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 a book or other source.
Note that in all 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. It is a good idea to check your code for these problems before submitting. 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`**).
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. (That is, assume that any human-supplied inputs are error checked before they reach 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.
Make sure your code has the specified type by including the given type declaration with your code.
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):

1. Your code, uploaded as a plain (text) file with the name given in the problem and with the suffix `.hs` or `.lhs` (that is, do *not* turn in a Word document or a PDF file for the code).
2. The output of running our tests on your code, which should also be uploaded as a plain text file. To do that, run the `main` function of the test module for the assignment and then copy the output and paste it into a plain text file (with a `.txt` suffix), then upload that file along with your code.

The following is a more detailed explanation of how to run our tests. Suppose you are to write a function named `f` in a module `F`, which would be placed in a file named `F.hs` (or `F.lhs`). In this example we would supply tests for `f` in a module `FTests` (in a file `FTests.hs`). To run our tests, you would open the test module, e.g., by double-clicking on `FTests.hs` in this example, with `ghci` (on Linux or MacOS or in the

Windows command prompt, which is the default on Windows).[1][2] When the test module (`FTests.hs` is loaded, it will load our testing harness module (`Testing.lhs` and other necessary modules, such as `FloatTesting.lhs`, as needed) and your code (in `F.hs`), and by default these are all assumed to be in the same directory as our testing module (so you should be sure they are all in that directory); if you get an error that one of these cannot be found, make sure they are all in the same directory (and readable). After you load these modules enter (at the prompt):

```
main
```

and that will run our tests. Then you can copy the output from our tests and paste it into an appropriate file. Another way to do this (on Linux, MacOS, and cygwin) is to use file redirection, by executing a shell command such as (in our example):

```
ghci FTests.hs > FTests.txt
```

and then typing `main` and `:quit` at the prompts. This will make `FTests.txt` contain the testing output.
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 (preferably pasted into the text of the message) if you need help getting it to compile or have trouble understanding error messages.
We will take a penalty in points if you do not turn in the output of running our tests on a problem that requires writing code. If you don't have time to get your code to type check and run, at least tell us about the problem 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. Read the "Following the Grammar with Haskell" [Lea13] document for recursion over interesting grammars. 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 related to the problems in this section.

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

```haskell
module WindowPlan where
data WindowPlan = Win String Int Int -- name, width, and height
                | Horiz [WindowPlan]
                | Vert [WindowPlan]
                deriving (Show, Eq)
```

[1]If you use `WinGHCi`, don't open `WinGHCi` first and then use that to open the files, as the process for `WinGHCi` will have the wrong working directory; instead it is best to open the test module with `WinGHCi` instead, by right-clicking on the `FTests.hs` file and selecting `WinGHCi` as the program to open it with.

[2]If upon opening the file, you receive a message about a type error, know that the type error is in your own code, not the code of the testing modules; if this happens, make sure that your function(s) have the type declarations they should and try running your code by itself on some test data that you create yourself.

In the `Win` constructor, which makes a single window, the three arguments stand for the name of the window (a String), the width of the window (an **`Int`**, in pixels), and the height of the window (another **`Int`**, also in pixels). Thus a window such as `(Win "Movie" 500 300)` is a window named "Movie" that has width of 500 pixels and a height of 300 pixels. (Assume that the width and height are always non-negative.) A `WindowPlan` of the form `(Horiz wps)`, where `wps` is a list of `WindowPlans` represents the window plans in `wps` arranged horizontally. Similarly, a `WindowPlan` of the form `(Vert wps)`, where `wps` is a list of `WindowPlans` represents the window plans in `wps` arranged vertically.

In Haskell, write a function

```
height :: WindowPlan -> Int
```

that takes a `WindowPlan`, `wp`, and returns the total height of the window plan (in pixels). The height is defined by cases as follows. The height of a `WindowPlan` of form `(Win` nm w h`)` is h. The height of a `WindowPlan` of the form `(Horiz [`wp_1`, ..., `wp_m`])` is 0 if the list is empty, and otherwise is the maximum of the heights of wp_1 through wp_m (inclusive). The height of a `WindowPlan` of the form `(Vert [`wp_1`, ..., `wp_m`])` is the sum of the heights of wp_1 through wp_m (inclusive), which is 0 if the list is empty. (You may assume that this sum is never greater than the largest **`Int`**.)

The file `HeightTests.hs` contains tests that show how the function should work, see Figure 1.

```
module HeightTests where
import Testing
import WindowPlan
import Height

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

tests :: [TestCase Int]
tests =
 [(eqTest (height (Win "olympics" 50 33)) "==" 33)
 ,(eqTest (height (Horiz [])) "==" 0)
 ,(eqTest (height (Vert [])) "==" 0)
 ,(eqTest (height (Horiz [(Win "olympics" 80 33), (Win "News" 20 10)])) "==" 33)
 ,(eqTest (height (Vert [(Win "olympics" 80 33), (Win "News" 20 10)])) "==" 43)
 ,(eqTest (height (Vert [(Win "Star Wars" 40 100), (Win "olympics" 80 33),
                         (Win "News" 20 10)]))
                  "==" 143)
 ,(eqTest (height (Horiz [(Vert [(Win "Tempest" 200 100), (Win "Othello" 200 77)
                                ,(Win "Hamlet" 1000 600)])
                         ,(Horiz [(Win "baseball" 50 40), (Win "track" 100 60)
                                 ,(Win "baking" 70 30)])
                         ,(Vert [(Win "Dancing with the Stars" 40 100)
                                ,(Win "olympics" 80 33), (Win "News" 20 10)])])) "==" 777) ]
```

Figure 1: Tests for problem 1.

