnerNode 20 (Leaf 15) (Leaf 25))))
InnerNode 7 (InnerNode 5 (Leaf 3) (Leaf 6)) (InnerNode 10 (Leaf 8) (InnerNode 20 (Leaf 15) (Leaf 25)))
```

5. Using the `BinaryTree` type created in class, write a function `removeMin` that takes a `Tree` as input. Assuming the tree is in proper order (all values in the left child are smaller than the value in the node, and all the values in the right child are equal or larger than the node), the function will return a new tree with the smallest value of the tree removed. (If you recall 233, if the minimum value is an internal node, you replace the value of the node with the smallest value of the right child.) The resulting tree should not have an internal node with both children empty.

```
*Main> removeMin (InnerNode 5 (InnerNode 0 Empty (InnerNode 4 (InnerNode 3 Empty (Leaf 2) Empty)) Empty)) (InnerNode 8 (Leaf 6) (Leaf 10)))
InnerNode 5 (InnerNode 1 Empty (InnerNode 4 (InnerNode 2 Empty (Leaf 3)) Empty)) (InnerNode 8 (Leaf 6) (Leaf 10))
```

6. While Haskell is similar to Scheme, Haskell's type rules prevent us from writing a function like the '-functions of the first Scheme homework. For example, we can't write the equivalent of `(reverse- '(1 2 3 4 5 (1 ((5 3) ())) 1 6 2))` because a list can't contain both int types and list types as elements. You will fix this by creating the following type.

   Create a type that allows us to have nested lists. Your type should have two kinds of values, elements and sublists. For example, the following will be a valid list:

```
[Element 1,Element 3,SubList [Element 4,SubList [SubList [Element 5],SubList []],Element 6]
```

7. Create the function `preverse` that takes your general list structure and reverses the list. The contents of any sublist is also reversed.

```
*Main> preverse [Element 4, Element 5, SubList [Element 6, Element 7, SubList [Element 3, Element 1], Element 2]]
[SubList [Element 2, SubList [Element 1, Element 3], Element 7, Element 6], Element 5, Element 4]
```

8. Create the function `partialsums` that takes your general list containing numbers. The output should have the same structure as the input list but the only (non-sublist) element in each list and sublist should be the sum of all numbers in that list.

```
*Main> partialsums [SubList [Element 1], SubList [Element 2], SubList [Element 3]]
[Element 6, SubList [Element 1], SubList [Element 2], SubList [Element 3]]
*Main> partialsums [Element 4, Element 5, SubList [Element 6, Element 7, SubList [Element 3, Element 1], Element 2]]
[Element 28, SubList [Element 19, SubList [Element 4]]]
```

Part 3: Haskell Monads

9. Create the function `sum_of_maxes` that takes two lists of lists of numbers and creates a single list of numbers. The kth value of the output list is the largest value of the kth sublist of the first input list added to the largest value of the kth sublist of the second input list. For example, if the first input list is `[[1,5,3],[3,7],[1,1,3]]` and if the second input list is `[[8,3],[1],[3,6,3,9,3]]`, the output is `[13, 8, 12]` because the maximum of each of the sublists of the first list are 5, 7, 3, and the maximums of each of the sublists of the second list are 8, 1, 9, then these maxaxmums are summed together. This routine should fail if any sublist is empty (so there is no maximum value), or if each input list has a different number of sublists. Use the `Maybe` monad of Haskell so that you can write this function so *that it does a single pass of the data*. The function should return `Just [13, 8, 12]` (from the example above) if given valid input and `Nothing` if the function fails.

```
*Main> sum_of_maxes [[1,5,3],[3,7],[1,1,3]] [[8,3],[1],[3,6,3,9,3]]
Just [13,8,12]
*Main> sum_of_maxes [[1,5,3],[3,7],[1,1,3]] [[8,3],[1],[3,6,3,9,3],[4,1]]
Nothing
*Main> sum_of_maxes [[],[3,4]] [[1],[3,6]]
Nothing
```

10. In Haskell, lists are monads. For example, `[1,2,3] >>= (\w -> [2*w])` produces the list `[2,3,4]`. In this problem, you are to figure out how that works.

    Create a list monad that generalizes a list. This will not be a Haskell Monad type, but instead one of our own creation like the `Value` type from lecture. For example, the following is a valid "list".

