# Homework 4: Advanced Haskell Topics

See Webcourses and the syllabus for due dates.

## Purpose

In this homework you will consolodate your knowledge of functional programming, and get a taste of two advanced techniques: making an instance of a type class and using infinite streams. [UseModels] [Concepts].

## Directions

Answers to English questions should be in your own words; don't just quote text from other sources.
For coding problems, we will take some points off for: code with the wrong type or wrong name, duplicated code, code with extra unnecessary cases, or code that is excessively hard to follow. You should always assume that the inputs given to each function will be well-typed, thus your code should not have extra cases for inputs that are not of the proper type. (Assume that any human inputs are error checked before they reach your code.) Make sure your code has the specified type by including the given type declaration with your code. Avoid duplicating code by using helping functions, library functions (when not prohibited in the problems), or by using syntactic sugars and local definitions (using **`let`** and **`where`**). It is a good idea to check your code for these problems before submitting.
For this homework we suggest that you work individually. (However, per the course's grading policy you can work in a group if you wish, provided that you carefully follow the policy on cooperation, as described in the course's grading policy.)
Don't hesitate to contact the staff if you are stuck at some point.

## What to Turn In

For English answers, please paste your answer into the assignment as a "text answer" in the problem's "assignment" on Webcourses. For a problem with a mix of code and English, follow both of the above.
For each problem that requires code, turn in (on Webcourses@UCF) your code and output of testing with our test cases. Please upload code as a plain (text) file with the name given in the problem or testing file and with the suffix `.hs` or `.lhs` (that is, do *not* give us a Word document or a PDF file for the code). Also paste the output from our tests into the Comment box for that "assignment".
For all Haskell programs, you must run your code with GHC. See the course's Running Haskell page for some help and pointers on getting GHC installed and running. Your code should compile properly (and thus type check); if it doesn't, then you probably should keep working on it. Email the staff with your code file if you need help getting it to compile or have trouble understanding error messages. If you don't have time to get your code to compile, at least tell us that you didn't get it to compile in your submission.
You are encouraged to use any helping functions you wish, and to use Haskell library functions, unless the problem specifically prohibits that.

## What to Read

You can read chapters 8, 13, 17, and 18 of the recommended textbook on Haskell [Tho11]. For more about Monads you may want to read Wadler's paper "Monads for Functional Programming" [Wad95] or Noel Witstanley's paper "What the hell are Monads?". Both of these are accessible from the course syllabus.
In the tutorial *Learn You a Haskell for Great Good!* chapter 8 covers type classes.
See also the course code examples page.

# Problems

## Type Classes and Instances

1. (5 points) [Concepts] User-defined **data** types in Haskell are not, by default, instances of the type class Show. An example is the type defined in the following module

   ```
   module Fraction where
   data Fraction = Integer :/ Integer
   ```

   Thus when you try to evaluate an expression of such a type in the interpreter, you see something like the following:

   ```
   *Fraction> 3 :/ 4

   <interactive>:5:1:
       No instance for (Show Fraction) arising from a use of ‘print’
       Possible fix: add an instance declaration for (Show Fraction)
       In a stmt of an interactive GHCi command: print it
   ```

   Is it a sensible language design that such user-defined types are not automatically instances of the Show class? Briefly explain your answer.

2. (5 points) [Concepts] User-defined **data** types, such as Fraction above, are also not automatically instances of the type class Eq in Haskell. Is it a sensible design for Haskell to not make user-defined types automatically instances of the Eq class? Briefly explain your answer.

3. (15 points) [UseModels] This question concerns writing instances of type classes.

   Consider the module Matrix from homework 3.

   Your task in this problem is to write a module MatrixShow that makes the type (Matrix a) an instance of the standard type class Show, for all types a that are themselves instances of the type class Show. To define an instance of Show, you will have to define the function show inside an instance declaration of the following form. (Your code would follow the code shown below.)

   ```
   module MatrixShow where
   import Matrix
   instance (Show a) => Show (Matrix a) where
   ```

   For a value mat of type (Matrix a), (show mat) should return a String that consists of $m$ lines, one for each $1 \leq i \leq m$, where $m$ is (numRows mat). Each of these lines shows one row of mat. A line showing the $i$th row, consists of a string that shows each of the $(i, j)$th elements of mat, for $1 \leq j \leq n$, each element of which is followed by a blank, and with the entire line ending with a newline (\n).

