Homework 3: Functional Programming in Haskell

See Webcourses2 and the syllabus for due dates.

Purpose

In this homework you will learn basic techniques of recursive programming over various types of (recursively-structured) 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 the textbook.
We will take some points off for: code with the wrong type or wrong name, duplicated code, code with extra unnecessary cases, or code that is excessively hard to follow. You should always assume that the inputs given to each function will be well-typed, thus your code should not have extra cases for inputs that are not of the proper type. Make sure your code has the specified type by including the given type declaration with your code. Avoid duplicating code by using helping functions, library functions (when not prohibited in the problems), or by using syntactic sugars and local definitions (using **`let`** and **`where`**). It is a good idea to check your code for these problems before submitting.
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 you turn in problems you worked on in a Webcourses2 group.)
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 Webcourses2) your code and output of testing with our test cases. Please upload code as a plain (text) file with the name given in the problem or testing file and with the suffix `.hs` or `.lhs` (that is, do not give us a Word document or a PDF file for the code). Also paste the output from our tests into the Comment box for that "assignment". For 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.

What to Read

For learning Haskell, you may want to read some of the Haskell tutorials. Use the Haskell 2010 Report as a guide to the details of Haskell. Also read "Following the Grammar with Haskell" [Lea13] and follow its suggestions for organizing your code. You may also want to read a tutorial on the concepts of functional programming languages, such as Hudak's computing survey article mentioned in the syllabus. See also the course code examples page (and the course resources page).

Problems

Recursion over Flat Lists

These problems are intended to give you an idea of how to write recursions by following the grammar for flat lists [Lea13].

1. [UseModels] This problem will have you write a solution in 2 ways. The problem is to write a function that takes a list of Integers and returns a list that is just like the argument but in which every element is 1 larger than the corresponding element in the argument list.

   (a) (5 points) Write the function

   ```
   add1_list_comp :: [Integer] -> [Integer]
   ```

   that solves the above problem by using a list comprehension.

   (b) (5 points) Write the function

   ```
   add1_list_rec :: [Integer] -> [Integer]
   ```

   that solves the above problem by writing out the recursion yourself; that is, without using a list comprehension and without using **`map`** or any other higher-order library function.

   There are test cases contained in `Add1ListTests.hs`, which is shown in Figure 1 on the following page.

   To run our tests, use the `Add1ListTests.hs` file. To make that work, you have to put your code in a module `Add1List`, which will need to be in in a file named `Add1List.hs` (or `Add1List.lhs`), in the same directory as `Add1ListTests.hs`. Your file `Add1List.hs` should thus start as follows.

   ```
   module Add1List where
   add1_list_rec :: [Integer] -> [Integer]
   add1_list_comp :: [Integer] -> [Integer]
   ```

   Then run our tests by running the `main` function in `Add1ListTests.hs`.

   As specified on the first page of this homework, turn in both your code file and the output of your testing. (The code file should be uploaded to Webcourses2, and the test output should be pasted in to the Comments box for that assignment.)

```
-- $Id: Add1ListTests.hs,v 1.3 2013/01/27 19:01:02 leavens Exp leavens $
module Add1ListTests where
import Testing
import Add1List  -- you have to put your solutions in module Add1List

version = "Add1ListTests $Revision: 1.3 $"
recursive_tests = (tests add1_list_rec)
comprehension_tests = (tests add1_list_comp)

-- do main to run our tests
main :: IO()
main = do startTesting version
          errs_comp <- run_test_list 0 comprehension_tests
          total_errs <- run_test_list errs_comp recursive_tests
          doneTesting total_errs

-- do test_comprehension to test just add1_list_comp
test_comprehension :: IO()
test_comprehension = dotests version comprehension_tests

-- do test_recursive to test just add1_list_rec
test_recursive :: IO()
test_recursive = dotests version recursive_tests

tests :: ([Integer] -> [Integer]) -> [TestCase [Integer]]
tests f =
    [(eqTest (f []) "==" [])
    ,(eqTest (f (1:[])) "==" (2:[]))
    ,(eqTest (f (3:1:[])) "==" (4:2:[]))
    ,(eqTest (f [1,5,7,1,7]) "==" [2,6,8,2,8])
    ,(eqTest (f [5 .. 20]) "==" [6 .. 21])
    ,(eqTest (f [9,5,-3,4,2,99999999,10])
          "==" [10,6,-2,5,3,100000000,11])
    ]
```

Figure 1: Tests for problem 1. In these tests `f` is one of your solutions.