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 ⟨WindowPlan⟩ 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).**

2. (20 points) [UseModels]

   This is another problem about the type `WindowPlan`. Write a function

   ```
   split :: String -> WindowPlan -> WindowPlan
   ```

   that takes a string, `name`, and a WindowPlan, `wp`, and returns a WindowPlan that is just like `wp`, except that for each window in `wp` whose name is (== to) `name` is changed to a `Horiz` window plan with both windows having the same name and half the width of the previous window plan. (Hint, use Haskell's `div` operator to do the division.) Figure 2 shows examples.

```
module SplitTests where
import WindowPlan; import Split; import Testing
main = dotests "SplitTests Revision : 1.2" tests
tests :: [TestCase WindowPlan]
tests =
 [(eqTest (split "olympics" (Win "olympics" 50 33))
   "==" (Horiz [(Win "olympics" 25 33), (Win "olympics" 25 33)]))
 ,(eqTest (split "masterpiece" (Horiz [])) "==" (Horiz []))
 ,(eqTest (split "nova" (Vert [])) "==" (Vert []))
 ,(eqTest (split "olympics" (Horiz [(Win "olympics" 79 33), (Win "local news" 21 10)]))
   "==" (Horiz [(Horiz (let w = (Win "olympics" 39 33) in [w, w])), (Win "local news" 21 10)]))
 ,(eqTest (split "local news" (Vert [(Win "olympics" 79 33)
                                    ,(Win "local news" 21 10)]))
   "==" (Vert [(Win "olympics" 79 33)
              ,(Horiz (let w = (Win "local news" 10 10) in [w, w]))]))
 ,(eqTest (split "Sienfeld"
           (Vert [(Win "Star Trek" 40 100), (Win "Sienfeld" 80 33)
                 ,(Win "Sienfeld" 30 10)]))
   "==" (Vert [(Win "Star Trek" 40 100)
                  ,(Horiz (let w = (Win "Sienfeld" 40 33) in [w,w]))
                  ,(Horiz (let w = (Win "Sienfeld" 15 10) in [w, w]))]))
 ,(eqTest (split "local news"
           (Horiz
            [(Vert [(Win "Tempest" 200 100), (Win "Othello" 200 77), (Win "Hamlet" 1000 600)])
            ,(Horiz [(Win "baseball" 50 40)
                         ,(Vert [(Win "local news" 100 60), (Win "ski jump" 70 30)])])
            ,(Vert [(Win "Star Trek" 40 100), (Horiz [(Win "olympics" 80 33)
                                                     ,(Win "local news" 20 10)])]) ]))
   "==" (Horiz
        [(Vert [(Win "Tempest" 200 100), (Win "Othello" 200 77), (Win "Hamlet" 1000 600)])
        ,(Horiz [(Win "baseball" 50 40)
                     ,(Vert [(Horiz (let w = (Win "local news" 50 60) in [w,w]))
                            ,(Win "ski jump" 70 30)])])
        ,(Vert [(Win "Star Trek" 40 100), (Horiz [(Win "olympics" 80 33)
                                                 ,(Horiz (let w = (Win "local news" 10 10) in [w,w]))])])
]))  ]
```

Figure 2: Tests for problem 2.

As always, after writing your code, run our tests, and turn in your solution and the output of our tests. **Be sure to follow the grammar, as we will take off points for not following the grammar (and you will have a harder time solving the problem if you don't follow the grammar).**

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

```
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 `simplify :: Statement -> Statement` that takes a Statement, `stmt`, and returns a Statement just like `stmt`, except that the following simplifications are made:

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

There are test cases contained in `SimplifyTests.hs`, which is shown in Figure 3.

```
module SimplifyTests where
import StatementsExpressions; import Simplify; import Testing
main = dotests "SimplifyTests $Revision: 1.2$" tests
tests :: [TestCase Statement]
tests =
 [(eqTest (simplify (IfStmt (VarExp "true") (ExpStmt (NumExp 7)))) "==" (ExpStmt (NumExp 7)))
 ,(eqTest (simplify (ExpStmt (BeginExp [] (NumExp 6)))) "==" (ExpStmt (NumExp 6)))
 ,(eqTest (simplify (ExpStmt (NumExp 7))) "==" (ExpStmt (NumExp 7)))
 ,(eqTest (simplify (ExpStmt (VarExp "q"))) "==" (ExpStmt (VarExp "q")))
 ,(eqTest (simplify (ExpStmt (VarExp "true"))) "==" (ExpStmt (VarExp "true")))
 ,(eqTest (simplify (ExpStmt (BeginExp [] (EqualsExp (VarExp "x") (VarExp "x")))))
   "==" (ExpStmt (EqualsExp (VarExp "x") (VarExp "x"))))
 ,(eqTest (simplify (AssignStmt "y" (EqualsExp (VarExp "jz") (VarExp "jz"))))
   "==" (AssignStmt "y" (EqualsExp (VarExp "jz") (VarExp "jz"))))
 ,(eqTest (simplify (IfStmt (VarExp "true") (AssignStmt "d" (VarExp "true"))))
   "==" (AssignStmt "d" (VarExp "true")))
 ,(eqTest (simplify
           (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 `Simplify`.

