CS 330 Homework

Fun Fun

1 Overview

Your responsibility in this homework is to solve problems using a functional approach. You will use Haskell. Instead of repeating common patterns of iteration, you will rely on higher-order functions like `map`, `fold`, and `filter`, which alleviate much of the boilerplate that we write in traditional imperative languages like C.

Since I/O is an imperative task, writing a complete interactive program in a pure functional language is awkward. Instead of overcoming this awkwardness and learning how Haskell's handles I/O, you will solve a handful of decontextualized non-interactive utility problems that have no overarching theme. The ideas we see in Haskell and other functional languages are still likely to benefit your programming where I/O is common, whether that's in Ruby, Scala, Java 8, or otherwise.

2 Requirements

In order to complete this homework, you must satisfy these requirements:

1. Place all code in file `<YOUR-REPOSITORY>/funfun/Funfun.hs`. Case matters.

2. Your code must be defined in a module named `Funfun`. You will need to rely on a couple of library modules. This is done by starting your file off with these lines:

   ```
   module Funfun where

   import Data.Function.Memoize
   import Data.List
   ```

   To get `memoize`, run:

   ```
   cabal update
   cabal install --enable-shared memoize
   ```

   This assumes Haskell package manager `cabal` is installed on your computer.

3. All code must run on `thing-0[456]`. You can test interactively it in the shell in the following manner:

   ```
   $ ghci
   > :l Funfun.hs
   > twoByTwo [1..10]
   ```

4. Write function type declarations for each function you are asked to write. Haskell's type inference makes declarations optional, but writing them usually improves error messages and your own understanding of what you are trying to do.

5. Write function `twoByTwo` to accept a list of anything. It yields a new list, in which each pair of neighboring elements is tupled together. For example, `twoByTwo [1..4]` → `[(1, 2), (3, 4)]`. If the parameter list contains an odd number of elements, ignore the last element.

   Implementation hint: pattern matching is helpful here to capture the cases of the parameter list.

6. Write function `sumOfSquares` to accept a list of anything in the `Num` typeclass. It yields the sum of the squared list elements, which will also be in the `Num` typeclass. For example, `sumOfSquares [1..3]` → `14`. Implement this function as a one-liner in the point-free style by composing these functions: `map` and `sum`.

7. Write function `squareOfSum` to accept a list of anything in the `Num` typeclass. It yields the square of the summed list elements, which will also be in the `Num` typeclass. For example, `squareOfSum [1..3]` → `36`. Implement this function as a one-liner in the point-free style by composing these functions: `sum` and `^`.

8. Write function `squmDiff` to accept an `Int` and return an `Int`. It considers the list starting at 1 and ending inclusively at the parameter. It returns the difference between the squared sum of this list and the summed squares of the list.

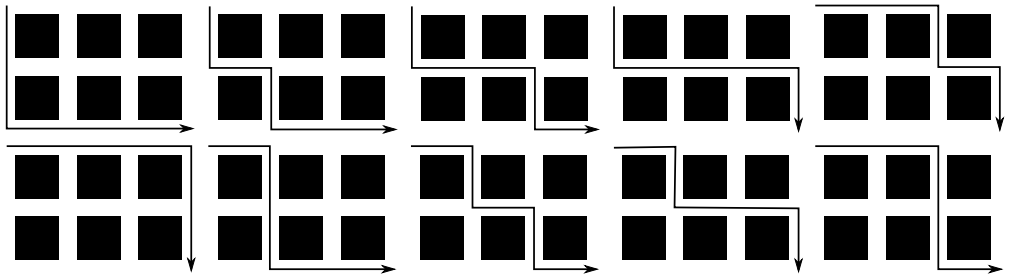
9. Write function `mode` to accept a list of anything in the `Ord` typeclass. It yields the statistical mode of the list, the element that appears most frequently. For example, `mode "abcadefzzz"` → `'z'`. Implement this function as a one-liner in the point-free style by composing these functions from modules `Data.List` and `Prelude`: `group`, `head`, `sort`, and `maximumBy`.

10. Write function `indices` to accept two lists of anything in the `Eq` typeclass. It yields a list of `Int`s—the indices at which an instance of the first list starts in the second list. For example, `indices [1, 3] [1, 3, 1, 3, 5, 1]` → `[0, 2]`.

    Implementation hint: functions `isPrefixOf`, `tails`, and `findIndices` can be used to solve this in one line.

11. Write function `flatten2` that accepts a list of lists of anything and yields a list of the same type, but flattened. We define *flattened* as having all elements from the inner lists migrated to an outer list. For example, `flatten2 [[4, 5], [7, 9]]` → `[4, 5, 7, 9]`. The flattening is not recursive—i.e., an inner list composed of lists is not itself flattened. Note that you are reducing a collection down to a new value, an operation that begs to be solved using one of the `fold` operations. Implement this function as a one-liner in the point-free style.