2. [UseModels] This problem will have you write a solution in 2 ways. The problem is to write a function that takes an element, `toDelete`, of some equality type `a`, and a list, `as`, of type `[a]`, and returns a list that is just like `as`, but which does not contain the element `toDelete`.

   (a) (5 points) Write the function

   ```
   deleteAll_list_comp :: (Eq a) => a -> [a] -> [a]
   ```

   that solves the above problem by using a list comprehension.

   (b) (5 points) Write the polymorphic function

   ```
   deleteAll_list_rec :: (Eq a) => a -> [a] -> [a]
   ```

   that solves the above problem by writing out the recursion yourself; that is, without using a list comprehension and without using any library functions.

   There are test cases contained in `DeleteAllTests.hs`, which is shown in Figure 2 on the next page.

   To run our tests, use the `DeleteAllTests.hs` file, as in the previous problem. To make that work, as before, you have to put your code in a module `DeleteAll`.

   As specified on the first page of this homework, turn in both your code file and the output of your testing. (The code file should be uploaded to Webcourses2, and the test output should be pasted in to the Comments box.)

```haskell
-- $Id: DeleteAllTests.hs,v 1.2 2013/01/27 02:49:08 leavens Exp leavens $
module DeleteAllTests where
import Testing
import DeleteAll  -- you have to put your solutions in this module

version = "DeleteAllTests $Revision: 1.2 $"
recursive_tests = (tests deleteAll_list_rec)
comprehension_tests = (tests deleteAll_list_comp)

-- do main to run our tests
main :: IO()
main = do startTesting version
          errs_comp <- run_test_list 0 comprehension_tests
          total_errs <- run_test_list errs_comp recursive_tests
          doneTesting total_errs

-- do test_comprehension to test just deleteAll_list_comp
test_comprehension :: IO()
test_comprehension = dotests version comprehension_tests

-- do test_recursive to test just deleteAll_list_rec
test_recursive :: IO()
test_recursive = dotests version recursive_tests

tests :: (Int -> ([Int] -> [Int])) -> [TestCase [Int]]
tests f =
    [(eqTest (f 3 []) "==" [])
    ,(eqTest (f 3 (1:[])) "==" (1:[]))
    ,(eqTest (f 1 (1:[])) "==" [])
    ,(eqTest (f 3 (3:1:[])) "==" (1:[]))
    ,(eqTest (f 1 (3:1:[])) "==" (3:[]))
    ,(eqTest (f 7 (3:1:[])) "==" (3:1:[]))
    ,(eqTest (f 7 [1,5,7,1,7]) "==" [1,5,1])
    ,(eqTest (f 1 [1,5,7,1,7]) "==" [5,7,7])
    ,(eqTest (f 20 ([1 .. 50] ++ (reverse [1 .. 50])))
      "==" (let out = [1 .. 19] ++ [21 .. 50] in (out ++ (reverse out))))
    ]
```

Figure 2: Tests for problem 2. In these tests `f` is one of your solutions.

3. (5 points) [UseModels] Write the function

   ```
   deleteFirst :: (Eq a) => a -> [a] -> [a]
   ```

   that takes an element, `toDelete`, of some equality type `a`, and a list, `as`, of type `[a]`, and returns a list that is just like `as`, but which does not contain the first occurrence (in `as`) of the element `toDelete`.

   Your solution must not use any Haskell library functions.

