Homework 3: Advanced Functional Programming

See Webcourses and the syllabus for due dates.

Purpose

In this homework you will learn more advanced techniques of functional programming such as recursion over more interesting grammars using higher-order functions to abstract from programming patterns, and using higher-order functions to model infinite data [UseModels] [Concepts]. Many of the problems exhibit polymorphism [UseModels] [Concepts]. The problems as a whole illustrate how functional languages work without hidden effects [EvaluateModels].

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. (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. You can avoid duplicating code by using: helping functions, library functions (when not prohibited in the problems), and syntactic sugars and local definitions (using **`let`** and **`where`**). It is a good idea to check your code for duplicated code before submitting.
Since the purpose of this homework is to ensure skills in functional programming, 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 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 English answers, please paste your answer into the assignment as a "text answer" in the problem's "assignment" on Webcourses.
For each problem that requires code, turn in (on Webcourses) 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 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 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

Besides reading chapters 11-17 of the recommended textbook on Haskell [Tho11], you may want to read some of the Haskell tutorials. 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

Recursion over Grammars

See the "Following the Grammar with Haskell" [Lea13] document for examples and hints about the problems in this section.

1. (20 points) [UseModels] This problem is about the type `WindowLayout`, which is defined in the file `WindowLayout.hs`.

```
-- $Id: WindowLayout.hs,v 1.1 2015/02/12 04:23:12 leavens Exp leavens $
module WindowLayout where
data WindowLayout = Window {wname :: String, width :: Int, height :: Int}
                  | Horizontal [WindowLayout]
                  | Vertical [WindowLayout]
                  deriving (Show, Eq)
```

   In Haskell, write a function

   `totalHeight :: WindowLayout -> Int`

   that takes a ⟨WindowLayout⟩, `wl`, and returns the total height of the window layout. The height is defined by cases as follows. The height of a ⟨WindowLayout⟩ of form

   `Window {wname =` nm`, width =` w`, height =` h`}`

   is h. The height of a ⟨WindowLayout⟩ of the form

   `Horizontal [`wl_1`, ...,` wl_m`]`

   is 0 if the list is empty, and otherwise is the maximum of the heights of wl_1 through wl_m (inclusive). The height of a ⟨WindowLayout⟩ of the form

   `Vertical [`wl_1`, ...,` wl_n`]`

   is the sum of the heights of wl_1 through wl_m (inclusive), which is 0 if the list is empty.

   The file `TotalHeightTests.hs` contains tests that show how the function should work, see Figure 1 on the following page.

   Be sure to follow the grammar! In particular, you need to use some helping function to work on the lists that are part of the ⟨WindowLayout⟩ grammar. We will take off points if you do not follow the grammar (and you will spend more time trying to get your code to work).

```
-- $Id: TotalHeightTests.hs,v 1.1 2015/02/12 04:23:12 leavens Exp leavens $
module TotalHeightTests where
import Testing
import WindowLayout
import TotalHeight

main = dotests "TotalHeightTests $Revision: 1.1 $" tests

tests :: [TestCase Int]
tests =
 [(eqTest (totalHeight (Window {wname = "olympics", width = 50, height = 33}))
          "==" 33)
 ,(eqTest (totalHeight (Horizontal [])) "==" 0)
 ,(eqTest (totalHeight (Vertical [])) "==" 0)
 ,(eqTest (totalHeight
            (Horizontal [(Window {wname = "olympics", width = 80, height = 33})
                        ,(Window {wname = "news", width = 20, height = 10})]))
   "==" 33)
 ,(eqTest (totalHeight
           (Vertical [(Window {wname = "olympics", width = 80, height = 33})
                     ,(Window {wname = "news", width = 20, height = 10})]))
   "==" 43)
 ,(eqTest (totalHeight
           (Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                     ,(Window {wname = "olympics", width = 80, height = 33})
                     ,(Window {wname = "news", width = 20, height = 10})]))
   "==" 143)
 ,(eqTest (totalHeight
         (Horizontal
            [(Vertical [(Window {wname = "Tempest", width = 200, height = 100})
                       ,(Window {wname = "Othello", width = 200, height = 77})
                       ,(Window {wname = "Hamlet", width = 1000, height = 600})])
            ,(Horizontal [(Window {wname = "baseball", width = 50, height = 40})
                         ,(Window {wname = "track", width = 100, height = 60})
                         ,(Window {wname = "golf", width = 70, height = 30})])
            ,(Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                       ,(Window {wname = "olympics", width = 80, height = 33})
                       ,(Window {wname = "news", width = 20, height = 10})])
            ]))
        "==" 777)
 ]
```

Figure 1: Tests for problem 1.

2. (20 points) [UseModels]

   This is another problem about Window Layouts. Write a function

   ```
   splitScreen :: String -> WindowLayout -> WindowLayout
   ```

   that takes a string, `name`, and a Window Layout, `wl`, and returns a Window Layout that is just like `wl`, except that for each window in `wl` whose `wname` is (== to) `name` is changed to a horizontal window layout with both windows having the same name and half the width of the previous layout. (Hint, use Haskell's `div` operator to do the division.)

   Figure 2 on the next page shows examples.