```
Pair 4 (Pair 5 (Pair 6 Null))
```

    Then create a binding function `lbind` and a return function `lreturn` to make a list monad. The code should work so that

```
(Pair 4 (Pair 5 (Pair 6 Null))) `lbind` (\x -> lreturn (2 * x))  =>  Pair 8 (Pair 10 (Pair 12 Null))
```

Homework Exercise 3

Criteria	Ratings					Pts
interleave3	**5.0 pts** **Full Marks** A correct and elegant Haskell solution to the problem.	**4.0 pts** **Good** A solution with minor mistakes but shows a good understanding of typed, functional coding.	**3.0 pts** **Reasonable** A typed and functional solution with significant mistakes. A correct functional solution that makes mistakes with the type, or correctly uses the Haskell types but the solutions is in the imperative style.	**2.0 pts** **Poor** The solution does not show an understanding of typed, functional coding, and cannot work for the problem, but is there is some reasonable logic for the problem in the Haskell code.	**1.0 pts** **Minimal** There is something in Haskell that can apply to the problem.	**0.0 pts** **No Marks** Nothing useful to grade	5.0 pts
removelast	**5.0 pts** **Full Marks** A correct and elegant and efficient Haskell solution to the problem.	**4.0 pts** **Good** A solution with minor mistakes but shows a good understanding of typed, functional coding OR a correct solution that is not efficient.	**3.0 pts** **Reasonable** A typed and functional solution with significant mistakes. A correct functional solution that makes mistakes with the type, or correctly uses the Haskell types but the solutions is in the imperative style.	**2.0 pts** **Poor** The solution does not show an understanding of typed, functional coding, and cannot work for the problem, but is there is some reasonable logic for the problem in the Haskell code.	**1.0 pts** **Minimal** There is something in Haskell that can apply to the problem.	**0.0 pts** **No Marks** Nothing useful to grade	5.0 pts
insertinorder	**5.0 pts** **Full Marks** A correct and elegant and efficient Haskell solution to the problem that properly uses CPS and tail recursion.	**4.0 pts** **Good** A solution with minor mistakes but shows a good understanding of typed, functional coding with CPS and tail recursion.	**3.0 pts** **Reasonable** A typed and functional CPS solution with significant mistakes OR a solution that uses CPS but is not tail recursive.	**2.0 pts** **Poor** The solution does not show an understanding of typed, functional coding and/or CPS, and cannot work for the problem, but is there is some reasonable logic for the problem in the Haskell code.	**1.0 pts** **Minimal** There is something in Haskell that can apply to the problem.	**0.0 pts** **No Marks** Nothing useful to grade	5.0 pts
rotate functions	**5.0 pts** **Full Marks** A correct and elegant Haskell solution to the problem that correctly uses the tree type from lecture.	**4.0 pts** **Good** A functional solution with only minor mistakes that demonstrates correct ways to use the tree type from lecture.	**3.0 pts** **Reasonable** A typed and functional solution with significant mistakes that uses the tree type from lecture. A correct functional solution that makes mistakes with the type, or correctly uses the Haskell types but the solutions is inefficient or not following good functional style.	**2.0 pts** **Poor** The solution does not show an understanding of typed, functional coding, and cannot work for the problem, but is there is some reasonable logic for the problem in the Haskell code OR is a reasonable Haskell solution but is not using the tree type from lecture.	**1.0 pts** **Minimal** There is something in Haskell that can apply to the problem.	**0.0 pts** **No Marks** Nothing useful to grade	5.0 pts
removeMin	**5.0 pts** **Full Marks** A correct and elegant Haskell solution to the problem that correctly uses the tree type from lecture.	**4.0 pts** **Good** A functional solution with only minor mistakes that demonstrates correct ways to use the tree type from lecture.	**3.0 pts** **Reasonable** A typed and functional solution with significant mistakes that uses the tree type from lecture. A correct functional solution that makes mistakes with the type, or correctly uses the Haskell types but the solutions is inefficient or not following good functional style.	**2.0 pts** **Poor** The solution does not show an understanding of typed, functional coding, and cannot work for the problem, but is there is some reasonable logic for the problem in the Haskell code OR is a reasonable Haskell solution but is not using the tree type from lecture.	**1.0 pts** **Minimal** There is something in Haskell that can apply to the problem.	**0.0 pts** **No Marks** Nothing useful to grade	5.0 pts