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

```
-- $Id: DeleteFirstTests.hs,v 1.1 2013/01/27 02:49:08 leavens Exp leavens $
module DeleteFirstTests where
import Testing
import DeleteFirst

-- do main to run our tests
main :: IO()
main = dotests "DeleteFirstTests $Revision: 1.1 $" tests

tests :: [TestCase [Int]]
tests =
    [(eqTest (deleteFirst 3 []) "==" [])
    ,(eqTest (deleteFirst 3 (1:[])) "==" (1:[]))
    ,(eqTest (deleteFirst 1 (1:[])) "==" [])
    ,(eqTest (deleteFirst 3 (3:1:[])) "==" (1:[]))
    ,(eqTest (deleteFirst 3 (3:1:3:[])) "==" (1:3:[]))
    ,(eqTest (deleteFirst 3 (3:3:3:[])) "==" (3:3:[]))
    ,(eqTest (deleteFirst 1 (3:1:[])) "==" (3:[]))
    ,(eqTest (deleteFirst 1 (1:3:1:[])) "==" (3:1:[]))
    ,(eqTest (deleteFirst 7 (3:1:[])) "==" (3:1:[]))
    ,(eqTest (deleteFirst 7 [1,5,7,1,7]) "==" [1,5,1,7])
    ,(eqTest (deleteFirst 1 [1,5,7,1,7]) "==" [5,7,1,7])
    ,(eqTest (deleteFirst 8 [8,8,8,8,8,8]) "==" [8,8,8,8,8])
    ,(eqTest (deleteFirst 8 [8,2,8,8,8,8,8,8]) "==" [2,8,8,8,8,8,8])
    ,(eqTest (deleteFirst 20 ([1 .. 50] ++ (reverse [1 .. 50])))
      "==" ([1 .. 19] ++ [21 .. 50] ++ (reverse [1 .. 50])))
    ]
```

Figure 3: Tests for problem 3.

Hint, look at the test cases in Figure 3 carefully, and think about what the differences are from problem 2.

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.

4. (5 points) [Concepts] Is it possible to use a list comprehension to solve problem 3 in the same direct way you could use a list comprehension to solve the previous problems? Briefly explain.

5. (10 points) [UseModels] Write the polymorphic function

   ```
   insertBefore :: (Eq a) => a -> a -> [a] -> [a]
   ```

   that takes an element, `place`, of some equality type `a`, a second element, `what`, also of type `a`, and a list of elements `as`, of type `[a]`, and returns a list that is just like `as`, but which has `what` inserted just before the first occurrence of the element `place`. If `place` does not occur in `as`, then `as` is returned unchanged.

   Your solution must not use any Haskell library functions.

   There are test cases contained in `InsertBeforeTests.hs`, which is shown in Figure 4.

```
-- $Id: InsertBeforeTests.hs,v 1.1 2013/01/27 02:49:08 leavens Exp $
module InsertBeforeTests where
import Testing
import InsertBefore

-- do main to run our tests
main :: IO()
main = dotests "InsertBeforeTests $Revision: 1.1 $" tests

tests :: [TestCase [Int]]
tests =
    [(eqTest (insertBefore 3 5 []) "==" [])
    ,(eqTest (insertBefore 3 5 (3:[])) "==" (5:3:[]))
    ,(eqTest (insertBefore 1 7 [2,1,0,-1]) "==" [2,7,1,0,-1])
    ,(eqTest (insertBefore 1 7 [2 .. 15]) "==" [2 .. 15])
    ,(eqTest (insertBefore 3 9 [3,1,3]) "==" [9,3,1,3])
    ,(eqTest (insertBefore 3 9 [5,3,1,3]) "==" [5,9,3,1,3])
    ,(eqTest (insertBefore 3 0 [3,3,3,0,2]) "==" [0,3,3,3,0,2])
    ,(eqTest (insertBefore 1 7 [6,3,1,0,1,3,7,1]) "==" [6,3,7,1,0,1,3,7,1])
    ,(eqTest (insertBefore 20 6 ([1 .. 50] ++ [1 .. 50]))
      "==" ([1 .. 19] ++ (6:[20 .. 50]) ++ [1 .. 50]))
    ]
```

Figure 4: Tests for problem 5.

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.

6. (10 points) [UseModels]

   Write a polymorphic function

   ```
   invert :: [(a,b)] -> [(b,a)]
   ```

   that takes a list of pairs and inverts (transposes) each pair in the list. If one considers that the argument is a binary relation, then the result is the mathematical inverse of that relation.

