CS 463 - Homework 4, Comparison Code (Haskell)

deadline:
Friday March 4th, 5pm on Blackboard

Table of Contents

1 Notes

1.1 Installation

1.1.1 Getting GHC (the compiler+interpreter)

You need to install the Haskell Platform on your machine in order to run your code. You'll have the `ghci` (and `ghc`) executables, plenty of libraries available, and more. On Windows, they may suggest installing the `Chocolatey` package manager to streamline the process.

Installing GHC

- The GHC installation page has been updated and now suggests that all platforms use GHCUP as the installation path:
  - https://www.haskell.org/ghcup/
  - the version offered is current enough for our class; last I checked that is 8.10.7, but anything 8.x or greater should be fine.

1.1.2 Getting libraries

Then, you'll want a couple of libraries to be available for our testing harness. It may be worth it to just install a package manager called `cabal`, with these commands:

Installing packages:

- install the `cabal-install` package, and use its commands.
  - available as part of the Haskell Platform
    - you can also download the package here (Downloads section), and follow the instructions on the same page under the Readme section:
      - https://hackage.haskell.org/package/cabal-install
  - each time you use it, run commands like this; for example, to install `HUnit`, a unit tester framework:

```
cabal update
cabal list hunit
```

  - note the name you want, e.g. `HUnit` (and not the many other variants/additions for now)

```
cabal install HUnit --lib
```

  - it should automatically download and install dependencies.
  - be sure to include the `--lib` tag to avoid a warning about seeing no executables in the package (lets cabal know this is fine as you'll only use it as library code)

Alternative: rather than make you go all the way through installing another package manager (e.g. `cabal-install` or `stack`), you can just put the source of `HUnit` and `CallStack` adjacent to your code and not mess with any of that.

- Some Extra Support Files - download and unzip this from our class directory, and put the two folders (Data/, Test/) *adjacent* to your source code and our tester. You only need this if you don't have `cabal-install` installed.

1.2 Tester

- Get the tester here: Tester4H.hs
- You can also start with this template file to ensure all definitions are present (though effectively empty). Just remember to change the name to `Homework4.hs`. Homework4_TEMPLATE.hs

Run it this way all at once from the command line:

```
ghci -e "main" Tester4H.hs
```

Or, load it up, and run (a) all tests on all functions, (b) all tests on one function, or (c) one test on one function (by its index in the tester file)

```
demo$ ghci Tester4H.hs
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
Loaded package environment from /Users/mudd/.ghc/x86_64-darwin-8.8.4/environments/default
[1 of 6] Compiling Homework4        ( Homework4.hs, interpreted )
[2 of 6] Compiling Test.HUnit.Lang  ( Test/HUnit/Lang.hs, interpreted )
[3 of 6] Compiling Test.HUnit.Base  ( Test/HUnit/Base.hs, interpreted )
[4 of 6] Compiling Test.HUnit.Text  ( Test/HUnit/Text.hs, interpreted )
[5 of 6] Compiling Test.HUnit       ( Test/HUnit.hs, interpreted )
[6 of 6] Compiling Main             ( Tester4H.hs, interpreted )
Ok, six modules loaded.

-- (a)
*Main> main
Cases: 105  Tried: 105  Errors: 0  Failures: 0
Counts {cases = 105, tried = 105, errors = 0, failures = 0}
```

```
-- (b)
*Main> testFunc test_coprime
Cases: 11  Tried: 11  Errors: 0  Failures: 0
Counts {cases = 11, tried = 11, errors = 0, failures = 0}

-- (c)
*Main> testOne test_coprime 0
Cases: 1  Tried: 1  Errors: 0  Failures: 0
Counts {cases = 1, tried = 1, errors = 0, failures = 0}

*Main> :quit
Leaving GHCi.
demo$
```

1.3 Implementation Notes

- Haskell is a whitespace language, so be sure to use a text editor that knows what Haskell code is. Keep in mind that we'll be doing at least a couple assignments in Haskell, so it's worth your time getting it all set up, more so than other languages this semester.
  - You might use emacs and add the Haskell Mode, but you can choose the working environment you want.
  - You might use VSCode:
    - make sure you have a module declaration at the top of the file.