   As always, after writing your code, run our tests, and turn in your solution and the output of our tests.

```
-- $Id: SplitScreenTests.hs,v 1.2 2015/02/12 04:23:12 leavens Exp leavens $
module SplitScreenTests where
import WindowLayout; import SplitScreen; import Testing
main = dotests "SplitScreenTests $Revision: 1.2 $" tests
tests :: [TestCase WindowLayout]
tests =
 [(eqTest (splitScreen "olympics"
                        (Window {wname = "olympics", width = 50, height = 33}))
   "==" (Horizontal [Window {wname = "olympics", width = 25, height = 33}
                    ,Window {wname = "olympics", width = 25, height = 33}]))
 ,(eqTest (splitScreen "masterpiece" (Horizontal [])) "==" (Horizontal []))
 ,(eqTest (splitScreen "nova" (Vertical [])) "==" (Vertical []))
 ,(eqTest (splitScreen "olympics"
            (Horizontal [(Window {wname = "olympics", width = 79, height = 33})
                        ,(Window {wname = "local news", width = 21, height = 10})]))
   "==" (Horizontal [(Horizontal (let w = Window {wname = "olympics", width = 39, height = 33} in [w, w]))
                    ,(Window {wname = "local news", width = 21, height = 10})]))
 ,(eqTest (splitScreen "local news"
           (Vertical [(Window {wname = "olympics", width = 79, height = 33})
                      ,(Window {wname = "local news", width = 21, height = 10})]))
   "==" (Vertical [(Window {wname = "olympics", width = 79, height = 33})
                  ,(Horizontal (let w = Window {wname = "local news", width = 10, height = 10} in [w, w]))]))
 ,(eqTest (splitScreen "Sienfeld"
           (Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                     ,(Window {wname = "Sienfeld", width = 80, height = 33})
                     ,(Window {wname = "Sienfeld", width = 30, height = 10})]))
   "==" (Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                  ,(Horizontal (let w = Window {wname = "Sienfeld", width = 40, height = 33} in [w,w]))
                  ,(Horizontal (let w = Window {wname = "Sienfeld", width = 15, height = 10} in [w, w]))]))
 ,(eqTest (splitScreen "local news"
           (Horizontal
            [(Vertical [(Window {wname = "Tempest", width = 200, height = 100})
                       ,(Window {wname = "Othello", width = 200, height = 77})
                       ,(Window {wname = "Hamlet", width = 1000, height = 600})])
            ,(Horizontal [(Window {wname = "baseball", width = 50, height = 40})
                         ,(Vertical
                           [(Window {wname = "local news", width = 100, height = 60})
                           ,(Window {wname = "equestrian", width = 70, height = 30})])])
            ,(Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                       ,(Horizontal
                         [(Window {wname = "olympics", width = 80, height = 33})
                         ,(Window {wname = "local news", width = 20, height = 10})])]) ]))
   "==" (Horizontal
         [(Vertical [(Window {wname = "Tempest", width = 200, height = 100})
                    ,(Window {wname = "Othello", width = 200, height = 77})
                    ,(Window {wname = "Hamlet", width = 1000, height = 600})])
         ,(Horizontal [(Window {wname = "baseball", width = 50, height = 40})
                      ,(Vertical
                        [(Horizontal (let w = Window {wname = "local news", width = 50, height = 60}
                                      in [w,w]))
                        ,(Window {wname = "equestrian", width = 70, height = 30})])])
         ,(Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                    ,(Horizontal
                      [(Window {wname = "olympics", width = 80, height = 33})
                      ,(Horizontal (let w = Window {wname = "local news", width = 10, height = 10}
                                    in [w,w]))])])  ]))  ]
```

Figure 2: Tests for problem 2.

3. (20 points) [UseModels] The problem uses the types `Statement` and `Expression`, which are found in the file `StatementsExpressions.hs`

```
-- $Id: StatementsExpressions.hs,v 1.1 2015/02/12 04:23:12 leavens Exp leavens $
module StatementsExpressions where

data Statement = ExpStmt Expression
               | AssignStmt String Expression
               | IfStmt Expression Statement
                 deriving (Eq, Show)

data Expression = VarExp String
    | NumExp Integer
    | EqualsExp Expression Expression
    | BeginExp [Statement] Expression
      deriving (Eq, Show)
```

Write a function

```
improve :: Statement -> Statement
```

that takes a Statement, `stmt`, and returns a Statement just like `stmt`, except that two simplifications are made:

1. Each Statement of the form `(IfStmt (VarExp "true")` s`)` is replaced by a simplified version of s in the output.
2. Each Expression of the form `(BeginExp []` e`)` is replaced by a simplified version of e in the output.

There are test cases contained in `ImproveTests.hs`, which is shown in Figure 3 on the following page.

As always, after writing your code, run our tests, and turn in your solution and the output of our tests as specified on the first page of this homework.

```
-- $Id: ImproveTests.hs,v 1.2 2015/02/12 04:23:12 leavens Exp leavens $
module ImproveTests where
import StatementsExpressions
import Improve
import Testing

main = dotests "ImproveTests $Revision: 1.2 $" tests

tests :: [TestCase Statement]
tests =
 [(eqTest (improve (IfStmt (VarExp "true") (ExpStmt (NumExp 7))))
          "==" (ExpStmt (NumExp 7)))
 ,(eqTest (improve (ExpStmt (BeginExp [] (NumExp 6))))
          "==" (ExpStmt (NumExp 6)))
 ,(eqTest (improve (ExpStmt (NumExp 7))) "==" (ExpStmt (NumExp 7)))
 ,(eqTest (improve (ExpStmt (VarExp "q"))) "==" (ExpStmt (VarExp "q")))
 ,(eqTest (improve (ExpStmt (VarExp "true"))) "==" (ExpStmt (VarExp "true")))
 ,(eqTest (improve (ExpStmt (BeginExp [] (EqualsExp (VarExp "x") (VarExp "x")))))
   "==" (ExpStmt (EqualsExp (VarExp "x") (VarExp "x"))))
 ,(eqTest (improve (AssignStmt "y" (EqualsExp (VarExp "jz") (VarExp "jz"))))
   "==" (AssignStmt "y" (EqualsExp (VarExp "jz") (VarExp "jz"))))
 ,(eqTest (improve (IfStmt (VarExp "true")
                    (AssignStmt "d" (VarExp "true"))))
   "==" (AssignStmt "d" (VarExp "true")))
 ,(eqTest (improve
           (AssignStmt "g"
            (BeginExp [(IfStmt (VarExp "true")
                               (AssignStmt "d" (BeginExp [] (VarExp "true"))))
                       ,(AssignStmt "z" (EqualsExp (VarExp "m")
                                                   (BeginExp [] (VarExp "m"))))]
             (BeginExp [AssignStmt "e" (EqualsExp (VarExp "y") (NumExp 2))
                        ,(IfStmt (VarExp "true") (ExpStmt (NumExp 3)))]
                        (BeginExp [(IfStmt (VarExp "true")) (ExpStmt (NumExp 1))]
                                  (VarExp "true"))))))
   "==" (AssignStmt "g"
         (BeginExp [(AssignStmt "d" (VarExp "true"))
                   ,(AssignStmt "z" (EqualsExp (VarExp "m") (VarExp "m")))]
          (BeginExp [AssignStmt "e" (EqualsExp (VarExp "y") (NumExp 2))
                     ,(ExpStmt (NumExp 3))]
                     (BeginExp [(ExpStmt (NumExp 1))] (VarExp "true"))))))
 ]
```

Figure 3: Tests for Improve.