   See Figure 5 for examples from our module `InvertTests`.

```
-- $Id: InvertTests.hs,v 1.1 2013/01/25 20:50:45 leavens Exp leavens $
module InvertTests where
import Testing
import Invert

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

tests :: [TestCase [(Char,Int)]]
tests =
    [(eqTest (invert []) "==" [])
    ,(eqTest (invert [(2,'b')]) "==" [('b',2)])
    ,(eqTest (invert [(3,'c'),(1,'a'),(2,'b')])
      "==" [('c',3),('a',1),('b',2)])
    ,(eqTest (invert [(42,'c'),(43,'c'),(41,'c'),(3,'c'),(1,'a'),(2,'b')])
      "==" [('c',42),('c',43),('c',41),('c',3),('a',1),('b',2)])
    ,(eqTest (invert (zip [1 .. 26] ['a' .. 'z']))
      "==" (zip ['a' .. 'z'] [1 .. 26]))
    ,(eqTest (invert (zip [1 .. 26] ['a' .. 'z']))
      "==" (zip ['a' .. 'z'] [1 .. 26]))
    ,(eqTest (invert (zip [10 .. 36] ['A' .. 'Z']))
      "==" (zip ['A' .. 'Z'] [10 .. 36]))
    ]
```

Figure 5: Tests for problem 6. (10 points)

---

7. (10 points) [UseModels]

   Write a function

   ```
   hep :: [Word] -> [Word]
   type Word = String
   ```

   that takes a list of words (i.e., Strings not containing blanks), `txt`, and returns a list just like `txt` but with the following substitutions made each time they appear as consecutive words in `txt`:

   - `you` is replaced by `u`,
   - `are` is replaced by `r`,
   - `your` is replaced by `ur`,
   - the three words `by the way` are replaced by the word `btw`,
   - the three words `for your information` is replaced by the word `fyi`,
   - `boyfriend` is replaced by `bf`,
   - `girlfriend` is replaced by `gf`,
   - the three words `be right back` are replaced by the word `brb`,

- the three words `laughing out loud` are replaced by the word `lol`,
- the two words `see you` are replaced by the word `cya`, and
- `great` is replaced by `gr8`.

This list is complete (for this problem).

The examples in Figure 6 are written using the `Testing` module supplied with the homework. They r also found in our testing file `HepTests.hs` which u can get from webcourses (in the zip file attached to problem 1). Be sure to turn in both ur code and the output of our tests on webcourses.

```
-- $Id: HepTests.hs,v 1.1 2013/01/25 20:50:45 leavens Exp leavens $
module HepTests where
import Testing
import Hep

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

tests :: [TestCase [String]]
tests =
    [(eqTest (hep []) "==" [])
    ,(eqTest (hep ["you","you","you","you"]) "==" ["u","u","u","u"])
    ,(eqTest (hep ["you","know","I","will","see","you","soon"])
      "==" ["u","know","I","will","cya","soon"])
    ,(eqTest (hep ["by","the","way","you","must","see","my","girlfriend","she","is","great"])
      "==" ["btw","u","must","see","my","gf","she","is","gr8"])
    ,(eqTest (hep (["for","your","information","you","are","a","pig"]
                   ++ ["see","you","later","when","you","find","me","a","boyfriend"]))
      "==" ["fyi","u","r","a","pig","cya","later","when","u","find","me","a","bf"])
    ,(eqTest (hep ["by","the","way","I","will","be","right","back"])
      "==" ["btw","I","will","brb"])
    ]
```

Figure 6: Tests for problem 7.

BTW, we will take some number of points off if u have repeated code in ur solution. U can avoid repeated code by using a helping function or a case-expression. A case-expression would be used in a larger expression to form the result list, like: `case w of ...`.

Iteration

8. (10 points) [UseModels]

   In Haskell, write a polymorphic function

   ```
   listMin :: (Ord a) => [a] -> a
   ```

   that takes a non-empty, finite list, `lst`, whose elements can be compared (hence the requirement in the type that a is an **Ord** instance), and returns a minimum element from `lst`. That is, the result should be an element of `lst` that is no greater than any other element of `lst`.

   Your code must not use any library functions.