nested list type	**5.0 pts** **Full Marks** A correct type definition for the nested list.	**4.0 pts** **Good** A type definition for the nested list with minor errors such as unnecessary constructors.	**3.0 pts** **Reasonable** Is a valid Haskell type definition that will create a list-like structure but has significant errors.	**2.0 pts** **Poor** The solution is not a valid Haskell type definition but shows some list logic OR is a valid Haskell type definition and but does not show logic for the list.	**1.0 pts** **Minimal** Is an attempt at a Haskell type definition.	**0.0 pts** **No Marks** Nothing useful to grade	5.0 pts
greverse	**5.0 pts** **Full Marks** A correct and elegant Haskell solution to the problem that correctly uses the type of question 6.	**4.0 pts** **Good** A solution that uses the type of question 6 with minor mistakes but shows a good understanding of typed, functional coding.	**3.0 pts** **Reasonable** A typed and functional solution with significant mistakes that uses the type from question 6. A correct functional solution that makes mistakes with the type, or correctly uses the Haskell types but the solutions is in the imperative style.	**2.0 pts** **Poor** The solution does not show an understanding of typed, functional coding, and cannot work for the problem, but is there is some reasonable logic for the problem in the Haskell code OR is a reasonable Haskell solution but is not using the type from question 6.lecture.	**1.0 pts** **Minimal** There is something in Haskell that can apply to the problem.	**0.0 pts** **No Marks** Nothing useful to grade	5.0 pts
partialsums	**5.0 pts** **Full Marks** A correct and elegant Haskell solution to the problem that correctly uses the type of question 6.	**4.0 pts** **Good** A solution that uses the type of question 6 with minor mistakes but shows a good understanding of typed, functional coding.	**3.0 pts** **Reasonable** A typed and functional solution with significant mistakes that uses the type from question 6. A correct functional solution that makes mistakes with the type, or correctly uses the Haskell types but the solutions is in the imperative style.	**2.0 pts** **Poor** The solution does not show an understanding of typed, functional coding, and cannot work for the problem, but is there is some reasonable logic for the problem in the Haskell code OR is a reasonable Haskell solution but is not using the type from question 6.	**1.0 pts** **Minimal** There is something in Haskell that can apply to the problem.	**0.0 pts** **No Marks** Nothing useful to grade	5.0 pts
sum_of_maxes	**5.0 pts** **Full Marks** A correct and elegant Haskell solution to the problem that correctly uses Haskell's Maybe monad and does only one traversal of the data.	**4.0 pts** **Good** A solution with minor mistakes but shows a good understanding of typed, functional coding and how to use the Maybe monad.	**3.0 pts** **Reasonable** A typed and functional solution with significant mistakes that uses the Maybe monad and has correct logic for the problem OR a good solution that uses the Maybe monad but does multiple traversals of the data like a "normal" non-Maybe recursion.	**2.0 pts** **Poor** The solution does not show an understanding of typed, functional coding with the Maybe monad OR a solution with only minor errors but uses a different monad.	**1.0 pts** **Minimal** There is something in Haskell that can apply to the problem but no attempt at using a monad.	**0.0 pts** **No Marks** Nothing useful to grade	5.0 pts
list monad	**5.0 pts** **Full Marks** Correctly defines a parameterized list type, a parameterized type for the monad, a properly typed bind and return functions.	**4.0 pts** **Good** Correct list and monad types and functions, but there are minor errors that prevent it from working in all cases.	**3.0 pts** **Reasonable** Defines a list type and a monad type but significant errors make it so that it can't work as described.	**2.0 pts** **Poor** The solution shows some monad and list logic but otherwise is not close to working	**1.0 pts** **Minimal** There is something in Haskell that has a list or monad type logic, but not both.	**0.0 pts** **No Marks** Nothing useful to grade	5.0 pts
						Total Points: 50.0	