4. (25 points) [UseModels] Consider the data type of quantified Boolean expressions defined as follows, in the file `QBExp.hs`.

```
-- $Id: QBExp.hs,v 1.1 2015/02/12 04:23:12 leavens Exp leavens $
module QBExp where
data QBExp = Varref String | QBExp `Or` QBExp
           | Not QBExp | Exists String QBExp
             deriving (Eq, Show)
```

   Your task is to write a function

   `freeQBExp :: QBExp -> [String]`

   that takes a `QBExp`, `qbe`, and returns a list containing just the strings that occur as a free variable reference in `qbe`. The following defines what "occurs as a free variable reference" means. A string `s` *occurs as a variable reference* in a `QBExp` if `s` appears in a subexpression of the form `(Varref s)`. Such a string `s` *occurs as a free variable reference* if and only if it occurs as a variable reference in a subexpression that is outside of any expression of the form `(Exists s e)`, which declares `s`.

   In the examples given in Figure 4 on the next page, note that the lists returned by `freeQBExp` should have no duplicates. In the tests, the `setEq` function constructs a test case that considers lists of strings to be equal if they have the same elements (so that the order is not important).

   Don't use tail recursion on this problem! Instead, use separate helping functions to prevent duplicates.

```
-- $Id: FreeQBExpTests.hs,v 1.1 2015/02/12 04:23:12 leavens Exp leavens $
module FreeQBExpTests where
import QBExp
import FreeQBExp
import Testing
main = dotests "FreeQBExpTests $Revision: 1.1 $" tests
tests :: [TestCase [String]]
tests = [setEq (freeQBExp (Varref "x")) "==" ["x"]
        ,setEq (freeQBExp (Not (Varref "y"))) "==" ["y"]
        ,setEq (freeQBExp (Not (Not (Varref "y")))) "==" ["y"]
        ,setEq (freeQBExp ((Varref "x") `Or` (Not (Varref "y")))) "==" ["x","y"]
        ,setEq (freeQBExp ((Not (Varref "y")) `Or` (Varref "x"))) "==" ["y","x"]
        ,setEq (freeQBExp (((Varref "y") `Or` (Varref "x"))
                           `Or` ((Varref "x") `Or` (Varref "y"))))
                    "==" ["y","x"]
        ,setEq (freeQBExp (Exists "y" (Not (Varref "y")))) "==" []
        ,setEq (freeQBExp (Exists "y" ((Not (Varref "y")) `Or` (Varref "z"))))
                    "==" ["z"]
        ,setEq (freeQBExp (Exists "z" (Exists "y" ((Not (Varref "y")) `Or` (Varref "z")))))
                    "==" []
        ,setEq (freeQBExp (Not
                           ((Varref "z")
                            `Or` (Exists "z" (Exists "y" ((Varref "y") `Or` (Varref "z")))))))
                    "==" ["z"]
        ,setEq (freeQBExp (((Varref "z") `Or` (Varref "q"))
                           `Or` (Not (Exists "z" (Exists "y" ((Varref "y") `Or` (Varref "z")))))))
                    "==" ["z","q"]  ]
    where setEq = gTest setEqual
          setEqual los1 los2 = (length los1) == (length los2)
                               && subseteq los1 los2
          subseteq los1 los2 = all (\e -> e `elem` los2) los1
```

Figure 4: Tests for problem 4.

Combinations of Previous Techniques

5. (20 points) [UseModels] In various contests the contestants are awarded places based on some score, and a list of winners is produced. For example, ebird.org maintains lists of the top 100 birders in Florida this year. In such ranked lists, contestants that have the same score are considered tied; for example, if Audrey and Carlos have both seen 187 bird species this year, then they are considered tied, and both are listed as being in (say) 12th place. In this scenario, the next birder, with 186 species, is listed as being in 14th place, as Audrey and Carlos take places 12 and 13 together, even though they are listed as tied for 12th place.

   In this problem you will write an general ranking function

   ```
   rank :: (Ord a) => [a] -> [(Int, a)]
   ```

   which for any type `a` that is an instance of the `Ord` class, takes a list of elements of type `a`, `things`, and returns a list of pairs of `Int`s and `a` elements. The result is sorted (in non-decreasing order) on the `a` elements of `things`, and the `Int` in each pair is the rank of the element in the pair. There are test cases contained in the file `RankTests.hs`, which is shown in Figure 5 on the following page. To run our tests, use the `RankTests.hs` file. To make that work, you have to put your code in a module `Rank`.