   In your code, you can assume that the argument list is non-empty and finite. There are test cases contained in `ListMinTests.hs`, which is shown in Figure 7.

   Hint: it may be useful to use tail recursion for this problem.

```
-- $Id: ListMinTests.hs,v 1.1 2013/01/25 20:50:45 leavens Exp leavens $
module ListMinTests where
import Testing
import ListMin

main = do startTesting "ListMinTests $Revision: 1.1 $"
          errs <- run_test_list 0 tests_ints
          total <- run_test_list errs tests_chars
          doneTesting total

tests_ints :: [TestCase Int]
tests_ints =
    [(eqTest (listMin (1:1:1:1:1:[])) "==" 1)
    ,(eqTest (listMin (26:[])) "==" 26)
    ,(eqTest (listMin (1:[])) "==" 1)
    ,(eqTest (listMin (1:26:[])) "==" 1)
    ,(eqTest (listMin (26:1:[])) "==" 1)
    ,(eqTest (listMin (1:2:3:4:1:3:5:26:[])) "==" 1)
    ,(eqTest (listMin (4:0:2:0:[])) "==" 0)
    ,(eqTest (listMin (99:86:12: -3:[])) "==" (-3))
    ,(eqTest (listMin (100000:8600000:12222: -999999:[])) "==" (-999999))
    ]

tests_chars :: [TestCase Char]
tests_chars =
    [
     (eqTest (listMin "upqieurqoeiruazvzkpsau") "==" 'a')
    ,(eqTest (listMin "see haskell.org for more about Haskell") "==" ' ')
    ]
```

Figure 7: Tests for `listMin`.

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.

9. (10 points) [UseModels]

   In Haskell, write a polymorphic function

   ```
   whatIndex :: (Eq a) => a -> [a] -> Integer
   ```

   that takes an element of some Eq type, `a`, `sought`, and a finite list, `lst`, and returns the 0-based index of the first occurence of `sought` in `lst`. However, if `sought` does not occur in `lst`, it returns -1.

   Your code must not use any library functions.

   In your code, you can assume that the argument list is finite. There are test cases contained in `WhatIndexTests.hs`, which is shown in Figure 8.

   Hint: it may be useful to use tail recursion for this problem.

```
-- $Id: WhatIndexTests.hs,v 1.1 2013/01/27 19:01:02 leavens Exp leavens $
module WhatIndexTests where
import WhatIndex
import Testing

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

tests :: [TestCase Integer]
tests =
    [(eqTest (whatIndex 3 []) "==" (-1))
    ,(eqTest (whatIndex 2 [1,2,3,2,1]) "==" 1)
    ,(eqTest (whatIndex 'a' ['a' .. 'z']) "==" 0)
    ,(eqTest (whatIndex 'b' ['a' .. 'z']) "==" 1)
    ,(eqTest (whatIndex 'c' ['a' .. 'z']) "==" 2)
    ,(eqTest (whatIndex 'q' ['a' .. 'z']) "==" 16)
    ,(eqTest (whatIndex (41,'c') [(42,'c'),(43,'c'),(41,'c'),(3,'c')])
      "==" 2)
    ,(eqTest (whatIndex True [False,False,False]) "==" (-1))
    ,(eqTest (whatIndex True [False,False,False,True]) "==" 3)
    ,(eqTest (whatIndex True [True,False,False,False]) "==" 0)
    ,(eqTest (whatIndex True [True,True,True]) "==" 0)
    ,(eqTest (whatIndex 1000 [1 .. 4000]) "==" 999)
    ]
```

Figure 8: Tests for `WhatIndex`.

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.

Recursion over Grammars

In the following problems, you can use whatever Haskell library functions you wish.

10. (10 points) [UseModels]

The problem uses the type `WindowLayout` [Lea13], shown below.

```
module WindowLayout where
data WindowLayout = Window {wname :: String, width :: Int, height :: Int}
                  | Horizontal [WindowLayout]
                  | Vertical [WindowLayout]
                  deriving (Show, Eq)
```

Working in a module in which Haskell's module `Data.Set` is imported as follows,

```
import qualified Data.Set as HSet
```

write a function

```
watching :: WindowLayout -> HSet.Set String
```

that takes a Window Layout, `wl`, and returns a Set of all window names appearing in the window layout.