      ```
      module Whatever where
      ```

    - make sure you're using the "Haskell" plugin e.g. version 1.2.0, and not the legacy plugin.
  - **note** - you should not need a lot of excessive indentations! Ther are many ways you can indent your code, as long as the compiler still knows what you mean. Especially early on, if you have questions about how to avoid lots of indentations, we're happy to help on either private piazza posts or "vanilla" code (things that don't solve the assignment but have the general shape you're working on) in a public post.
- You'll be implementing the same functions as before, in a file named `Homework4.hs`. It must have this specific name for testing purposes, much like a Java class name. Put your own name and netID in a comment at the top. Also, you must include these lines at the top, to make it a proper Haskell module, and to hide the Prelude-defined version of `zipWith` and of `any`. Of course use your own name/netID.

  ```
  -- Name: Georgie Mason
  -- NetID: gmason76

  module Homework4 where
  import Prelude hiding (zipWith,any)
  ```

- Again, **a *major* theme should be writing helper functions** that solve parts of the task. We can also think of the required function as being the driver function that just calls another function that needs more arguments. Turning a loop-style algorithm into recursion can often be thought of as making all variables that are used in the loop become arguments for the recursive function. If you took my advice and wrote a direct and simple algorithm, you already have somthing that can almost entirely port over to Haskell with no extra mental overhead, and you'll only need to deal with recursion vs iteration, and then syntax and typing issues.
- To turn in your work, submit the single file to BlackBoard under the appropriate submission. Since everyone's file is named the same, keep the grader happy by actually putting your name in the file as directed.

2 Functions

2.1 primeFactors

```
primeFactors :: Int -> [Int]
```

given a positive integer, return a list of its prime factors, in increasing order. Include the correct number of occurrences, e.g. `primeFactors 24` is `[2,2,2,3]`.

```
*Homework4> primeFactors 5
[5]
*Homework4> primeFactors 50
[2,5,5]
*Homework4> primeFactors 66
[2,3,11]
*Homework4> primeFactors 463
[463]
*Homework4> primeFactors 512
[2,2,2,2,2,2,2,2,2]
```

2.2 coprime

```
coprime :: Int -> Int -> Bool
```

Given two positive integers, return whether they are co-prime or not. Coprime numbers don't share any divisors other than 1.

```
*Homework4> coprime 7 11
True
*Homework4> coprime 10 21
True
*Homework4> coprime 50 100
False
*Homework4> coprime 367 463
True
*Homework4> coprime 330 463
True
*Homework4> coprime 222 262
False
```

2.3 trib

```
trib :: Int -> Int
```

Given a non-negative integer n, calculate the nth tribonnaci number (where values are the sums of the previous three items in the sequence, or 1 when there aren't enough values ahead of it). For our purposes, the sequence begins as `1,1,1,...` (with indexes 0, 1, 2, …). You need to implement a fast version (meaning O(n)); if your code takes some exponential amount of time to complete, you will not get credit for this function.

```
*Homework4> trib 0
1
*Homework4> trib 1
1
*Homework4> trib 2
```

```
1
*Homework4> trib 3
3
*Homework4> trib 4
5
*Homework4> trib 5
9
*Homework4> trib 6
17
*Homework4> map trib [0..20]
[1,1,1,3,5,9,17,31,57,105,193,355,653,1201,2209,4063,7473,13745,25281,46499,85525]
```

2.4 maxTwo

```
maxTwo :: [Int] -> [Int]
```

Given a list of integers `xs`, find the two largest values and return them in a list (largest first). Largest duplicates should be preserved (included) in the output. Do not modify the original list.