   Hint: you can use `sort` from the module `Data.List`. You may also find it helpful to use a helping function so that you can have some additional variables, even if you are not using tail recursion.

```
-- $Id: RankTests.hs,v 1.6 2015/02/19 19:22:59 leavens Exp leavens $
module RankTests where
import Rank
import Testing
main :: IO ()
main = dotests2 "$Revision: 1.6 $" testsString testsBirders
testsString :: [TestCase [(Int, String)]]
testsString =
    [(eqTest (rank []) "==" [])
    ,(eqTest (rank ["one"]) "==" [(1,"one")])
    ,(eqTest (rank ["one","one"]) "==" [(1,"one"),(1,"one")])
    ,(eqTest (rank ["two","one","one"]) "==" [(1,"one"),(1,"one"),(3,"two")])
    ,(eqTest (rank ["abel", "charlie", "baker", "abel", "charlie", "delta", "echo"])
      "==" [(1,"abel"), (1,"abel"), (3,"baker"),
            (4,"charlie"), (4,"charlie"), (6,"delta"), (7,"echo")])
     ,(eqTest (rank ["baker", "baker", "abel", "baker", "baker"])
       "==" [(1,"abel"),(2,"baker"),(2,"baker"),(2,"baker"),(2,"baker")])
    ]
data Birder = Person String Int deriving (Show)
instance Eq Birder where { (Person _ c1) == (Person _ c2) = c1 == c2 }
-- The following Ord instance makes the person with the highest count least
instance Ord Birder where
  (Person _ count1) < (Person _ count2) = (count1 > count2) -- yes, backwards!
  compare (Person _ count1) (Person _ count2) = compare count2 count1
flBirders :: [Birder]
flBirders =  -- data from ebird.org
    [(Person "Audrey" 305),(Person "Graham" 319),(Person "John" 293)
    ,(Person "Scott" 269),(Person "Ron" 269),(Person "Tom" 267),(Person "Thomas" 225)
    ,(Person "Steven & Darcy" 295),(Person "David" 294),(Person "Chris" 312)
    ,(Person "Rangel" 281),(Person "Charles" 280),(Person "Andy" 276)
    ,(Person "Angel & Mariel" 274),(Person "Mark" 273),(Person "Kevin" 270)
    ,(Person "josh" 295),(Person "Jonathan" 290),(Person "adam" 286)
    ,(Person "Gary" 223),(Person "Brian" 257),(Person "Janet" 256)
    ,(Person "Michael" 266),(Person "Steven" 263),(Person "Eric" 261)
    ,(Person "Nancy" 223),(Person "Carlos" 224),(Person "Peter" 225)
    ]
testsBirders :: [TestCase [(Int, Birder)]]
testsBirders =
    [(eqTest (rank []) "==" [])
    ,(eqTest (rank [(Person "Tom" 532),(Person "Pat" 532)]) "=="
                [(1,(Person "Tom" 532)),(1, (Person "Pat" 532))])
    ,(eqTest (rank [(Person "Pat" 532),(Person "Tom" 532)]) "=="
                [(1,(Person "Pat" 532)),(1, (Person "Tom" 532))])
    ,(eqTest (rank [(Person "Pat" 532),(Person "Tom" 532),(Person "Neil" 703)])
      "==" [(1,(Person "Neil" 703)),(2,(Person "Pat" 532)),(2, (Person "Tom" 532))])
    ,(eqTest (rank flBirders)
      "==" [(1,Person "Graham" 319),(2,Person "Chris" 312),(3,Person "Audrey" 305)
           ,(4,Person "Steven & Darcy" 295),(4,Person "josh" 295),(6,Person "David" 294)
           ,(7,Person "John" 293),(8,Person "Jonathan" 290),(9,Person "adam" 286)
           ,(10,Person "Rangel" 281),(11,Person "Charles" 280),(12,Person "Andy" 276)
           ,(13,Person "Angel & Mariel" 274),(14,Person "Mark" 273),(15,Person "Kevin" 270)
           ,(16,Person "Scott" 269),(16,Person "Ron" 269),(18,Person "Tom" 267)
           ,(19,Person "Michael" 266),(20,Person "Steven" 263),(21,Person "Eric" 261)
           ,(22,Person "Brian" 257),(23,Person "Janet" 256),(24,Person "Thomas" 225)
           ,(24,Person "Peter" 225),(26,Person "Carlos" 224)
           ,(27,Person "Gary" 223),(27,Person "Nancy" 223)])    ]
```

Figure 5: Tests for problem 5.

Higher-Order Functions

These problems are intended to give you an idea of how to use and write higher-order functions.

6. (5 points) [UseModels] [Concepts] In cryptography, one would like to apply functions defined over the type `Int` to data of type `Char`. However, in Haskell, these two types are distinct. In Haskell, write a function

   ```
   toCharFun :: (Int -> Int) -> (Char -> Char)
   ```

   that takes a function `f`, of type is `Int -> Int`, and returns a function that operates on characters. In your implementation you can use the `fromEnum` and `toEnum` functions that Haskell provides (found in the `Enum` instance that is built-in for the type `Char`).

   Hint: note that `(fromEnum 'a')` is 97 and `(toEnum 100) :: Char` is `'d'`.

   There are test cases contained in the file `ToCharFunTests.hs`, which is shown in Figure 6.

```
-- $Id: ToCharFunTests.hs,v 1.3 2015/02/12 04:27:19 leavens Exp leavens $
module ToCharFunTests where
import ToCharFun  -- your solution goes in module ToCharFun
import Testing

main = dotests "ToCharFunTests $Revision: 1.3 $" tests
tests :: [TestCase Char]
tests = [eqTest (toCharFun (+3) 'a') "==" 'd'
        ,eqTest (toCharFun (+1) 'b') "==" 'c'
        ,eqTest (toCharFun (+7) 'a') "==" 'h'
        ,eqTest (toCharFun (\c -> 10*c `div` 12) 'h') "==" 'V'
        ]
```

Figure 6: Tests for problem 6.

To run our tests, use the `ToCharFunTests.hs` file. To make that work, you have to put your code in a module `ToCharFun`, which will need to be in a file named `ToCharFun.hs` (or `ToCharFun.lhs`), in the same directory as `ToCharFunTests.hs` 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 Webcourses, and the test output should be pasted in to the comments box for that assignment.)

7. (10 points) [UseModels] [Concepts] Using Haskell's built-in `map` function, write the function

   ```
   mapInside :: (a -> b) -> [[a]] -> [[b]]
   ```

   that for some types `a` and `b` takes a function `f` of type `a -> b`, and a list of lists of type `a`, `lls`, and returns a list of type `[[b]]` that consists of applying f to each element inside each list in `lls`, preserving the structure of the lists.

   There are test cases contained in the file `MapInsideTests.hs`, which is shown in Figure 7 on the following page.

   Note that your code must use `map`. For full credit, write a solution that does not use any pattern matching.

   As specified on the first page of this homework, turn in both your code file and the output of your testing.

```
-- $Id: MapInsideTests.hs,v 1.3 2015/02/12 04:23:12 leavens Exp leavens $
module MapInsideTests where
import Testing
import ToCharFun -- used for testing
import MapInside  -- you have to put your solutions in module MapInside

version = "MapInsideTests $Revision: 1.3 $"

-- do main to run our tests
main :: IO()
main = do startTesting version
          errs_ints <- run_test_list 0 int_tests
          total_errs <- run_test_list errs_ints string_tests
          doneTesting total_errs

int_tests :: [TestCase [[Int]]]
int_tests =
    [eqTest (mapInside (+1) []) "==" []
    ,eqTest (mapInside (+1) [[]]) "==" [[]]
    ,eqTest (mapInside (*2) [[3,4,5],[4,0,2,0],[],[8,7,6]])
      "==" [[6,8,10],[8,0,4,0],[],[16,14,12]]
    ,eqTest (mapInside (*2) [[1 .. 10], [2,4 .. 20], [7]])
      "==" [[2,4 .. 20], [4,8 .. 40], [14]]
    ,eqTest (mapInside (3*) [[0 .. 10], [0,2 .. 10], [7]])
      "==" [[0,3 .. 30], [0,6 .. 30], [21]]
    ,eqTest (mapInside (\n -> 3*n + 1) [[0,7,17,27], [94,5]])
      "==" [[1,22,52,82],[283,16]]
    ]

string_tests :: [TestCase [[Char]]]
string_tests =
    [eqTest (mapInside (toCharFun (+1)) ["A string", "is a list!"])
            "==" ["B!tusjoh","jt!b!mjtu\""]
    ,eqTest (mapInside (toCharFun (\x -> x-7)) ["UCF","CS","is","great"])
            "==" ["N<?","<L","bl","`k^Zm"]
    ,eqTest (mapInside (toCharFun (+7)) ["N<?","<L","bl","`k^Zm"])
            "==" ["UCF","CS","is","great"]
     ]
```

Figure 7: Tests for problem 7.