Note that the type constructor is called `HSet.Set` because Haskell's `Data.Set` module is imported with a qualified naming scheme, so that all of its exported names must be prefixed with "`HSet.`". See the documentation for `Data.Set` on Haskell.org for more details about the `Data.Set` module and the functions it provides.

There are test cases contained in `WatchingTests.hs`, which is shown in Figure 9 on the next 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. Note, to use `Data.Set` and its functions, your module `Watching` must import `Data.Set`, as shown in the **`import`** directive above.

```
-- $Id: WatchingTests.hs,v 1.1 2013/01/27 19:01:02 leavens Exp $
module WatchingTests where
import WindowLayout
import qualified Data.Set as HSet
import Watching
import Testing

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

tests :: [TestCase (HSet.Set String)]
tests =
 [(eqTest (watching (Window {wname = "olympics", width = 50, height = 33}))
          "==" (HSet.singleton "olympics"))
 ,(eqTest (watching (Horizontal [])) "==" HSet.empty)
 ,(eqTest (watching (Vertical [])) "==" HSet.empty)
 ,(eqTest (watching
            (Horizontal [(Window {wname = "olympics", width = 80, height = 33})
                        ,(Window {wname = "local news", width = 20, height = 10})]))
   "==" ((HSet.singleton "olympics") `HSet.union` (HSet.singleton "local news")))
 ,(eqTest (watching
           (Vertical [(Window {wname = "olympics", width = 80, height = 33})
                     ,(Window {wname = "local news", width = 20, height = 10})]))
   "==" ((HSet.singleton "olympics") `HSet.union` (HSet.singleton "local news")))
 ,(eqTest (watching
           (Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                     ,(Window {wname = "olympics", width = 80, height = 33})
                     ,(Window {wname = "local news", width = 20, height = 10})]))
   "==" ((HSet.singleton "Star Trek") `HSet.union` (HSet.singleton "olympics")
         `HSet.union` (HSet.singleton "local news")))
 ,(eqTest (watching
          (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 = "equestrian", width = 70, height = 30})])
             ,(Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                        ,(Window {wname = "olympics", width = 80, height = 33})
                        ,(Window {wname = "local news", width = 20, height = 10})])
             ]))
   "==" ((HSet.unions
          (map HSet.singleton ["Tempest","Othello","Hamlet","baseball","track"
                              ,"equestrian","Star Trek","olympics","local news"]))))
 ]
```

Figure 9: Tests for `Watching`.

11. (15 points) [UseModels]

    This is another problem about Window Layouts. Write a function

    ```
    changeChannel :: String -> String -> WindowLayout -> WindowLayout
    ```

    that takes two strings, `new` and `old` and a Window Layout, `wl`, and returns a Window Layout that is just like `wl`, except that all windows whose `wname` field's value is (== to) `old` in the argument `wl` are changed to `new` in the result.

    Figure 10 on the following page shows examples.

    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: ChangeChannelTests.hs,v 1.2 2013/01/28 02:49:56 leavens Exp leavens $
module ChangeChannelTests where
import WindowLayout
import qualified Data.Set as HSet
import ChangeChannel
import Testing