12. Imagine a neighborhood with streets running directly east-west and north-south. It is composed of $w \times h$ blocks. You are trying to get from the northwest corner to the southeast corner. How many different routes can you choose from? Consider only the shortest paths, in which each step takes you closer to your destination. You never go north or west, only east and south. For a 3 by 2 grid, you have 10 choices:



Write function `nroutes` that accepts the width and height of this neighborhood as `Int`s and yields the number of routes. For example, `nroutes 3 2` → `10`.

Instead of trying to crack the formula to compute the number of routes, use a naive recursive implementation. Consider two trivial cases: the neighborhood has degenerated to invalid dimensions (yielding no routes) or you are already at your destination (yielding exactly one 1 route). In the general case, you have two choices: move south or move east. If you move south, then you have only to compute the number of routes to get through a smaller neighborhood whose height is one less than the original. If you move east, then you have only to compute the number of routes to get through a smaller neighborhood whose width is one less than the original. However, don't just pick one—you are computing the *total* number of routes.

The number of recursive calls grows quickly, and you are likely to run out of memory or time in the universe before you find a solution. Write a helper function that does the work described above, and then define `nroutes` by `memoize2`ing the helper function. Memoization caches the results of function calls keyed on the actual parameters. Subsequent calls given the same parameters will be retrieved from cache instead of being reevaluated.

13. Write a type synonym named `XY` that resolves to a tuple of `Double`s.

14. Write function `translate` that accepts two parameters: an offset `XY` and a point `XY`. Yield a new `XY` representing the point shifted by the offset. For example, `translate (1, 2) (-1, -1)` → `(0, 1)`.

15. Write function `scale` that accepts two parameters: a scale factor `XY` and a point `XY`. Yield a new `XY` representing the point scaled by the factors. For example, `scale (3, 2) (-1, -4)` → `(-3, -8)`.

16. Write function `rotate` that accepts two parameters: a `Double` number of degrees and a point `XY`. Yield a new `XY` representing the point rotated by the specified number of degrees. For example, `rotate 45 (1, 2)` $\rightsquigarrow$ `(-0.707, 2.121)`. Use the following equations to rotate in two dimensions:

$$x' = x * \cos\theta - y * \sin\theta$$
$$y' = x * \sin\theta + y * \cos\theta$$

17. Write function `transformAll` that accepts two parameters: a function that transforms its parameter `XY` to a new `XY` (like `translate`, `scale`, and `rotate`) and a list of `XY`s. It yields a list of all the transformed `XY`s. For example, `transformAll (\(x, y) -> (y, x)) [(1, 2), (3, 4)]` $\rightarrow$ `[(2, 1), (4, 3)]`. Don't write much code here; use your functional arsenal.

18. Write function `gauntlet` that accepts two parameters: a list of functions that transform their parameter `XY`s to new `XY`s (like a list of `translate`s, `scale`s, and `rotate`s) and an `XY`. It yields an `XY` that has been successively transformed by all the functions in the list, with the head applied first. For example, `gauntlet [scale (3, 4), translate (1, 2)] (10, 20)` $\rightarrow$ `(31, 82)`. Don't write much code here; use your functional arsenal.

19. Write function `gauntletAll` that accepts two parameters: a list of functions that transform their parameter `XY`s to new `XY`s (like a list of `translate`s, `scale`s, and `rotate`s) and a list of `XY`s. It yields a list of `XY`s built by transforming each `XY` in the second parameter by all the functions in the first parameter. For example, `gauntletAll [scale (3, 4), translate (1, 2)] [(10, 20), (0, 2)]` $\rightarrow$ `[(31, 82), (1, 10)]`. Don't write much code here; use your functional arsenal.

To be eligible for later-week submission, you must successfully complete functions `twoByTwo`, `squareOfSum`, `sumOfSquares`, and `squmDiff`. Additionally, the unit tests must be able to compile and link against your code. Since Haskell is statically typed, this means *all* functions must be stubbed out. Have these stubs yield the empty list, `(0, 0)`, or some other appropriate trivial value.

3 Submission

To submit your work for grading:

1. Run the grading script from your homework directory using `../specs/grade`.
2. Commit and push your work to your repository.
3. Verify that your solution is on Bitbucket by viewing your repository in a web browser.

A passing grading script does not guarantee you credit. Your grade is conditioned on a few things:

- You must meet the requirements described above. The grading script checks some of them, but not all.
- You must successfully submit your code to your repository. Expect to have issues with Git.
- You must not plagiarize. Write your own code. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. Your instructor employs a vast repertoire of tools to sniff out academic dishonesty, including: drones, moles, and a piece of software called MOSS that rigorously compares your code to every other submission. You don't want to live in a world serviced by those who squeaked by through questionable means. For your future self, career, and family, do your own work.

The grading script allows you to signal your instructor when requirements are met. You only need to send an email if you are resubmitting.