8. (20 points) [UseModels] [Concepts] Write a higher-order function

```
encryptWith :: (Int -> Int) -> [String] -> [String]
```

that takes a cryptographic function, `f`, and a list of strings `text`, and returns a (poorly) encrypted version of each string in the list by using `toCharFun f` to encrypt each character.

There are test cases contained in the file `EncryptWithTests.hs`, which is shown in Figure 8.

```
-- $Id: EncryptWithTests.hs,v 1.1 2015/02/12 04:23:12 leavens Exp leavens $
module EncryptWithTests where
import EncryptWith  -- your solution goes in this module
import Testing

main = dotests "EncryptWithTests $Revision: 1.1 $" tests
tests :: [TestCase [String]]
tests = [eqTest (encryptWith (rot (+1)) ["a","good","egg"])
         "==" ["b","hppe","fhh"]
        ,eqTest (encryptWith id ["nothing","special","here"])
         "==" ["nothing","special","here"]
        ,eqTest (encryptWith rot13 ["super","bowl","chips","UCF!","Whooo"])
         "==" ["\128\130}r\DEL","o|\132y","puv}\128","bPS.","du|||"]
        ,eqTest (encryptWith rot13 ["Attack","at","dawn!"])
         "==" ["N\129\129npx","n\129","qn\132{."]
        ,eqTest (encryptWith (rot (\x->x-2)) ["Alan","Turing","code","breaker","genius"])
         "==" ["?j_l","Rspgle","ambc","`pc_icp","eclgsq"]
        ,eqTest (encryptWith (rot (\x->x-32)) ["a","very","bad","code","this","is"])
         "==" ["A","VERY","BAD","CODE","THIS","IS"]
        ,eqTest (encryptWith (rot (+32)) ["BEWARE","THE","IDES","OF","MARCH"])
         "==" ["beware","the","ides","of","march"]
        ]
    where rot :: (Int -> Int) -> (Int -> Int)
          rot f = \i -> (f i) `mod` (fromEnum (maxBound :: Char))
          rot13 = rot (+13)
```

Figure 8: Tests for problem 8.

Functions as Data and Abstract Data Types

9. (15 points) [UseModels] Write a function

   `composeList :: [(a -> a)] -> (a -> a)`

   that takes a list of functions, and returns a function which is their composition.

   Hint: note that `composeList []` is the identity function.

   Don't use `last` or `init` in your solution, as that will make your solution $O(n^2)$.

   There are test cases contained in the file `ComposeListTests.hs`, which is shown in Figure 9.

```
-- $Id: ComposeListTests.hs,v 1.3 2015/02/12 04:23:12 leavens Exp leavens $
module ComposeListTests where
import ComposeList
import Testing

main = dotests "ComposeListTests $Revision: 1.3 $" tests

tests :: [TestCase Bool]
tests =
    [assertTrue ((composeList [] [1,2,3]) == [1,2,3])
    ,assertTrue ((composeList [(*5),(+2)] 4) == 30)
    ,assertTrue ((composeList [tail,tail,tail] [1,2,3,4,5]) == [4, 5])
    ,assertTrue ((composeList [(3*),(4+),(10*)] 1) == (3*(4+10)))
    ,assertTrue ((composeList [(\x -> 3:x),(\y -> 4:y)] []) == 3:(4:[]))
    ,assertTrue ((composeList [(\x -> 'a':x),(\y -> ' ':y)] "star") == "a star")
    ,assertTrue ((composeList (map (+) [1 .. 1000]) 0) == (sum [1 .. 1000]))
    ]
```

Figure 9: Tests for problem 9.

---

10. (30 points) [UseModels] In this problem you will write operations for an abstract data type `Matrix`, by writing definitions for the module `Matrix` that completes the module definition in Figure 10.

```
module Matrix (Matrix, fillWith, fromRule, numRows, numColumns,
               at, mtranspose, mmap) where

-- newtype is like "data", but has some efficiency advantages
newtype Matrix a = Mat ((Int,Int), (Int,Int) -> a)

fillWith :: (Int,Int) -> a -> (Matrix a)
fromRule :: (Int,Int) -> ((Int,Int) -> a) -> (Matrix a)
numRows :: (Matrix a) -> Int
numColumns :: (Matrix a) -> Int
at :: (Matrix a) -> (Int, Int) -> a
mtranspose :: (Matrix a) -> (Matrix a)
mmap :: (a -> b) -> (Matrix a) -> (Matrix b)

-- Without changing what is above, implement the above functions.
```

Figure 10: Start of the module `Matrix`.

That is, you are to complete the module by writing a function definition for each function declared in the module. To explain these, note that an $m \times n$ matrix is one with m rows and n columns. Element indexes range from 1 to the number of rows or columns (unlike the convention in C or Haskell). With that convention you are to implement the following functions:

- `fillWith` takes a pair (m, n) and an element e and produces an $m \times n$ matrix each of whose elements are e.
- `fromRule` takes a pair (m, n) and a function g (the rule), and produces an $m \times n$ matrix whose (i, j)th element is $g(i, j)$.
- `numRows` takes an $m \times n$ matrix and returns the number of rows in the matrix, m.
- `numColumns` takes an $m \times n$ matrix and returns the number of columns in the matrix, n.
- `at` takes an $m \times n$ matrix and a pair of `Int`s, (i, j), and returns the (i, j)th element of the matrix, provided that $1 \leq i \leq m$ and $1 \leq j \leq n$. If either index is outside of those ranges, an error occurs (at runtime).
- `mtranspose` takes an $m \times n$ matrix and returns an $n \times m$ matrix where the (i, j)th element of the result is the (j, i)th element of the argument matrix.
- `mmap` takes an $m \times n$ matrix and a function f and returns an $m \times n$ matrix whose (i, j)th element is the result of applying f to the (i, j)th element of the argument matrix.