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

tests :: [TestCase WindowLayout]
tests =
 [(eqTest (changeChannel "goldfinger" "olympics"
                         (Window {wname = "olympics", width = 50, height = 33}))
   "==" (Window {wname = "goldfinger", width = 50, height = 33}))
 ,(eqTest (changeChannel "masterpiece" "dancing" (Horizontal [])) "==" (Horizontal []))
 ,(eqTest (changeChannel "nova" "Star Trek" (Vertical [])) "==" (Vertical []))
 ,(eqTest (changeChannel "masterpiece" "local news"
            (Horizontal [(Window {wname = "olympics", width = 80, height = 33})
                        ,(Window {wname = "local news", width = 20, height = 10})]))
   "==" (Horizontal [(Window {wname = "olympics", width = 80, height = 33})
                    ,(Window {wname = "masterpiece", width = 20, height = 10})]))
 ,(eqTest (changeChannel "Dr. No" "olympics"
           (Vertical [(Window {wname = "olympics", width = 80, height = 33})
                     ,(Window {wname = "local news", width = 20, height = 10})]))
   "==" (Vertical [(Window {wname = "Dr. No", width = 80, height = 33})
                  ,(Window {wname = "local news", width = 20, height = 10})]))
 ,(eqTest (changeChannel "Sienfeld" "local news"
           (Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                     ,(Window {wname = "olympics", width = 80, height = 33})
                     ,(Window {wname = "Sienfeld", width = 20, height = 10})]))
   "==" (Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                  ,(Window {wname = "olympics", width = 80, height = 33})
                  ,(Window {wname = "Sienfeld", width = 20, height = 10})]))
 ,(eqTest (changeChannel "Sienfeld" "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})
                         ,(Window {wname = "local news", width = 100, height = 60})
                         ,(Window {wname = "equestrian", width = 70, height = 30})])
            ,(Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                       ,(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})
                         ,(Window {wname = "Sienfeld", width = 100, height = 60})
                         ,(Window {wname = "equestrian", width = 70, height = 30})])
            ,(Vertical [(Window {wname = "Star Trek", width = 40, height = 100})
                       ,(Window {wname = "olympics", width = 80, height = 33})
                       ,(Window {wname = "Sienfeld", width = 20, height = 10})])
            ])) ]
```

Figure 10: Tests for problem 11.

12. (10 points) [UseModels]

    The problem uses the types `Statement` and `Expression` [Lea13], shown below.

```
-- $Id: StatementsExpressions.hs,v 1.1 2013/01/25 13:44:12 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

```
eqOptim :: Statement -> Statement
```

    that takes a Statement, `stmt`, and returns a Statement just like `stmt`, except that all expressions of the form `EqualsExp (VarExp` x`) (VarExp` x`))` that occur in `stmt` are replaced by `(VarExp "true")`.

    There are test cases contained in `EqOptimTests.hs`, which is shown in Figure 11 on the next 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: EqOptimTests.hs,v 1.1 2013/01/27 22:52:22 leavens Exp leavens $
module EqOptimTests where
import StatementsExpressions
import EqOptim
import Testing

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

tests :: [TestCase Statement]
tests =
 [(eqTest (eqOptim (ExpStmt (EqualsExp (VarExp "x") (VarExp "x"))))
          "==" (ExpStmt (VarExp "true")))
 ,(eqTest (eqOptim (ExpStmt (EqualsExp (VarExp "x") (VarExp "yy"))))
          "==" (ExpStmt (EqualsExp (VarExp "x") (VarExp "yy"))))
 ,(eqTest (eqOptim (ExpStmt (NumExp 7))) "==" (ExpStmt (NumExp 7)))
 ,(eqTest (eqOptim (ExpStmt (VarExp "q"))) "==" (ExpStmt (VarExp "q")))
 ,(eqTest (eqOptim (ExpStmt (VarExp "true"))) "==" (ExpStmt (VarExp "true")))
 ,(eqTest (eqOptim (ExpStmt (BeginExp [] (EqualsExp (VarExp "x") (VarExp "x")))))
   "==" (ExpStmt (BeginExp [] (VarExp "true"))))
 ,(eqTest (eqOptim (AssignStmt "y" (EqualsExp (VarExp "jz") (VarExp "jz"))))
   "==" (AssignStmt "y" (VarExp "true")))
 ,(eqTest (eqOptim (IfStmt (EqualsExp (VarExp "c") (VarExp "c"))
                           (AssignStmt "d" (EqualsExp (VarExp "f") (VarExp "f")))))
   "==" (IfStmt (VarExp "true") (AssignStmt "d" (VarExp "true"))))
 ,(eqTest (eqOptim (AssignStmt "g"
                    (BeginExp [(IfStmt (EqualsExp (VarExp "c") (VarExp "c"))
                                       (AssignStmt "d" (EqualsExp (VarExp "f") (VarExp "f"))))
                               ,(AssignStmt "z" (EqualsExp (VarExp "m") (VarExp "m")))]
                     (BeginExp [AssignStmt "e" (EqualsExp (VarExp "y") (NumExp 2))]
                               (EqualsExp (VarExp "a") (VarExp "a"))))))
   "==" (AssignStmt "g"
         (BeginExp [(IfStmt (VarExp "true") (AssignStmt "d" (VarExp "true")))
                   ,(AssignStmt "z" (VarExp "true"))]
          (BeginExp [AssignStmt "e" (EqualsExp (VarExp "y") (NumExp 2))]
                    (VarExp "true")))))
 ]
```

Figure 11: Tests for `EqOptim`.

13. (20 points) [UseModels]

    The problem uses the types `Statement` and `Expression` [Lea13], as in the previous problem.

    Write a function

    ```
    simplify :: 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 `SimplifyTests.hs`, which is shown in Figure 12 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: SimplifyTests.hs,v 1.1 2013/01/27 22:52:22 leavens Exp leavens $
module SimplifyTests where
import StatementsExpressions
import Simplify
import Testing

main = dotests "SimplifyTests $Revision: 1.1 $" 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 12: Tests for `Simplify`.

14. (20 points) [UseModels]

    The problem uses the types `Text`, from the module `Text` shown below.