- *Hint*: doing one traversal and no modification is the suggested approach.

```
*Homework4> maxTwo  [1,2,3,4,5,1,2]
[5,4]
*Homework4> maxTwo  [3,6,1,2,6,4]
[6,6]
*Homework4> maxTwo  [-3,-4,-5,-2,-10]
[-2,-3]
```

2.5 reversed

```
reversed :: [a] -> [a]
```

Given a list of ay type of values, create the list whose values are in the opposite order. Do not modify the original list.

```
*Homework4> reversed [1,2,3,4,5]
[5,4,3,2,1]
*Homework4> reversed [4]
[4]
*Homework4> reversed [True, False, False]
[False,False,True]
```

2.6 clockwise

```
clockwise :: [[a]] -> [[a]]
```

Given a list of lists, *assume it is rectangular* (all rows have same length), create and return the list of lists that contains the same values, but rotated clockwise.

```
*Homework4> clockwise [[1,2,3],[4,5,6]]
[[4,1],[5,2],[6,3]]
```

```
*Homework4> clockwise [[1,2,3,4,5]]
[[1],[2],[3],[4],[5]]
*Homework4> clockwise [[1],[2],[3],[4],[5]]
[[5,4,3,2,1]]
```

2.7 any

```
any :: [Bool] -> Bool
```

Given a list of `Bool` values, return `True` if any of them is `True`, and return `False` if every single one of them is `False`. Do not modify the original list.

```
*Homework4> any [False, False, False]
False
*Homework4> any [True, False, True, True]
True
*Homework4> any []
False
```

2.8 select

```
select :: (a->Bool) -> [a] -> [a]
```

Given a predicate function (`::a->Bool`) as well as a list of values (`::[a]`), create a list of all items from the argument list that pass the predicate function. Preserve ordering in your output.

- note: there are built in functions `even`, `odd` (defined in the `Prelude`) that you might want to use for some manual testing (but they of course won't be part of your solution).

```
*Homework4> select even [1,2,3,4,5]
[2,4]
*Homework4> select (\x -> mod x 2 == 1) [1,2,3,4,5]
[1,3,5]
*Homework4> let window x = 5<x && x<10
*Homework4> select window [1..10]
[6,7,8,9]
```

2.9 zipWith

```
zipWith :: (a->b->c) -> [a] -> [b] -> [c]
```

Given a two-argument function and two lists of arguments to supply, create a list of the results of applying the function to each same-indexed pair of arguments from the two lists. (Zip up the two lists with the function). The answer is as long as the shortest of the two input lists. Do not modify the argument lists.

Note: `(+)` is an example of making an infix operator behave like a regular two-argument function. `(+)` means the same thing as `(\x -> \y -> x+y)`.

```
*Homework4> zipWith (+) [1,2,3] [10,20,30]
[11,22,33]
```

```
*Homework4> zipWith (+) [1,2,3] [10,20,30,40,50,60]
[11,22,33]
*Homework4> zipWith (*) [1,2,3,4] [2,4,6,8]
[2,8,18,32]
*Homework4> zipWith (*) [1,2,3,4] []
[]
```

2.10 augdentity

```
augdentity :: Int -> Int -> [[Int]]
```

Given positive ints `r` and `c` indicating number of rows and columns, create a 2D list that represents the "augmented identity matrix" with that dimension: It's the `k x k` identity matrix (where `k = min(r,c)`), and augmented rightwards or downwards as needed with zeroes in order to be of size `r x c`. Stated another way, it's an `r x c` matrix filled with zeroes that has ones along its main diagonal.

```
*Homework4> augdentity 3 3
[[1,0,0],[0,1,0],[0,0,1]]
*Homework4> augdentity 3 5
[[1,0,0,0,0],[0,1,0,0,0],[0,0,1,0,0]]
*Homework4> augdentity 5 3
[[1,0,0],[0,1,0],[0,0,1],[0,0,0],[0,0,0]]
*Homework4> augdentity 2 2
[[1,0],[0,1]]
```