There are test cases contained in `MatrixTests.hs`, which is shown in Figure 11 on the following page.

To aid in testing, we have also provided code to make `Matrix` an instance of the Haskell type classes `Show` and `Eq`. These instances are found in the file `MatrixInstances.hs`.

To make the tests work, you have to put your code in a module named `Matrix`. As specified on the first page of this homework, turn in both your code file and the output of your testing.

```
-- $Id: MatrixTests.hs,v 1.3 2015/02/12 04:23:12 leavens Exp leavens $
module MatrixTests where
import Matrix
import MatrixInstances
import Testing

main = dotests "MatrixTests $Revision: 1.3 $" tests

-- helpful definitions follow, NOT for you to implement!
allIndexes :: (Int,Int) -> [(Int,Int)]
allIndexes (m,n) = [(i,j) | i <- [1..m], j <- [1..n]]
initial = (fillWith (2,3) "initial")
m10x3 = fillWith (10,3) "u"
m5x7 = fromRule (5,7) (\(i,j) -> show (i,j))
m10ipj = fromRule (5,7) (\(i,j) -> show (10*i+j))

tests :: [TestCase String] -- the actual tests
tests = (map (\(i,j) -> eqTest (initial `at` (i,j)) "==" "initial")
         (allIndexes (2,3)))
        ++ (map (\(i,j) -> eqTest (m10x3 `at` (i,j)) "==" "u")
            (allIndexes (10,3)))
        ++ (map (\(i,j) -> eqTest (m5x7 `at` (i,j)) "==" (show (i,j)))
            (allIndexes (5,7)))
        ++ (map (\(i,j) -> eqTest (m10ipj `at` (i,j)) "==" (show (10*i+j)))
            (allIndexes (5,7)))
        ++ (map (\(i,j) -> eqTest ((mtranspose m5x7) `at` (i,j))
                           "==" (show (j,i)))
            (allIndexes (7,5)))
        ++ (map (\(i,j) -> eqTest ((mmap reverse m10ipj) `at` (i,j))
                           "==" (reverse (show (10*i+j))))
            (allIndexes (5,7)))
```

Figure 11: Tests for problem 10.

11. (20 points) [UseModels] Complete the module definition in Figure 12 below, by defining the functions `sameShape`, `pointwiseApply`, `add`, and `sub`.

```
-- $Id: MatrixAdd.hs,v 1.2 2015/02/12 04:23:12 leavens Exp leavens $
module MatrixAdd where
import Matrix
import MatrixInstances

sameShape :: (Matrix a) -> (Matrix a) -> Bool
pointwiseApply :: (a -> a -> b) -> (Matrix a) -> (Matrix a) -> (Matrix b)
add :: (Num a) => (Matrix a) -> (Matrix a) -> (Matrix a)
sub :: (Num a) => (Matrix a) -> (Matrix a) -> (Matrix a)
```

Figure 12: Beginning of the module `MatrixAdd`, for you to complete.

The predicate `sameShape` takes arguments of type `Matrix a` and returns **`True`** when they have the same dimensions.

The function `pointwiseApply` takes a curried function `op` of two arguments and two matricies, `m1`, and `m2`, which have the same shape and whose elements are the same type as the argument types of `op`, and returns a matrix of the same shape as `m1` and `m2`, in which the `(i,j)`th element is `(m1 `at` (i,j)) `op` (m2 `at` (i,j))`. If the two matrices do not have the same shape, then an error should be raised (using Haskell's `error` function).

The `add` and `sub` operations are the usual pointwise addition and subtraction of matrices.

You must define `add` and `sub` by using `pointwiseApply`.

There are test cases contained in `MatrixAddTests.hs`, which is shown in Figure 13 on the following page.

As always, after writing your code, run our tests, and turn in your solution and the output of our tests as specified on the first page of this homework.

```
-- $Id: MatrixAddTests.hs,v 1.2 2015/02/12 04:23:12 leavens Exp leavens $
module MatrixAddTests where
import Matrix
import MatrixInstances
import MatrixAdd
import Testing

main = dotests "MatrixAddTests $Revision: 1.2 $" tests

-- helpers for testing below, NOT something you have to implement
allIndexes :: (Int,Int) -> [(Int,Int)]
allIndexes (m,n) = [(i,j) | i <- [1..m], j <- [1..n]]
zeros = fillWith (4,3) 0
id4x3 = fromRule (4,3) (\(i,j) -> if i == j then 1 else 0)
m4x3 = fromRule (4,3) (\(i,j) -> 10*i+j)
m9 = fillWith (4,3) 9

tests :: [TestCase (Matrix Int)]
tests =
    [eqTest (zeros `add` id4x3) "==" id4x3
    ,eqTest (m9 `sub` zeros) "==" m9
    ,eqTest (m9 `sub` id4x3)
     "==" (fromRule (4,3) (\(i,j) -> if i == j then 8 else 9))
    ,eqTest (m9 `sub` m4x3)
     "==" (fromRule (4,3) (\(i,j) -> 9 - (10*i+j)))
    ,eqTest (m9 `add` m4x3)
     "==" (fromRule (4,3) (\(i,j) -> 9 + (10*i+j)))
    ,eqTest (m9 `add` m4x3)
     "==" (fromRule (4,3) (\(i,j) -> 9 + (10*i+j)))
    ]
```

Figure 13: Tests for problem 11.

12. (30 points) [Concepts] [UseModels] A set can be described by a "characteristic function" (whose range is **Bool**) that determines if an element occurs in the set. For example, the function ϕ such that $\phi(\text{"coke"}) = \phi(\text{"pepsi"}) = \texttt{True}$ and for all other arguments x, $\phi(x) = \texttt{False}$ is the characteristic function for a set containing the strings "coke" and "pepsi", but nothing else. Allowing the user to construct a set from a characteristic function gives one the power to construct sets that may "contain" an infinite number of elements (such as the set of all prime numbers).