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. Be sure to follow the grammar!

4. (25 points) [UseModels] Consider the data type of quantified Boolean expressions defined as follows, in the file `QBExp.hs`. This module also defines `(Set a)` to mean the type `[a]` without duplicates.

```
-- $Id: QBExp.hs,v 1.3 2019/10/08 13:21:02 leavens Exp leavens $
module QBExp where
data QBExp = Varref String | QBExp `And` QBExp
           | Not QBExp | Forall String QBExp   deriving (Eq, Show)
type Set a = [a] -- without any duplicates
```

Your task is to write a function

```
freeQBExp :: QBExp -> (Set String)
```

that takes a `QBExp`, `qbe`, and returns a set 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 `(Forall s e)`, which declares (i.e., binds) `s`.

In the examples given in Figure 4, 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 for the main function in this problem (as it will horribly complicate your code)! Instead, use separate helping functions to prevent duplicates. And be sure to follow the grammar!

```
module FreeQBExpTests where
import QBExp; import FreeQBExp; import Testing
main = dotests "FreeQBExpTests $Revision: 1.3$" 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") `And` (Not (Varref "y")))) "==" ["x","y"]
        ,setEq (freeQBExp ((Not (Varref "y")) `And` (Varref "x"))) "==" ["y","x"]
        ,setEq (freeQBExp (((Varref "y") `And` (Varref "x"))
                           `And` ((Varref "x") `And` (Varref "y")))) "==" ["y","x"]
        ,setEq (freeQBExp (Forall "y" (Not (Varref "y")))) "==" []
        ,setEq (freeQBExp (Forall "y" ((Not (Varref "y")) `And` (Varref "z")))) "==" ["z"]
        ,setEq (freeQBExp (Forall "z" (Forall "y" ((Not (Varref "y")) `And` (Varref "z"))))) "==" []
        ,setEq (freeQBExp (Not
                          ((Varref "z")
                           `And` (Forall "z" (Forall "y" ((Varref "y") `And` (Varref "z")))))))
                    "==" ["z"]
        ,setEq (freeQBExp (((Varref "z") `And` (Varref "q"))
                           `And` (Not (Forall "z" (Forall "y" ((Varref "y") `And` (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 (rated by number of bird species seen in the year). Also lower numbers are better; everyone wants to be “first rated.” In such rated 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 (say) 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 a general rating function

   ```
   rate :: (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 rating of the element in the pair. There are test cases contained in the file `RateTests.hs`, which is shown in Figure 5 on the following page. To run our tests, use the `RateTests.hs` file. To make that work, you have to put your code in a module `Rate`.

   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.

```
module RateTests where
import Rate; import Testing
main :: IO ()
main = dotests2 "$Revision: 1.2$" testsString testsBirders
testsString :: [TestCase [(Int, String)]]
testsString = -- alphabetic ordering
    [(eqTest (rate []) "==" [])
    ,(eqTest (rate ["one"]) "==" [(1,"one")])
    ,(eqTest (rate ["one","one"]) "==" [(1,"one"),(1,"one")])
    ,(eqTest (rate ["two","one","one"]) "==" [(1,"one"),(1,"one"),(3,"two")])
    ,(eqTest (rate ["abel", "charlie", "baker", "abel", "charlie", "delta", "echo"])
      "==" [(1,"abel"), (1,"abel"), (3,"baker"),
            (4,"charlie"), (4,"charlie"), (6,"delta"), (7,"echo")])
     ,(eqTest (rate ["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 (rate []) "==" [])
    ,(eqTest (rate [(Person "Tom" 532),(Person "Pat" 532)]) "=="
                [(1,(Person "Tom" 532)),(1, (Person "Pat" 532))])
    ,(eqTest (rate [(Person "Pat" 532),(Person "Tom" 532)]) "=="
                [(1,(Person "Pat" 532)),(1, (Person "Tom" 532))])
    ,(eqTest (rate [(Person "Pat" 532),(Person "Tom" 532),(Person "Neil" 703)])
      "==" [(1,(Person "Neil" 703)),(2,(Person "Pat" 532)),(2, (Person "Tom" 532))])
    ,(eqTest (rate 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. (10 points) [UseModels] [Concepts] In cryptography, one would like to apply functions defined over the type `Int` to data of type `Char` and vice versa. However, in Haskell, these two types are distinct. In Haskell, write two functions

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

   The first function, `toCharFun` takes a function of type `Int -> Int`, and returns a function that operates on `Char`s. The second function, `fromCharFun` takes a function of type `Char -> Char`, and returns a function that operates on `Int`s.

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

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

