CS 330 Homework

Fun Fun: Functional Languages

1 Overview

Your responsibility in this homework is to solve problems in a functional approach using Haskell. Your instructor finds it quite difficult to write a complete program (with I/O and staging of data structures) in most functional languages, so this assignment will instead focus on smaller problems. Your instructor's hope is that you can apply the ideas seen in functional languages to your everyday programming, whatever language that might be in.

2 Requirements

In order to complete this homework, please satisfy the following specification:

1. Write all code in a file named `Funfun.hs` in a directory named `funfun`. The case of the file name is important; Haskell's module system requires an initial capital letter. A skeleton of this file with some testing code has been placed in W330.

2. Write a function `everyOther` that takes a list as a parameter and returns a list containing only every other element, starting with the first. For example `everyOther [1..5]` → `[1, 3, 5]`. (3 cases)

3. Write two functions `fsts` and `snds` that both accept a list of pairs as a parameter. `fsts` returns a list of the first components in the pairs, `snds` returns a list of second components. For example, `fsts [(1,2), (3,4)]` → `[1,3]`. Write these in point free style and use `map`. (28 characters total)

4. Write a function `bounds1Accum` that accepts a list of numbers, a minimum, and a maximum. It returns the minimal and maximal value found amongst the list elements and the two parameters. The minimal and maximal values are returned as a pair. For example, `bounds1Accum [] 1 5` → `(1, 5)` and `bounds1Accum [6, 9, 3] 5 10` → `(3, 10)`. (Hint: the base case is trivial. The recursive case tries to best the parameters and the head.) Note that this function is tail recursive; it uses two accumulators. (2 cases)

5. Write a function `bounds1` that wraps around `bounds1Accum`, making it easier for clients to call. It takes a list parameter. If the list is empty, it returns `Nothing`. Otherwise, it returns `Just` the result of calling `bounds1Accum` with the head of the list as the initial accumulator values. (2 cases)

6. Write a function `bounds2` that accepts a list of pairs as a parameter—pretend that each elements is an (x, y) coordinate pair. It returns a pair of `Maybe`s whose first

element is the x extrema and whose second element is the y extrema. For example, `bounds2 [(1,2), (3,4), (0,0)]` → `(Just (0,3), Just (0,4))`. `bounds2` of the empty list is a pair of `Nothing`s. (1 case; 54 characters total; use three of the functions you just wrote!)

7. Write a function `filtermask` that accepts a function, a mask list, and a data list. Much like `filter`, it returns a subset of the data list. Which elements are filtered depends on the mask list. It applies the function to an element of the mask list and the corresponding element in the data list, and if the function returns true, the corresponding element in the data list is included in the returned list. For example, suppose we want to filter out data items that are greater than the mask items. We could do so with: `filtermask (\m d -> d > m) [1, 5, 7] [5, 6, 7]` → `[5, 6]`. (3 cases)

8. Write a function `filterTrues` that partially applies `filtermask` with a lambda function that returns true if the mask item is true. Use point free style. For example, `filterTrues [True, False, True] [1, 2, 3]` → `[1, 3]`. (37 characters, 1 case)

9. Write a function `sumTrues` that accepts a mask and a data list and sums up the data values whose corresponding mask values are true. Use `filterTrues`. For example, `sumTrues [True, False, True] [1, 2, 3]` → 4. (48 characters, 1 case)

10. Write a function `filterDiffs` that partially applies `filtermask` with a lambda function that returns true if the mask item is different from the data item. Use point free style. For example, `filterDiffs ["a", "B", "Z"] ["A", "B", "C"]` → `["A", "C"]`. (41 characters, 1 case)

11. Write a function `joinDiffs` that accepts a mask and a data list and concatenates the data values that are different from the mask values, each separated by a newline. Assemble the results with a call to `foldl` and a lambda concatenator. For example, `joinDiffs ["a", "B", "Z"] ["A", "B", "C"]` → `"A\nC\n"`. (83 characters, 1 case)

12. This next few problems involve writing functions to play a game of Battleship. After you complete these, you can test your game by executing `play` or `playLonely`.

    (a) Write a type `Row` consisting of constructors named `A` through `J`. Write a type class `Column` consisting of constructors named `One` through `Ten`. Have both derive `Enum`, `Ord`, `Show`, `Bounded`, `Eq`, and `Read`. These two are very much like typesafe enums. You will use data of these types to identify board positions.

    (b) Write a type `Address` with one constructor named `Address` taking a `Row` and `Column`. Have it derive `Show`, `Read`, and `Eq`.

    (c) Write a type `Cell` with one constructor named `Cell` taking an `Address` and a `Bool`. Have it derive `Show`, `Read`, and `Eq`. A cell will be used to identify the location and sunk status of one segment of your ship. If the `Bool` is true, the segment at the given address is sunk.

(d) Write a type Ship with one constructor named Ship taking a list of Cells. Have it derive Show and Eq.

(e) Write a type Ships with one constructor named Ships taking a list of Ships. Have it derive Show and Eq.

(f) Write a function toAddress that takes in a row and column, each in $[0-9]$. Construct an Address and return it. Use toEnum to index into your Row and Column enums lists.

(g) Write a function toRowColumn that takes in an Address and returns a pair of integers—the row and column number of that address. Use fromEnum. You could use this function to index into a 2-D array for your board, though we don't assign this task.

(h) Define allAddressesA as the list of all possible board Addresses. Use a list comprehension that uses ranges of your Row and Column types.

(i) Define allAddressesB as the list of all possible board Addresses. Use a list comprehension that uses integral ranges and turns the integral pairs into addresses with toAddress.

(j) Write a function targetShip that accepts a Ship and a target address as its two parameters. If the ship has a cell with the target address, that cell is marked sunk. An updated version of the ship is returned. If the ship is empty, the empty Ship is returned. (1 map, 116 characters)

(k) Write a function targetShips that accepts a list of Ships and a target address as its two parameters. It attempts to sink each ship in the ship list. It returns a list of updated ships. If no ships are present, the empty list is returned. (1 map, 86 characters)

(l) Write a function isSunk that accepts a Ship as its parameter. If the ship is the empty ship, return true. Otherwise, return true if all cells are marked sunk. (1 fold, 78 characters)

(m) Write a function areSunk that accepts a list of Ships as its parameter. If no ships are present, return true. Otherwise, return true if all ships are marked sunk. (1 fold, 78 characters)

13. Zip up your directory to funfun.zip with

```
(cd .. && zip -r funfun.zip funfun)
```

This command assumes your current working directory is funfun. The *.zip file is placed in the parent directory.

14. To test your code, run the following:

```
(cd .. && ~/w330/funfun/test_funfun)
```

This script only tests a few things like proper names and basic functionality. This script does not test all requirements. You need to do your own testing too. Failure to do so will likely result in a rejected submission. After you pass these tests, the script will prompt you to submit and notify us of your submission.

3 Meta

The reference implementation is 44 lines of code, counting separating whitespace. That's really small, and is typical of coding in a functional style. However, don't be deceived. One spends considerably more time writing each line, at least initially.