```
module Text where

type Sentence = String
data Text = Paragraph [Sentence]
          | Section [Text]
            deriving (Eq, Show)
```

    Write a function

```
textMap :: (Sentence -> Sentence) -> Text -> Text
```

    that takes a function, `f` from Sentences to Sentences, and a Text, `txt`, and produces a Text that has the same shape as `txt`, but with `f` applied to each Sentence in `txt`.

    There are test cases contained in `TextMapTests.hs`, which is shown in Figure 13 on the next 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.

Points

This homework's total points: 155.

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.

```
-- $Id: TextMapTests.hs,v 1.2 2013/01/28 02:30:09 leavens Exp leavens $
module TextMapTests where
import Testing
import Text
import TextMap
import Data.Char

-- helpers for the tests, of type Sentence -> Sentence
capFirst (c:cs) = (toUpper c) : cs
capFirst cs = cs
period sent = sent ++ "."

-- do main to run our tests
main :: IO()
main = dotests "TextMapTests $Revision: 1.2 $" tests

tests :: [TestCase Text]
tests =
 [(eqTest (textMap period (Paragraph [])) "==" (Paragraph []))
 ,(eqTest (textMap period (Section [])) "==" (Section []))
 ,(eqTest (textMap id (Paragraph ["a man","a plan"]))
   "==" (Paragraph ["a man","a plan"]))
 ,(eqTest (textMap capFirst (Section [(Paragraph ["a man","a plan"])]))
   "==" (Section [(Paragraph ["A man","A plan"])]))
 ,(eqTest (textMap period (Section [(Paragraph ["a man","a plan"])
                                   ,(Paragraph ["a canal","Panama"])]))
   "==" (Section [(Paragraph ["a man.","a plan."])
                 ,(Paragraph ["a canal.","Panama."])]))
 ,(eqTest (textMap period
           (Section [Paragraph ["a woman"]
                     ,(Section [Paragraph ["radiation"], (Paragraph ["rays","curie"])])]))
   "==" (Section [Paragraph ["a woman."]
                 ,(Section [Paragraph ["radiation."], (Paragraph ["rays.","curie."])])]))
 ,(eqTest (textMap (period . capFirst)
           (Section [Paragraph ["a woman"]
                     ,(Section [Paragraph ["radiation"], (Paragraph ["rays","curie"])])]))
   "==" (Section [Paragraph ["A woman."]
                 ,(Section [Paragraph ["Radiation."] ,(Paragraph ["Rays.","Curie."])])]))
 ,(eqTest (textMap period
           (Section [(Section [(Section [(Section [(Paragraph ["OK"])])])])]
                     ,(Paragraph ["I am","You are","Me too"]) ,(Section [])]))
   "==" (Section [(Section [(Section [(Section [(Paragraph ["OK."])])])])]
                 ,(Paragraph ["I am.","You are.","Me too."]) ,(Section [])]))
 ]
```

Figure 13: Tests for TextMap.