```
module ToFromCharFunTests where
import ToCharFun  -- your solution goes in module ToFromCharFun
import Testing; import Data.Char -- defines toUpper
main = do startTesting "ToFromCharFunTests Revision : 1.1"
          startTesting "toCharFun"
          errs <- run_test_list 0 toTests
          startTesting "fromCharFun"
          errs <- run_test_list errs fromTests
          doneTesting errs
toTests :: [TestCase Char]
toTests = [eqTest (toCharFun (+3) 'a') "==" 'd'
          ,eqTest (toCharFun (+1) 'b') "==" 'c'
          ,eqTest (toCharFun (+7) 'a') "==" 'h'
          ,eqTest (toCharFun (+13) 'c') "==" 'p'
          ,eqTest (toCharFun (\c -> 10*c `div` 12) 'h') "==" 'V'  ]
fromTests :: [TestCase Int]
fromTests = [eqTest (fromCharFun (\c -> c) 7) "==" 7
            ,eqTest (fromCharFun (\c -> if c == 'b' then 'x' else c) 98) "==" 120
            ,eqTest (fromCharFun toUpper 97) "==" 65
            ,eqTest (fromCharFun vf 97) "==" 101
            ,eqTest (fromCharFun vf 117) "==" 97  ]
    where vf 'a' = 'e'
          vf 'e' = 'i'
          vf 'i' = 'o'
          vf 'o' = 'u'
          vf 'u' = 'a'
          vf c = c
```

Figure 6: Tests for problem 6.

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 `filter` function, write the function

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

   that for some type `a` takes a predicate `pred` of type `a -> Bool`, and a list of lists of type `a`, `lls`, and returns a list of type `[[a]]` that consists of the elements of each element inside each list in `lls`, that satisfies `pred` (i.e., for which `pred` applied to that element returns **True**).

   There are test cases contained in the file `FilterInsideTests.hs`, which is shown in Figure 7.

```
module FilterInsideTests where
import Testing; import Data.Char (isLower, isUpper, isLetter)
import FilterInside  -- you have to put your solutions in module FilterInside
-- do main to run our tests
main = do startTesting "FilterInsideTests Revision : 1.1"
          startTesting "filterInside on [[Int]] lists"
          errs_ints <- run_test_list 0 int_tests
          startTesting "filterInside on [[Char]] lists"
          total_errs <- run_test_list errs_ints string_tests
          doneTesting total_errs
int_tests :: [TestCase [[Int]]]
int_tests =
    [eqTest (filterInside (==1) []) "==" []
    ,eqTest (filterInside (==1) [[]]) "==" [[]]
    ,eqTest (filterInside (>=2) [[3,4,5],[4,0,2,0],[],[8,7,6]])
      "==" [[3,4,5],[4,2],[],[8,7,6]]
    ,eqTest (filterInside odd [[1 .. 10], [2,4 .. 20], [7]])
      "==" [[1, 3 .. 9], [], [7]]
    ,eqTest (filterInside even [[0 .. 10], [0,2 .. 10], [7]])
      "==" [[0, 2 .. 10], [0, 2 .. 10], []]
    ,eqTest (filterInside (<= 7) [[0,7,17,27], [94,5]]) "==" [[0,7],[5]]  ]
string_tests :: [TestCase [[Char]]]
string_tests =
    [eqTest (filterInside isLower ["A string", "is a list!"])
             "==" ["string","isalist"]
    ,eqTest (filterInside isUpper ["UCF","CS","is","great"])
             "==" ["UCF","CS","",""]
    ,eqTest (filterInside isLetter ["Haskell is","Wonderful","gr8 OK?"])
             "==" ["Haskellis","Wonderful","grOK"]      ]
```

Figure 7: Tests for problem 7.

Note that your code must use `filter` in an essential way. For full credit, write a solution that does not use any pattern matching, and which does not have repeated or unnecessary code. Hint: you might also want to use other higher-order functions.

As always, turn in both your code file and the output of your testing.

Functions as Data and Abstract Data Types

8. (15 points) [UseModels] In Haskell the built-in generic type **Maybe** a is defined as follows.

   ```
   data Maybe a = Nothing | Just a
   ```

   In this problem we will work with (what in this problem we call) "filters," which are functions of type (**Maybe** a -> **Maybe** a). When passed **Nothing**, which is used to represent the lack of a value, a filter will return **Nothing**. When passed a value, such as (**Just** x), which is used to represent the value x, a filter can either pass along the value by returning (**Just** x) or it can reject (filter out) the value by returning **Nothing**. An example is the function between shown in Figure 8, such that between 0.0 10.0, which has type (**Maybe Double**) -> (**Maybe Double**), filters out all numbers (strictly) less than 0.0 or greater than 10.0.

   This problem is to write a function

   ```
   composeFilters :: [(Maybe a -> Maybe a)] -> (Maybe a -> Maybe a)
   ```

   that takes a list of filters, $[f_1, f_2, \ldots, f_n]$ (for $n \geq 0$) and returns the filter $f_1 \circ f_2 \circ \cdots \circ f_n$, which is their composition. Note that $f_1 \circ f_2 \circ \cdots \circ f_n$ groups to the right, so as usual for mathematical function composition: $((f_1 \circ f_2 \circ \cdots \circ f_n)\, x) = (f_1 (f_2 \cdots (f_n\, x) \cdots))$. Note that (composeFilters []) is the identity function.

   Don't use last or init in your solution, as that will make your solution $O(n^2)$. We will take points off for solutions that are not clear or that have duplicated code.

   Tests are in the file ComposeFiltersTests.hs, shown in Figure 8.