   There are test cases contained in the file MatrixShowTests.hs, which is shown in Figure 1. As usual, to run our tests, use the main function in the MatrixShowTests.hs file. To make that work, you have to put your code in a module named MatrixShow.

   As specified on the first page of this homework, turn in both your code file and the output of your testing. (The code file should be uploaded to Webcourses, and the test output should be pasted in to the comments box for that assignment.)

```
-- $Id: MatrixShowTests.hs,v 1.2 2013/10/08 02:11:06 leavens Exp $
module MatrixShowTests where
import Matrix
import MatrixShow
import Control.Monad (forM_)
import Testing
main :: IO ()
main = do heading
          run_tests tests
heading :: IO ()
heading = do startTesting "MatrixShowsTests $Revision: 1.2 $"
debugIt :: IO () -- do debugIt if you want to see how matrices look printed
debugIt = do startTesting "MatrixShowTests debugging $Revision: 1.2 $"
             forM_ matrices (\(title, mat) -> do putStrLn title
                                                 print mat)
matrices :: [(String, Matrix Int)]
matrices =
    [("mat11", fillWith (1,1) 75)
    ,("mat43", fromRule (4,3) (\(i,j) -> 10*i+j))
    ,("mat26", fromRule (2,6) (\(i,j) -> 100*i*i + j))
    ,("mat53", fromRule (5,3) (\(i,j) -> (sine i) + (cosine j)))
    ]
mbv :: Maybe b -> b
mbv mb = case mb of
           Just x -> x
           Nothing -> undefined
run2string :: String -> String
run2string title = show (mbv (lookup title matrices))
tests :: [TestCase String]
tests = [eqTest (show (fromRule (2,3) (\(i,j) -> (i,j))))
         "==" (   "(1,1) (1,2) (1,3) \n"
               ++ "(2,1) (2,2) (2,3) \n")
        ,eqTest (run2string "mat11")
         "==" ("75 \n")
        ,eqTest (run2string "mat43")
         "==" (   "11 12 13 \n"
               ++ "21 22 23 \n"
               ++ "31 32 33 \n"
               ++ "41 42 43 \n")
        ,eqTest (run2string "mat26")
         "==" (   "101 102 103 104 105 106 \n"
               ++ "401 402 403 404 405 406 \n")
        ,eqTest (run2string "mat53")
         "==" (   "13 4 -1 \n"
               ++ "14 5 0 \n"
               ++ "6 -3 -8 \n"
               ++ "-2 -11 -16 \n"
               ++ "-4 -13 -18 \n")
        ]
-- helpers for testing below, NOT something you have to implement
sine :: Int -> Int
sine x = truncate (10 * (sin (fromIntegral x)))
cosine :: Int -> Int
cosine x = truncate (10 * (cos (fromIntegral x)))
```

Figure 1: Tests for problem 3.

## Infinite Stream Programming

In this section you will explore programming with infinite streams. These are essentially lists that are not finite, and for which `[]` is not a possible value. To emphasize this, we use the type synonym `IStream`, declared in the module `IStream`.

```
-- $Id: IStream.hs,v 1.1 2013/10/09 21:12:54 leavens Exp leavens $
module IStream where
type IStream a = [a] -- with the understanding that [] is not an element!
```

4. (5 points) [Concepts] [MapToLanguages]

   In Java (or C#) can one write an Iterator that acts like an infinite stream in Haskell and allows the program to work with a finite initial portion of a potentially unbounded sequence of elements (such as all the integers) without going into an infinite loop? Answer "yes" or "no" and then briefly explain your answer.

5. (9 points) [Concepts]

   This is a problem about lazy evaluation in Haskell. Suppose `x` is a Haskell variable of declared type `Integer`. For example, we might have defined `x` by the following definition.

   ```
   x = expensiveComputation ()
   ```

   Which of the following expressions, when executed, will cause Haskell to execute `expensiveComputation ()` and fully determine the value of `x`? (Tell us all the correct answers, there may be more than one.)