    In a module named `InfSet`, you will declare a polymorphic type constructor `Set`, which can be declared something like as follows:

```
type Set a = ...
-- or perhaps something like --
data Set a = ...
```

    Hint: think about using a function type as part of your representation of sets.

    Then fill in the operations of the module `InfSet`, which are described informally as follows.

    1. The function

       ```
       fromFunc :: (a -> Bool) -> (Set a)
       ```

       takes a characteristic function, f and returns a set such that each value x (of type `a`) is in the set just when fx is `True`.

    2. The function

       ```
       unionSet :: Set a -> Set a -> Set a
       ```

       takes two sets, with characteristic functions f and g, and returns a set such that each value x (of type `a`) is in the set just when either (fx) or (gx) is true.

    3. The function

       ```
       intersectSet :: Set a -> Set a -> Set a
       ```

       takes two sets, with characteristic functions f and g, and returns a set such that each value x (of type `a`) is in the set just when both (fx) and (gx) are true.

    4. The function

       ```
       inSet :: a -> Set a -> Bool
       ```

       tells whether the first argument is a member of the second argument.

    5. The function

       ```
       complementSet :: Set a -> Set a
       ```

       which returns a set that contains everything (of the appropriate type) not in the original set.

    Tests for this are given in the Figure 14 on the next page.

    Note (hint, hint) that the following equations must hold, for all `f`, `g`, and `x` of appropriate types.

```
inSet x (unionSet (fromFunc f) (fromFunc g)) == (f x) || (g x)
inSet x (intersectSet (fromFunc f) (fromFunc g)) == (f x) && (g x)
inSet x (fromFunc f) == f x
inSet x (complementSet (fromFunc f)) == not (f x)
```

```
-- $Id: InfSetTests.hs,v 1.2 2015/02/12 04:23:12 leavens Exp leavens $
module InfSetTests where
import InfSet
import Testing

main = dotests "InfSetTests $Revision: 1.2 $" tests

tests :: [TestCase Bool]
tests =
    [assertTrue (inSet "coke" (fromFunc (\ x -> x == "coke")))
    ,assertFalse (inSet "pepsi" (fromFunc (\ x -> x == "coke")))
    ,assertFalse (inSet "coke" (complementSet (fromFunc (\x -> x == "coke"))))
    ,assertTrue (inSet "oil" (complementSet (fromFunc (\x -> x == "coke"))))
    ,assertTrue (inSet "pepsi" (unionSet (fromFunc (\ x -> x == "coke"))
                                (fromFunc (\ x -> x == "pepsi"))))
    ,assertTrue (inSet "coke" (unionSet (fromFunc (\x -> x == "coke"))
                               (fromFunc (\x -> x == "pepsi"))))
    ,assertFalse (inSet "sprite" (unionSet (fromFunc (\x -> x == "coke"))
                                  (fromFunc (\x -> x == "pepsi"))))
    ,assertFalse (inSet "coke" (intersectSet (fromFunc (\x -> x == "coke"))
                                (fromFunc (\x -> x == "pepsi"))))
    ,assertFalse (inSet "pepsi" (intersectSet (fromFunc (\x -> x == "coke"))
                                (fromFunc (\x -> x == "pepsi"))))
    ,assertTrue (inSet "dr. p" (intersectSet (fromFunc (\x -> "coke" <= x))
                                 (fromFunc (\x -> x <= "pepsi"))))
    ,assertTrue (inSet "pepsi" (intersectSet (fromFunc (\x -> "coke" <= x))
                                (fromFunc (\x -> x <= "pepsi"))))
    ,assertFalse (inSet "beer" (intersectSet (fromFunc (\x -> "coke" <= x))
                                (fromFunc (\x -> x <= "pepsi"))))
    ,assertFalse (inSet "wine" (intersectSet (fromFunc (\x -> "coke" <= x))
                                (fromFunc (\x -> x <= "pepsi"))))
    ,assertTrue (inSet "wine" (unionSet (fromFunc (\x -> "coke" <= x))
                                (fromFunc (\x -> x <= "pepsi"))))
    ]
```

Figure 14: Tests for the module `InfSet`. Recall that `assertTrue e` is equivalent to `eqTest e "==" True`, and `assertFalse e` is equivalent to `eqTest e "==" False`.

Functional Abstractions of Programming Patterns

13. (10 points) [UseModels] [Concepts] Using Haskell's built-in `foldr` function, write the polymorphic function

```
concatMap :: (a -> [b]) -> [a] -> [b]
```

This function can be considered to be an abstraction of problems like `deleteAll`. An application such as `(concatMap f ls)` applies `f` to each element of `ls`, and concatenates the results of those applications together (preserving the order). Note that application of `f` to an element of type `a` returns a list (of type `[b]`), and so the overall process collects the elements of these lists together into a large list of type `[b]`.

Your solution must have the following form:

```
module ConcatMap where
import Prelude hiding (concatMap)
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f ls = foldr ...
```

where the "`...`" is where you will put the arguments to `foldr` in your solution.

Note: your code in `...` must not call `concatMap` (let `foldr` do the recursion).

There are test cases contained in `ConcatMapTests.hs`, which is shown in Figure 15.

```
-- $Id: ConcatMapTests.hs,v 1.4 2015/02/12 04:23:12 leavens Exp leavens $
module ConcatMapTests where
import Prelude hiding (concatMap)
import ConcatMap
import Testing

main :: IO()
main = dotests "ConcatMapTests $Revision: 1.4 $" tests

-- some definitions using concatMap, for testing, not for you to implement
deleteAll toDel ls = concatMap (\e -> if e == toDel then [] else [e]) ls
xerox ls = concatMap (\e -> [e,e]) ls
next3 lst = concatMap (\n -> [n,n+1,n+2]) lst
tests :: [TestCase Bool]
tests =
    [assertTrue ((deleteAll 'c' "abcdefedcba") == "abdefedba")
    ,assertTrue ((deleteAll 3 [3,3,3,7,3,9]) == [7,9])
    ,assertTrue ((deleteAll 3 []) == [])
    ,assertTrue ((xerox "") == "")
    ,assertTrue ((xerox "okay") == "ookkaayy")
    ,assertTrue ((xerox "balon") == "bbaalloonn")
    ,assertTrue ((next3 []) == [])
    ,assertTrue ((next3 [1,2,3]) == [1,2,3,2,3,4,3,4,5])
    ]
```

Figure 15: Tests for problem 13.