```
module ComposeFiltersTests where
import ComposeFilters; import Testing
main = dotests "ComposeFiltersTests Revision : 1.3" tests
-- between (below) is a filter for testing; Note: not for you to implement!
between :: Double -> Double -> (Maybe Double) -> (Maybe Double)
between _ _ Nothing = Nothing
between lb ub (Just x) = if lb <= x && x <= ub then (Just x) else Nothing
-- the following makes a filter, like the above, from an aribtrary predicate
toFilter :: (a -> Bool) -> (Maybe a -> Maybe a) -- This is also not for you to implement!
toFilter _ Nothing = Nothing
toFilter pred (Just x) = if pred x then (Just x) else Nothing
tests :: [TestCase Bool]
tests =
    [assertTrue ((composeFilters [] (Just 3.14159)) == (Just 3.14159))
    ,assertTrue ((composeFilters [between 0.0 50.1] Nothing) == Nothing)
    ,assertTrue ((composeFilters [between 0.0 50.1] (Just 3.14)) == (Just 3.14))
    ,assertTrue ((composeFilters [between 0.0 50.1, between 1.0 5.0] (Just 3.14)) == (Just 3.14))
    ,assertTrue ((composeFilters [between 7.0 10.0] (Just 3.14)) == Nothing)
    ,assertTrue ((composeFilters [between 7.0 10.0, between 0.0 50.1, between 1.0 5.0] (Just 3.14)) == Nothing)
    ,assertTrue ((composeFilters [between 0.0 50.1, between 7.0 10.0, between 1.0 5.0] (Just 3.14)) == Nothing)
    ,assertTrue ((composeFilters [toFilter (>3), toFilter (<=5)] (Just 4)) == (Just 4))
    ,assertTrue ((composeFilters [toFilter (>'a'), toFilter (<'z')] (Just 'u')) == (Just 'u'))
    ,assertTrue ((composeFilters [toFilter (>'a'), toFilter (<'z'),toFilter (<'Z')] (Just 'u')) == Nothing)
    ,assertTrue ((composeFilters [toFilter (>7.0), toFilter (<0.0)] (Just 2.78)) == Nothing)  ]
```

Figure 8: Tests for problem 8.

FVector Generic Type and Examples for Testing

The following generic type definition, which is in the module `FVector` (file `FVector.hs`), is used in the next few problems.

```
module FVector where
data FVector t = FV (Int -> t) -- rule that determines the ith element
                    Int        -- the size >= 0, indexes go from 0 to (size - 1)

-- Return the element at the index given by the second argument
at :: (FVector t) -> Int -> t
at fv@(FV rule siz) i = if (legalIndex fv i)
                        then (rule i)
                        else (error ("index " ++ (show i) ++ " out of bounds"))

-- Return the size of the FVector argument
size :: (FVector t) -> Int
size (FV _ siz) = siz

-- Is the second argument (the Int) a legal index into the given FVector?
legalIndex :: (FVector t) -> Int -> Bool
legalIndex (FV _ siz) i = 0 <= i && i < siz

-- Return a list of the legal indexes up to the given size
indexList :: Int -> [Int]
indexList siz = [0 .. (siz - 1)]

-- Return a list of the elements in the given FVector
elements :: (FVector t) -> [t]
elements (FV rule siz) = map rule (indexList siz)

-- Provide the ability to turn an FVector into a String (if the elements allow)
instance (Show t) => Show (FVector t) where
    show (FV rule siz) = (show siz) ++ "<[" ++ (showElements rule siz) ++ "]>"
        where showElements rule siz = concatMap (showElement rule) (indexList siz)
              showElement rule i = if i < siz -1
                                   then (show (rule i)) ++ ", "
                                   else (show (rule i))

-- Provide the ability to compare FVectors for equality (if the elements allow)
instance (Eq t) => Eq (FVector t) where
    fv1@(FV rule1 siz1) == fv2@(FV rule2 siz2) =
        (siz1 == siz2)
        && (and (map (\(e1,e2) -> e1 == e2)  -- i.e., (uncurry (==))
                     (zip (elements fv1) (elements fv2))))
```

Figure 9: The module defining the generic type `FVector`, for use in later problems. There are several helping functions and two instances of standard classes also exported by this module.

The definitions given in the module `FVectorExamples` (see Figure 10 on the next page) are examples of functions that produce FVectors of the given size. These examples are used in later tests.

For testing FVectors of Doubles we use the module `FVectorTesting` (see Figure 11).

```
module FVectorExamples (module FVector, module FVectorExamples) where
import FVector
-- Note: THESE ARE NOT FOR YOU TO IMPLEMENT; they are examples for later testing
empty :: FVector Double
empty = (FV (\i -> error "should not be called") 0)
zeros :: Int -> (FVector Double)
zeros siz = (FV (\i -> 0.0) siz)
nats :: Int -> (FVector Double)
nats siz = (FV (\i -> fromIntegral i) siz)
negs :: Int -> (FVector Double)
negs siz = (FV (\i -> (- (fromIntegral i))) siz)
hundredDown :: Int -> (FVector Double)
hundredDown siz = (FV (\i -> fromIntegral (100 - i)) siz)
halves :: Int -> (FVector Double)
halves siz = (FV (\i -> (fromIntegral i)/(fromIntegral (2^i))) siz)
powersOf :: (Num a) => a -> Int -> (FVector a)
powersOf n siz = (FV (\i -> n^i) siz)
powersOf2 :: Int -> (FVector Double)
powersOf2 = powersOf 2
powersOf01:: Int -> (FVector Double)
powersOf01 = powersOf 0.1
```

Figure 10: FVector examples, for use in later tests.

