Homework 5: Advanced Haskell Topics

See Webcourses2 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 tyep class and using declarative parallelism [UseModels] [Concepts].

Directions

Answers to English questions should be in your own words; don't just quote text from other sources.
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. 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 carefully follow the policy on cooperation 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 Webcourses2) 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

In the tutorial *Learn You a Haskell for Great Good!* chapter 8 covers type classes.
See Simon Marlow's "Parallel and Concurrent Programming in Haskell," for more about parallel programming in Haskell.
Use the Haskell 2010 Report as a guide to the details of Haskell.
See also the course code examples page (and the course resources page).

Problems

Type Classes and Instances

1. (5 points) [Concepts] Function types (such as `(a -> b)` or `(Int -> Bool)`) in Haskell are not instances of the type class `Eq`. Thus when you try to evaluate an `==` expression where the expressions on either side have a function type, you see something like the following:

```
Prelude> (\x -> x+1) == (+1)

<interactive>:4:13:
    No instance for (Eq (a0 -> a0))
      arising from a use of ‘==’
    Possible fix: add an instance declaration for (Eq (a0 -> a0))
    In the expression: (\ x -> x + 1) == (+ 1)
    In an equation for ‘it’: it = (\ x -> x + 1) == (+ 1)
```

   Why is it that function types are not instances of the `Eq` class? Briefly explain your answer.

2. (5 points) [Concepts] Function types are also not instances of the type class `Show` in Haskell. Consider the following interaction with the interpreter:

```
Prelude> (\y -> y*2)

<interactive>:5:1:
    No instance for (Show (a0 -> a0))
      arising from a use of ‘print’
    Possible fix: add an instance declaration for (Show (a0 -> a0))
    In a stmt of an interactive GHCi command: print it
```

   Is it a sensible design for Haskell to not make function types an instance of the `Show` class? Briefly explain your answer.

3. (30 points) [UseModels] This question concerns writing instances of type classes.

   Consider the module GraphDisplay below. This module defines and exports two data types: Graph and Display. These data types are both record types that have a function field.

```
-- $Id: GraphDisplay.hs,v 1.2 2013/03/24 09:40:54 leavens Exp $
module GraphDisplay where
data Graph = Section {size :: Int, center :: (Int,Int),
                      fun :: (Int -> Int)}
data Display = Grid {delta :: Int, middle :: (Int,Int),
                     points :: ((Int,Int) -> Char)}

point = '*'
nothing = ' '

graph2display :: Graph -> Display
graph2display (Section {size = n, center = (x,y), fun = f}) =
    Grid {delta = n `div` 2, middle = (x,y),
          points = (\(i,j) -> if f i == j then point else nothing)}
```

   Your task in this problem is to write a module GraphDisplayInstance that makes the type Display an instance of the standard 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 GraphDisplayInstance where
import GraphDisplay
```

   For a Display of the form (Grid {delta = d, middle = (x,y), points = g}), show should return a String that consists of a newline character (\n) followed by $2 \times \mathtt{d} + 1$ lines, one for each j in the range y+d down to y-d, and then 2 lines that display the x coordinate scale. For each j in the range y+d down to y-d, the corresponding line consists of (from left to right):

   - either two blanks, or, for the row corresponding to y, the characters "y " (y followed by a blank),
   - the characters of the numeral j, padded to the left with blanks so that they are the same size (in number of characters) as all the other such numeric labels,
   - a character for each i in the range [(x-d) .. (x+d)], which is the result of applying g to the pair (i, j).
   - a newline character.

   After these lines for each j, the result string contains 2 lines that display the x coordinate scale. The first of these two lines consists of:

   - 2 blanks, and then
   - the characters of the numeral i, for each i in the list [(x-d), ((x-d)+xsize) .. (x+d)], where xsize is the maximum size of the characters in the numbers that represent [(x-d) .. (x+d)]. These numerals should be padded on hte left, so that the rightmost digit of the number lines up in the column corresponding to i.
   - a newline character,
   - Blanks followed by the string "x\n", so that the letter "x" appears in the column corresponding to the number x.

   There are test cases contained in the file GraphDisplayInstanceTests.hs, which is shown in Figures 1 to 4. As usual, to run our tests, use the GraphDisplayInstanceTests.hs file. To make that work, you have to put your code in a module GraphDisplayInstance, which will need to be in a file named GraphDisplayInstance.hs (or the same with a .lhs suffix), in the same directory as our testing file and Testing.lhs.

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 Webcourses2, and the test output should be pasted in to the comments box for that assignment.)