---

14. (30 points) [UseModels] [Concepts] In this problem you will write a function

```
foldWindowLayout :: ((String,Int,Int) -> r) -> ([r] -> r) -> ([r] -> r)
                 -> WindowLayout -> r
```

that abstracts from all the `WindowLayout` examples we have seen (such as those earlier in this homework and on the course examples page). For each type `r`, the function `foldWindowLayout` takes 3 functions: `wf`, `hf`, and `vf`, which correspond to the three variants in the grammar for `WindowLayout`. In more detail:

- `wf`, operates on a tuple of the information from a `Window` variant and returns a value of type `r`,
- `hf`, takes a list of the results of mapping `(foldWindowLayout wf hf vf)` over the list in a `Horizontal` variant, and
- `vf`, takes a list of the results of mapping `(foldWindowLayout wf hf vf)` over the list in a `Vertical` variant.

There are test cases contained in `FoldWindowLayoutTests.hs`, which is shown in Figure 16 on the next page and Figure 17 on page 26.

15. [UseModels] [Concepts] Use your `foldWindowLayout` function to implement

(a) (15 points) `totalHeight` from problem 1, and

(b) (15 points) `splitScreen` from problem 2.

Hint: look at the testing file for the previous problem.

Write these solutions into modules `TotalHeight` and `SplitScreen`, respectively, that both import the module `FoldWindowLayout` from the previous problem, and then run the tests appropriate to each of the window layout problems. Hand in your code for these new implementations of `totalHeight` and `splitScreen` and paste the output of both tests for these into the comment box for this "assignment" on webcourses.

Points

This homework's total points: 305.

References

[Lea13] Gary T. Leavens. Following the grammar with Haskell. Technical Report CS-TR-13-01, Dept. of EECS, University of Central Florida, Orlando, FL, 32816-2362, January 2013.

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

```
-- $Id: FoldWindowLayoutTests.hs,v 1.3 2015/02/12 04:23:12 leavens Exp leavens $
module FoldWindowLayoutTests where
import WindowLayout
import FoldWindowLayout
import Testing

main = dotests "FoldWindowLayoutTests $Revision: 1.3 $" tests

-- uses of foldWindowLayout for testing purposes, not for you to implement
watching' = foldWindowLayout (\(wn,_,_) -> [wn]) concat concat
changeChannel new old =
    let changeName new old nm = if nm == old then new else nm
    in foldWindowLayout
           (\(nm,w,h) -> Window {wname = changeName new old nm,
                                 width = w, height = h})
           Horizontal
           Vertical
doubleSize = foldWindowLayout
              (\(wn,w,h) -> Window {wname = wn, width = 2*w, height = 2*h})
              Horizontal
              Vertical
addToSize n = foldWindowLayout
               (\(wn,w,h) -> Window {wname = wn, width = n+w, height = n+h})
               Horizontal
               Vertical
multSize n = foldWindowLayout
               (\(wn,w,h) -> Window {wname = wn, width = n*w, height = n*h})
               Horizontal
               Vertical
totalWidth = foldWindowLayout
               (\(_,w,_) -> w)
               sum
               sum

-- a WindowLayout for testing
hlayout =
    (Horizontal
     [(Vertical [(Window {wname = "Tempest", width = 200, height = 100})
                ,(Window {wname = "Othello", width = 200, height = 77})
                ,(Window {wname = "Hamlet", width = 1000, height = 600})])
     ,(Horizontal [(Window {wname = "baseball", width = 50, height = 40})
                  ,(Window {wname = "track", width = 100, height = 60})
                  ,(Window {wname = "golf", width = 70, height = 30})])
     ,(Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                ,(Window {wname = "olympics", width = 80, height = 33})
                ,(Window {wname = "news", width = 20, height = 10})])])
```

Figure 16: Tests for problem 14, part 1.

```
tests :: [TestCase Bool]
tests =
    [assertTrue ((totalWidth hlayout) == 1760)
    ,assertTrue ((doubleSize hlayout) == (multSize 2 hlayout))
    ,assertTrue ((watching' hlayout)
                 == ["Tempest","Othello","Hamlet","baseball","track","golf",
                     "Star Trek","olympics","news"])
    ,assertTrue
     ((changeChannel
       "pbs" "news"
       (Vertical [(Window {wname = "news", width = 10, height = 5})
                 ,(Window {wname = "golf", width = 50, height = 25})
                 ,(Window {wname = "news", width = 30, height = 70})]))
      ==
      (Vertical [(Window {wname = "pbs", width = 10, height = 5})
                ,(Window {wname = "golf", width = 50, height = 25})
                ,(Window {wname = "pbs", width = 30, height = 70})]))
    ,assertTrue
     ((addToSize 100 hlayout)
      ==
      (Horizontal
       [(Vertical [(Window {wname = "Tempest", width = 300, height = 200})
                  ,(Window {wname = "Othello", width = 300, height = 177})
                  ,(Window {wname = "Hamlet", width = 1100, height = 700})])
       ,(Horizontal [(Window {wname = "baseball", width = 150, height = 140})
                    ,(Window {wname = "track", width = 200, height = 160})
                    ,(Window {wname = "golf", width = 170, height = 130})])
       ,(Vertical [(Window {wname = "Star Trek", width = 140, height = 200})
                  ,(Window {wname = "olympics", width = 180, height = 133})
                  ,(Window {wname = "news", width = 120, height = 110})])]) )
    ,assertTrue
     ((doubleSize hlayout)
      ==
      (Horizontal
       [(Vertical [(Window {wname = "Tempest", width = 400, height = 200})
                  ,(Window {wname = "Othello", width = 400, height = 154})
                  ,(Window {wname = "Hamlet", width = 2000, height = 1200})])
       ,(Horizontal [(Window {wname = "baseball", width = 100, height = 80})
                    ,(Window {wname = "track", width = 200, height = 120})
                    ,(Window {wname = "golf", width = 140, height = 60})])
       ,(Vertical [(Window {wname = "Star Trek", width = 80, height = 200})
                  ,(Window {wname = "olympics", width = 160, height = 66})
                  ,(Window {wname = "news", width = 40, height = 20})])]) )
    ]
```

Figure 17: Tests for problem 14, part 2.