```
module FVectorTesting where
import Testing; import FloatTesting
import FVector

-- For testing purposes when the data is a RealFloat (Float or Double)
fvWithin :: (Show a, RealFloat a, Tolerance a) =>
                 (FVector a) -> String -> (FVector a) -> (TestCase (FVector a))
fvWithin = gTest (\fv1 fv2 -> size fv1 == size fv2
                              && all (uncurry (~=~))
                                     (zip (elements fv1) (elements fv2)))

fvRel :: (Show a, RealFloat a, Tolerance a) =>
                 (FVector a) -> String -> (FVector a) -> (TestCase (FVector a))
fvRel = gTest (\fv1 fv2 -> size fv1 == size fv2
                           && all (uncurry (~~~))
                                  (zip (elements fv1) (elements fv2)))
```

Figure 11: FVector testing module for use in later tests.

9. (10 points) [UseModels] This is a problem about the `FVector` type in Figure 9 on page 12. In Haskell, write a function

   ```
   scaleFVector :: (Num t) => t -> (FVector t) -> (FVector t)
   ```

   that for any numeric type `t` takes a value of type `t`, `x`, and a `(FVector t)` value, `fv`, and returns an `FVector` such that the i^{th} element of the result is `x` times the i^{th} element of `fv`. Thus the result is the scalar multiplication of `x` and `fv`. Figure 12 shows examples, written using the `Testing` module from the homework and using the definitions given in the `FVectorExamples` module (see Figure 10 on the preceding page).

```
module ScaleFVectorTests where
import Testing; import FloatTesting; import FVectorTesting
import FVectorExamples
import ScaleFVector -- your code is in the ScaleFVector module
main = dotests "ScaleFVectorTests Revision : 1.1" tests
tests :: [TestCase (FVector Double)]
tests = [fvWithin (scaleFVector 1.0 (halves 6)) "~=~" (halves 6)
        ,(fvWithin (scaleFVector 2.0 (powersOf2 6))
            "~=~" (FV (\i -> 2.0 * (2.0^i)) 6))
        ,(fvWithin (scaleFVector 3.0 (hundredDown 8))
            "~=~" (FV (\i -> 3.0 * (fromIntegral (100 - i))) 8))
        ,(fvWithin (scaleFVector 5.0 (nats 20))
            "~=~" (FV (\i -> 5.0 * (fromIntegral i)) 20))
        ,fvWithin (scaleFVector 3.14 empty) "~=~" empty  ]
```

Figure 12: Tests for problem 9.

---

10. (15 points) [UseModels] This is another problem about the `FVector` generic type in Figure 9 on page 12. In Haskell, write the function

    ```
    mapFVector :: (a -> b) -> (FVector a) -> (FVector b)
    ```

    which for any types `a` and `b` takes a function, `fun`, of type `(a -> b)` and a `(FVector a)` value, `fv`, and returns an `(FVector b)` value, whose value at index `i` is the result of applying `fun` to `(at fv i)`. Figure 13 shows examples, written using the definitions in `FVectorExamples` (see Figure 10 on page 13).

```
module MapFVectorTests where
import Testing; import FloatTesting; import FVectorTesting
import FVectorExamples
import MapFVector -- your solution goes in this module
main = dotests "MapFVectorTests Revision : 1.1" tests
tests :: [TestCase (FVector Double)]
tests = [fvWithin (mapFVector id (powersOf01 5)) "~=~" (powersOf01 5)
        ,fvWithin (mapFVector (+2.0) (hundredDown 8))
             "~=~" (FV (\i -> (fromIntegral (102 - i))) 8)
        ,fvWithin (mapFVector (\d -> 3.0*d+1.0) (nats 7))
             "~=~" (FV (\i -> 3.0*(fromIntegral i)+1.0) 7)
        ,fvWithin (mapFVector (\d -> d*d+2.0) (nats 7))
             "~=~" (FV (\i -> (fromIntegral (i*i))+2.0) 7)
        ,fvWithin (mapFVector (*2.0) (halves 8))
             "~=~" (FV (\i -> (fromIntegral i)/(fromIntegral (2^i))*2.0) 8)
        ,fvWithin (mapFVector (\d -> error "shouldn't be") empty) "~=~" empty  ]
```

Figure 13: Tests for problem 10.

We will take points off for code that uses intermediate lists (or `map`); instead create an appropriate rule for a new `FVector`.

11. (15 points) This is also a problem about the generic type `FVector`, see Figure 9 on page 12. In Haskell, write the function

    ```
    addFVector :: (Num t) => (FVector t) -> (FVector t) -> (FVector t)
    ```

    that for any numeric type `t`, takes two `(FVector t)` values, `fv1` and `fv2` and if they have the same size, then it returns a new vector where the i^{th} element of the result is the sum of the i^{th} element of `fv1` and the i^{th} element of `fv2`. If `fv1` and `fv2` have different sizes, then an error should be signaled (by calling Haskell's **`error`** function with a string argument explaining the error). Figure 14 shows examples (of non-error cases) written using the definitions in `FVectorExamples` (see Figure 10 on page 13).