```
-- $Id: GraphDisplayInstanceTests.hs,v 1.4 2013/03/31 21:14:30 leavens Exp $
module GraphDisplayInstanceTests where
import GraphDisplayInstance
import GraphDisplay
import Control.Monad (forM_)
import Testing

main :: IO ()
main = do heading
          run_tests tests

heading :: IO ()
heading = do startTesting "GraphDisplayInstancesTests $Revision: 1.4 $"

debugIt :: IO ()
debugIt = do forM_ graphs (\(title, gr) -> do putStr title
                                              print (graph2display gr))

graphs :: [(String,Graph)]
graphs = [("y=3", Section {size = 4, center = (0,3), fun = (\_ -> 3)})
         ,("y=x", Section {size = 4, center = (0,0), fun = id})
         ,("y=x+3", Section {size = 10, center = (-3,0), fun = (\x -> x+3)})
         ,("y=square x", Section {size = 24, center = (0,9), fun = square})
         ,("y=cube x", Section {size = 30, center = (6,6), fun = cube})
         ,("y=sine x", Section {size = 20, center = (0,0), fun = sine})
         ]

mbv :: Maybe b -> b
mbv mb = case mb of
           Just x -> x
           Nothing -> undefined

run2string :: String -> String
run2string title = show (graph2display (mbv (lookup title graphs)))

tests :: [TestCase String]
tests = [eqTest (run2string "y=3")
           "==" ("\n  5     \n"
                 ++"  4     \n"
                 ++"y 3*****\n"
                 ++"  2     \n"
                 ++"  1     \n"
                 ++"  -2 0 2\n"
                 ++"     x\n")
        ,eqTest (run2string "y=x")
         "==" ("\n   2    *\n"
               ++"   1   * \n"
               ++"y  0  *  \n"
               ++"  -1 *   \n"
               ++"  -2*    \n"
               ++"   -2 0 2\n"
               ++"      x\n")
```

Figure 1: Tests for problem 3, part 1 of 4.

```
,eqTest (run2string "y=x+3")
 "==" ("\n   5           *\n"
       ++"   4          * \n"
       ++"   3         *  \n"
       ++"   2        *   \n"
       ++"   1       *    \n"
       ++"y  0      *     \n"
       ++"  -1     *      \n"
       ++"  -2    *       \n"
       ++"  -3   *        \n"
       ++"  -4  *         \n"
       ++"  -5*           \n"
       ++"   -8-6-4-2 0 2\n"
       ++"          x\n")
,eqTest (run2string "y=square x")
 "==" ("\n  21                         \n"
       ++"  20                         \n"
       ++"  19                         \n"
       ++"  18                         \n"
       ++"  17                         \n"
       ++"  16         *       *       \n"
       ++"  15                         \n"
       ++"  14                         \n"
       ++"  13                         \n"
       ++"  12                         \n"
       ++"  11                         \n"
       ++"  10                         \n"
       ++"y  9          *     *         \n"
       ++"   8                         \n"
       ++"   7                         \n"
       ++"   6                         \n"
       ++"   5                         \n"
       ++"   4           *   *         \n"
       ++"   3                         \n"
       ++"   2                         \n"
       ++"   1            * *          \n"
       ++"   0             *           \n"
       ++"  -1                         \n"
       ++"  -2                         \n"
       ++"  -3                         \n"
       ++"  -12 -9 -6 -3  0  3  6  9 12\n"
       ++"                x\n")
```

Figure 2: Tests for problem 3, continued, part 2 of 4.