   A. `x + 0`

   B. `let z=x in z`

   C. `x == 4020`

   D. `((\_ -> 4020) x)`

   E. `if x > 4020 then x else x`

6. (10 points) [UseModels] In Haskell, write the function

   ```
   imerge :: (Ord t) => (IStream t) -> (IStream t) -> (IStream t)
   ```

   that takes two infinite stream of some `Ord` type `t`, both of which are in non-decreasing order, and returns an infinite stream containing just the elements in the argument streams that is also in non-decreasing order. The order of the elements in the result is such that each $e$ in one of the input streams appears in the result before any element that is larger than $e$ appears in the result.

   There are examples in Figure 2 on the following page.

7. (10 points) [UseModels] Write a function

   ```
   repeatingListOf :: [t] -> (IStream t)
   ```

   that, for any type `t` takes a non-empty finite list `elements` of elements of type `t` returns an infinite stream whose elements repeat the individual elements of `elements` endlessly. See Figure 3 on the following page for examples.

```
-- $Id: IMergeTests.hs,v 1.1 2013/10/09 21:12:54 leavens Exp leavens $
module IMergeTests where
import IMerge
import Testing
main :: IO()
main = dotests "IMergeTests $Revision: 1.1 $" tests
tests :: [TestCase [Integer]]
tests =
    [eqTest (take 10 (imerge [0,2 ..] [1,3 ..])) "==" [0,1,2,3,4,5,6,7,8,9]
    ,eqTest (take 15 (imerge [-1,1 ..] [4,8 ..]))
     "==" [-1,1,3,4,5,7,8,9,11,12,13,15,16,17,19]
    ,eqTest (take 16 (imerge (map (\x -> x*x) [0,2 ..])
                             (map (\x -> x^3) [-3,-1 ..])))
     "==" [-27,-1,0,1,4,16,27,36,64,100,125,144,196,256,324,343]
    ,eqTest (take 17 (imerge [1,1 ..] [3,3 ..]))
     "==" [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
    ,eqTest (take 17 (imerge [4,4 ..] ([2,2,2]++[3,3 ..])))
     "==" [2,2,2,3,3,3,3,3,3,3,3,3,3,3,3,3,3]
    ]
```

Figure 2: Tests for Problem 6 on the previous page.

```
-- $Id: RepeatingListOfTests.hs,v 1.1 2013/10/09 21:12:54 leavens Exp leavens $
module RepeatingListOfTests where
import RepeatingListOf
import Testing
main :: IO()
main = dotests "RepeatingListOfTests $Revision: 1.1 $" tests
tests :: [TestCase [Integer]]
tests =
    [eqTest (take 10 (repeatingListOf [1])) "==" [1,1,1,1,1,1,1,1,1,1]
    ,eqTest (take 10 (drop 5000 (repeatingListOf [7])))
     "==" [7,7,7,7,7,7,7,7,7,7]
    ,eqTest (take 11 (drop 50000 (repeatingListOf [2,3,4,5,6])))
     "==" [2,3,4,5,6,2,3,4,5,6,2]
    ,eqTest (take 13 (repeatingListOf [9,22,-1,3,7]))
     "==" [9,22,-1,3,7,9,22,-1,3,7,9,22,-1]
    ,eqTest (take 17 (repeatingListOf [4,0,2,0,5,0,2,1]))
     "==" [4,0,2,0,5,0,2,1,4,0,2,0,5,0,2,1,4]
    ]
```

Figure 3: Tests for Problem 7 on the previous page.

8. (15 points) [UseModels] In Haskell, write a function

```
blockIStream :: Int -> (IStream Char) -> (IStream [Char])
```

that takes an Int, `blockSize`, an infinite stream of characters, `istrm`, and which returns an infinite stream of lists of characters, where each list in the result has exactly `blockSize` elements. The result consists of the first `blockSize` elements of `istrm`, followed by a list containing the next `blockSize` elements of `istrm`, and so on. That is, a call to `blockIStream` chunks the characters in `istrm` into strings of length `blockSize`. (Hint: in your solution, you may want to use the built-in Haskell functions `take` and `drop`.)