```
module AddFVectorTests where
import Testing; import FloatTesting; import FVectorTesting
import FVector; import FVectorExamples
import AddFVector -- your code goes in this module
main = dotests "AddFVectorTests $Revision: 1.2$" tests
tests :: [TestCase (FVector Double)]
tests = [fvWithin (addFVector empty empty) "~=~" empty
        ,fvWithin (addFVector (nats 11) (nats 11))
             "~=~" (FV (\i -> (fromIntegral (i+i))) 11)
        ,fvWithin (addFVector (nats 11) (negs 11)) "~=~" (zeros 11)
        ,fvWithin (addFVector (halves 9) (FV (\i -> 1.0) 9))
             "~=~" (FV (\i -> (fromIntegral i)/(fromIntegral (2^i))+1.0) 9)
        ,fvWithin (addFVector (powersOf2 11) (powersOf2 11))
             "~=~" (FV (\i -> (fromIntegral (2^i))*2.0) 11)
        ,fvWithin (addFVector (powersOf01 5) (zeros 5)) "~=~" (powersOf01 5)   ]
```

Figure 14: Tests for problem 11.

We will take points off for code that uses intermediate lists; instead create an appropriate rule for a new `FVector`.

12. (15 points) This is also a problem about the generic type `FVector`, see Figure 9 on page 12. In Haskell, write the function

    ```
    sumFVector :: (Num t) => (FVector t) -> t
    ```

    that for any numeric type `t`, takes an `(FVector t)` value, `fv`, and returns the sum of that vector's elements. If the vector is empty (has no elements), then zero is returned. Figure 15 shows examples, written using the definitions in `FVectorExamples` (see Figure 10 on page 13).

```
module SumFVectorTests where
import Testing; import FloatTesting
import FVector; import FVectorExamples
import SumFVector -- your code goes in this module
main = dotests "SumFVectorTests Revision : 1.1" tests
tests :: [TestCase Double]
tests = [withinTest (sumFVector empty) "~=~" 0.0
        ,withinTest (sumFVector (zeros 1000)) "~=~" 0.0
        ,withinTest (sumFVector (nats 11)) "~=~" 55.0
        ,withinTest (sumFVector (negs 11)) "~=~" (-55.0)
        ,withinTest (sumFVector (hundredDown 201)) "~=~" 0.0
        ,withinTest (sumFVector (halves 9)) "~=~" 1.9609375
        ,withinTest (sumFVector (powersOf2 11)) "~=~" 2047.0
        ,withinTest (sumFVector (powersOf01 5)) "~=~" 1.1111    ]
```

Figure 15: Tests for problem 12.

We will take points off for code that uses intermediate lists; instead create an appropriate rule for a new `FVector`.

13. (30 points) [Concepts] [UseModels] A set can be described by a “characteristic predicate” (i.e., a 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 predicate for a set containing the strings "coke" and "pepsi", but nothing else. Allowing the user to construct a set from a characteristic predicate gives one the power to construct sets that may “contain” an infinite number of elements (such as the set of all even 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

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

       takes a characteristic predicate, p and returns a set such that each value x (of type a) is in the set just when px is True.

    2. The function

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

       takes two sets, with characteristic predicates p and q, and returns a set such that each value x (of type a) is in the set just when either (px) or (qx) is true.

    3. The function

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

       takes two sets, with characteristic predicates p and q, and returns a set such that each value x (of type a) is in the set just when both (px) and (qx) 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 16 on the following page.

    Note (hint, hint) that the following equations must hold, for all p, q, and x of appropriate types.

```
inSet x (unionSet (fromPred p) (fromPred q)) == (p x) || (q x)
inSet x (intersectSet (fromPred p) (fromPred q)) == (p x) && (q x)
inSet x (fromPred p) == p x
inSet x (complementSet (fromPred p)) == not (p x)
```

```
module InfSetTests where
import InfSet
import Testing

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

tests :: [TestCase Bool]
tests =
    [assertTrue (inSet 2 (fromPred even))
    ,assertFalse (inSet 3 (fromPred even))
    ,assertTrue (inSet 3 (fromPred odd))
    ,assertTrue (inSet "coke" (fromPred (\ x -> x == "coke")))
    ,assertFalse (inSet "pepsi" (fromPred (\ x -> x == "coke")))
    ,assertFalse (inSet "coke" (complementSet (fromPred (\x -> x == "coke"))))
    ,assertTrue (inSet "oil" (complementSet (fromPred (\x -> x == "coke"))))
    ,assertTrue (inSet "pepsi" (unionSet (fromPred (\ x -> x == "coke"))
                                (fromPred (\ x -> x == "pepsi"))))
    ,assertTrue (inSet "coke" (unionSet (fromPred (\x -> x == "coke"))
                               (fromPred (\x -> x == "pepsi"))))
    ,assertFalse (inSet "sprite" (unionSet (fromPred (\x -> x == "coke"))
                                  (fromPred (\x -> x == "pepsi"))))
    ,assertFalse (inSet "coke" (intersectSet (fromPred (\x -> x == "coke"))
                                (fromPred (\x -> x == "pepsi"))))
    ,assertFalse (inSet "pepsi" (intersectSet (fromPred (\x -> x == "coke"))
                                (fromPred (\x -> x == "pepsi"))))
    ,assertTrue (inSet "dr. p" (intersectSet (fromPred (\x -> "coke" <= x))
                                 (fromPred (\x -> x <= "pepsi"))))
    ,assertTrue (inSet "pepsi" (intersectSet (fromPred (\x -> "coke" <= x))
                                (fromPred (\x -> x <= "pepsi"))))
    ,assertFalse (inSet "beer" (intersectSet (fromPred (\x -> "coke" <= x))
                                (fromPred (\x -> x <= "pepsi"))))
    ,assertFalse (inSet "wine" (intersectSet (fromPred (\x -> "coke" <= x))
                                (fromPred (\x -> x <= "pepsi"))))
    ,assertTrue (inSet "wine" (unionSet (fromPred (\x -> "coke" <= x))
                                (fromPred (\x -> x <= "pepsi"))))      ]
```

Figure 16: 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

14. (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 17.