```
,eqTest (run2string "y=cube x")
 "==" ("\n  21                              \n"
       ++"  20                              \n"
       ++"  19                              \n"
       ++"  18                              \n"
       ++"  17                              \n"
       ++"  16                              \n"
       ++"  15                              \n"
       ++"  14                              \n"
       ++"  13                              \n"
       ++"  12                              \n"
       ++"  11                              \n"
       ++"  10                              \n"
       ++"   9                              \n"
       ++"   8           *                  \n"
       ++"   7                              \n"
       ++"y  6                              \n"
       ++"   5                              \n"
       ++"   4                              \n"
       ++"   3                              \n"
       ++"   2                              \n"
       ++"   1          *                   \n"
       ++"   0         *                    \n"
       ++"  -1        *                     \n"
       ++"  -2                              \n"
       ++"  -3                              \n"
       ++"  -4                              \n"
       ++"  -5                              \n"
       ++"  -6                              \n"
       ++"  -7                              \n"
       ++"  -8       *                      \n"
       ++"  -9                              \n"
       ++"   -9-7-5-3-1 1 3 5 7 9111315171921\n"
       ++"                   x\n")
,eqTest (run2string "y=sine x")
 "==" ("\n   10                       \n"
       ++"    9     *       *     *  \n"
       ++"    8           *          \n"
       ++"    7      *               \n"
       ++"    6                 *    \n"
       ++"    5*                     \n"
       ++"    4                   * \n"
       ++"    3                      \n"
       ++"    2    *                 \n"
       ++"    1             *        \n"
       ++"y   0          *           \n"
       ++"   -1       *              \n"
       ++"   -2                *     \n"
       ++"   -3                      \n"
       ++"   -4 *                    \n"
       ++"   -5                    *\n"
       ++"   -6   *                  \n"
       ++"   -7              *       \n"
       ++"   -8         *            \n"
       ++"   -9  *     *       *     \n"
       ++"  -10                      \n"
       ++"   -10 -7 -4 -1  2  5  8\n"
       ++"               x\n")          ]
```

Figure 3: Tests for problem 3, continued, part 3 of 4.

```haskell
-- helpers for testing below, NOT something you have to implement
square x = x^2
cube x = x^3
sine :: Int -> Int
sine x = truncate (10 * (sin (fromIntegral x)))
-- end of helpers for testing
```

Figure 4: Tests for problem 3, part 4 of 4.

Parallel Programming

4. (25 points) [UseModels] [Concepts] Write, in Haskell, a function

   ```
   parSort :: Ord a => [a] -> [a]
   ```

   that takes a list and sorts it (in non-decreasing order). Your implementation should use the parallel processing facilities in Haskell, so that it will run faster on a multicore machine. There are test cases shown in Figure 5.

```
-- $Id: ParSortTests.hs,v 1.2 2013/04/08 15:22:52 leavens Exp leavens $
module ParSortTests where
import ParSort
import Testing

main :: IO ()
main = dotests "ParSortTests $Revision: 1.2 $" tests

tests :: [TestCase String]
tests =
    [eqTest (parSort "") "==" ""
    ,eqTest (parSort "defbac") "==" "abcdef"
    ,eqTest (parSort "The quick brown fox jumped over the lazy dogs.")
            "==" "        .Tabcddeeeefghhijklmnoooopqrrstuuvwxyz"
    ,eqTest (parSort "I sought, and soon discovered, the three headstones on the slope")
            "==" "          ,,Iaacdddeeeeeeeeeghhhhhilnnnnooooooprrsssssttttttuv"
    ]
```

Figure 5: Tests for problem 4.

   Note that you will only get full credit if your code is written so that it will run in parallel.

5. (0 points) [Concepts] In this optional problem, if you have a multicore computer, measure the speedup of your code on the task of sorting the words in the novel *Wuthering Heights*, which is found in the file `wuthering_heights.txt` included in the tests.

   Post to the discussion on webcourses2 for this problem your speedup and how many cores you used.

   Hint: try the measurements using the code in `ParSortMain.hs` (see Figure 6 on the next page), which has comments to tell you how to compile it and get detailed output about the sparks used.

Points

This homework's total points: 65.

References

[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.

```haskell
-- $Id: ParSortMain.hs,v 1.2 2013/03/31 20:53:49 leavens Exp $
-- compile with:
--    ghc -O2 ParSortMain.hs -threaded -rtsopts -eventlog
-- run with:
--    ./ParSortMain +RTS -N -s -ls
module Main where
import ParSort
import Control.Monad (forM_)
import System.IO

main = do putStrLn "starting tests..."
          fh <- openFile "wuthering_heights.txt" ReadMode
          book <- hGetContents fh
          let sorted = parSort (words book)
          forM_ sorted putStrLn
          putStrLn "... done with tests"
```

Figure 6: Main program for measuring `parSort`.