See Figure 4 for our tests.

```
-- $Id: BlockIStreamTests.hs,v 1.1 2013/10/09 21:12:54 leavens Exp leavens $
module BlockIStreamTests where
import BlockIStream
import RepeatingListOf -- from previous problem, used to make tests
import Testing
main :: IO()
main = dotests "BlockIStreamTests $Revision: 1.1 $" tests
tests :: [TestCase [[Char]]]
tests =
    [eqTest (take 8 (blockIStream 3 (from 'a')))
     "==" ["abc","def","ghi","jkl","mno","pqr","stu","vwx"]
    ,eqTest (take 13 (blockIStream 1 nowIsTheTime))
     "==" ["N","o","w"," ","i","s"," ","t","h","e"," ","t","i"]
    ,eqTest (take 13 (blockIStream 2 nowIsTheTime))
     "==" ["No","w ","is"," t","he"," t","im","e ","fo","r ","al","l ","go"]
    ,eqTest (take 10 (blockIStream 4 nowIsTheTime))
     "==" ["Now ","is t","he t","ime ","for ","all ","good"," ...","Now ","is t"]
    ,eqTest (take 6 (blockIStream 7
                      (repeatingListOf
                       "When in the course of human events ...")))
     "==" ["When in"," the co","urse of"," human ","events ","...When"]
    ]
-- The following are used for testing only, NOT for you to implement!
from c = c:(from (toEnum ((fromEnum c)+1)))
nowIsTheTime = (repeatingListOf "Now is the time for all good ...")
```

Figure 4: Tests for Problem 8.

9. (15 points) [UseModels] In Haskell, write a function

```
encryptIStream :: Int -> ([Char] -> [Char]) -> (IStream Char) -> (IStream Char)
```

that takes 3 arguments: an Int, `blockSize`, a function from Strings of size blockSize to Strings of size blockSize, `encrypt`, and an infinite stream of characters, `istrm`. The `encryptIStream` function returns an infinite stream of characters that is formed by forming `istrm` into Strings of size `blockSize`, encrypting those blocks using `encrypt`, and then concatenating the results together. Figure 5 shows some examples, written using various (cryptographically poor) encryption functions.

```
-- $Id: EncryptIStreamTests.hs,v 1.1 2013/10/09 21:12:54 leavens Exp leavens $
module EncryptIStreamTests where
import IStream
import EncryptIStream
import RepeatingListOf -- from previous problem, used to make tests
import Testing
main :: IO()
main = dotests "EncryptIStreamTests $Revision: 1.1 $" tests
tests :: [TestCase [Char]]
tests =
    [eqTest (take 24 (runTest 3 noEncryption (from 'a')))
     "==" "abcdefghijklmnopqrstuvwx"
    ,eqTest (take 26 (runTest 2 reverseNoEncryption nowIsTheTime))
     "==" "oN wsit eht mi eof rla log"
    ,eqTest (take 40 (runTest 4 (ceaserCypher 1) nowIsTheTime))
     "==" "Opx!jt!uif!ujnf!gps!bmm!hppe!///Opx!jt!u"
    ,eqTest (take 48 (runTest 8 (reverseCeaserCypher 3)
                      (repeatingListOf
                       "When in the course of human events ...")))
     "==" "#ql#qhkZuxrf#hkwxk#ir#hvqhyh#qdpkZ111#vwkw#ql#qh"
    ]
-- The following are used for testing only, NOT for you to implement!
runTest :: Int -> ([Char] -> [Char]) -> (IStream Char) -> (IStream Char)
runTest n encrypt istrm = encryptIStream n (wrap encrypt) istrm
    where wrap f str = if length str == n
                       then f str
                       else error "wrong block size passed to encryption function!"
noEncryption = id
reverseNoEncryption = reverse
charSize = (fromEnum (maxBound :: Char))+1
ceaserCypher :: Int -> String -> String
ceaserCypher n str =
    map (\c -> toEnum (((fromEnum c)+n) `mod` charSize)) str
reverseCeaserCypher n str = reverse (ceaserCypher n str)
from c = c:(from (toEnum (((fromEnum c)+1) `mod` charSize)))
nowIsTheTime = (repeatingListOf "Now is the time for all good ...")
```

Figure 5: Tests for Problem 9.

# Points

This homework's total points: 89.

# References

[Tho11] Simon Thompson. *Haskell: the craft of functional programming*. Addison-Wesley, Harlow, England, third edition, 2011.

[Wad95] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming, Proceedings of the Baastad Spring School*, volume 925 of *Lecture Notes in Computer Science*, New York, NY, May 1995. Springer-Verlag.