```
module ConcatMapTests where
import Prelude hiding (concatMap)
import ConcatMap
import Testing

main :: IO()
main = dotests "ConcatMapTests $Revision: 1.5$" 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 17: Tests for problem 14.

15. (30 points) [UseModels] [Concepts] In this problem you will write a function

```
foldWindowPlan :: ((String,Int,Int) -> r) -> ([r] -> r) -> ([r] -> r)
                  -> WindowPlan -> r
```

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

- `wf`, operates on a tuple of the information from a `Win` variant and returns a value of type `r`,
- `hf`, takes a list of the results of mapping `(foldWindowPlan wf hf vf)` over the list in a `Horiz` variant, and returns a value of type `r`,
- `vf`, takes a list of the results of mapping `(foldWindowPlan wf hf vf)` over the list in a `Vert` variant, and returns a value of type `r`.

There are test cases contained in `FoldWindowPlanTests.hs`, which is shown in Figure 18 on the following page and Figure 19 on page 23.

16. [UseModels] [Concepts] Use your `foldWindowPlan` function to implement
    (a) (15 points) `height` from problem 1, and
    (b) (15 points) `split` from problem 2.

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

    Write these solutions into modules `Height` and `Split`, respectively, that both import the module `FoldWindowPlan` from the previous problem, and then run the tests appropriate to each of these window plan problems. Hand in your code for these new implementations of `height` and `split` as well as the output of both tests on this "assignment" on webcourses.

Points

This homework's total points: 295.

```
-- $Id: FoldWindowPlanTests.hs,v 1.1 2019/10/09 15:48:55 leavens Exp leavens $
module FoldWindowPlanTests where
import WindowPlan
import FoldWindowPlan
import Testing

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

-- uses of foldWindowPlan for testing purposes, not for you to implement
watching' = foldWindowPlan (\(wn,_,_) -> [wn]) concat concat
changeChannel new old =
    let changeName new old nm = if nm == old then new else nm
    in foldWindowPlan
           (\(nm,w,h) -> (Win (changeName new old nm) w h))
           Horiz
           Vert
doubleSize = foldWindowPlan
              (\(wn,w,h) -> (Win wn (2*w) (2*h)))
              Horiz
              Vert
addToSize n = foldWindowPlan
               (\(wn,w,h) -> (Win wn (n+w) (n+h)))
               Horiz
               Vert
multSize n = foldWindowPlan
               (\(wn,w,h) -> (Win wn (n*w) (n*h)))
               Horiz
               Vert
totalWidth = foldWindowPlan
               (\(_,w,_) -> w)
               sum
               sum

-- a WindowPlan for testing
hplan =
    (Horiz
     [(Vert [(Win "Tempest" 200 100), (Win "Othello" 200 77), (Win "Hamlet" 1000 600)])
     ,(Horiz [(Win "baseball" 50 40), (Win "track" 100 60), (Win "golf" 70 30)])
     ,(Vert [(Win "Star Trek" 40 100), (Win "olympics" 80 33), (Win "news" 20 10)])])

tests :: [TestCase Bool]
tests =
    [assertTrue ((totalWidth hplan) == 1760)
    ,assertTrue ((doubleSize hplan) == (multSize 2 hplan))
    ,assertTrue ((watching' hplan)
                 == ["Tempest","Othello","Hamlet","baseball","track","golf",
```

Figure 18: Tests for problem 15, part 1.

```
,assertTrue
 ((changeChannel
   "pbs" "news"
   (Vert [(Win "news" 10 5), (Win "golf" 50 25), (Win "news" 30 70)]))
  ==
  (Vert [(Win "pbs" 10 5), (Win "golf" 50 25), (Win "pbs" 30 70)]))
,assertTrue
 ((addToSize 100 hplan)
  ==
  (Horiz
   [(Vert [(Win "Tempest" 300 200), (Win "Othello" 300 177), (Win "Hamlet" 1100 700)])
   ,(Horiz [(Win "baseball" 150 140), (Win "track" 200 160), (Win "golf" 170 130)])
   ,(Vert [(Win "Star Trek" 140 200), (Win "olympics" 180 133), (Win "news" 120 110)])]) )
,assertTrue
 ((doubleSize hplan)
  ==
  (Horiz
   [(Vert [(Win "Tempest" 400 200), (Win "Othello" 400 154), (Win "Hamlet" 2000 1200)])
   ,(Horiz [(Win "baseball" 100 80), (Win "track" 200 120), (Win "golf" 140 60)])
   ,(Vert [(Win "Star Trek" 80 200), (Win "olympics" 160 66), (Win "news" 40 20)])]) )    ]
```

Figure 19: Tests for problem 15, part 2.

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.