Homework 2: Functional Programming in Haskell

See Webcourses 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 a book or other source.
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.)
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

[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.

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 1-7 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.
Also read "Following the Grammar with Haskell" [Lea13] and follow its suggestions for planning and 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

Functions on Tuples

1. (5 points) [UseModels] In Haskell write the function

   ```
   sum3 :: (Integer,Integer,Integer) -> Integer
   ```

   which takes a triple of `Integer`s, `(x,y,z)`, and returns an `Integer` that is the sum of `x`, `y`, and `z`. The following are examples, written using the `Testing` module, which are included in the `hw2-tests.zip` file.

   ```
   tests :: [TestCase Integer]
   tests = [eqTest (sum3 (0,0,0)) "==" 0
           ,eqTest (sum3 (0,1,2)) "==" 3
           ,eqTest (sum3 (75,100,50)) "==" 225
           ,eqTest (sum3 (-30,10,557)) "==" 537
           ,eqTest (sum3 (624,986,212532)) "==" 214142
           ,eqTest (sum3 (100,1000,314)) "==" 1414
           ]
   ```

   To run our tests, use the `Sum3Tests.hs` file. To make that work, you have to put your code in a module `Sum3`, which will need to be in a file named `Sum3.hs` (or `Sum3.lhs`), in the same directory as `Sum3Tests.hs`. Your file `Sum3.hs` should thus start as follows.

```
module Sum3 where
sum3 :: (Integer,Integer,Integer) -> Integer
```

Then run our tests by running the `main` function in `Sum3Tests.hs`. Our tests are written using the `Testing.lhs`, which is included in hw2-tests.zip.

```
-- $Id: Sum3Tests.hs,v 1.1 2019/09/09 01:49:40 leavens Exp leavens $
module Sum3Tests where
import Sum3
import Testing

main = dotests "Sum3Tests $Revision: 1.1 $" tests
tests :: [TestCase Integer]
tests = [eqTest (sum3 (0,0,0)) "==" 0
        ,eqTest (sum3 (0,1,2)) "==" 3
        ,eqTest (sum3 (75,100,50)) "==" 225
        ,eqTest (sum3 (-30,10,557)) "==" 537
        ,eqTest (sum3 (624,986,212532)) "==" 214142
        ,eqTest (sum3 (100,1000,314)) "==" 1414
        ]
```

Figure 1: Tests for problem 1.

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

We will take off one point each for type errors, syntax errors and for confusing code. We will take 3 points off for not using the right type of argument (a triple).

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].

2. [Concepts]

   (a) (5 points) In Haskell, which of the following is equivalent to the list `[3,5,3]`?

   1. `(3,5,3)`
   2. `3:(5:3)`
   3. `(3:5):3`
   4. `(((3:5):3):[])`
   5. `3:(5:(3:[]))`

   (b) (10 points) Suppose that ohno is the list `['o', 'h', 'n', 'o']` and that `yikes` is the list `"yikes"`. For each of the following, say whether it is legal or illegal in Haskell, and if it is illegal, say why it is illegal.

   1. `ohno:'y'`
   2. `ohno ++ yikes`
   3. `ohno:yikes`
   4. `['o']:yikes`
   5. `'o':yikes`

   (c) (5 points) Haskell has built in functions **`head`** and **`tail`** defined as follows.

```
head            :: [a] -> a
head (x:_)      =  x
head []         =  error "Prelude.head: empty list"

tail            :: [a] -> [a]
tail (_:xs)     =  xs
tail []         =  error "Prelude.tail: empty list"
```

   For example, **`head`** `[1 ..]` equals `1` and **`tail`** `[1 ..]` equals `[2 ..]`. Consider the following function.

```
dismember lst =
    let first = head lst
    in let rest = tail lst
       in (lst, first:rest)
```

   What is the result of the call `dismember [3,4,7,5,8]`?

   (We suggest that you think about it first, and only use the Haskell system to check your answer.)

3. [UseModels] This problem will have you write a solution in two 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 100 greater than the corresponding element in the argument list.

   (a) (5 points) Write the function

   ```
   add100_list_comp :: [Integer] -> [Integer]
   ```

   that solves the above problem by using a list comprehension.

   (b) (5 points) Write the function

   ```
   add100_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 `Add100ListTests.hs`, which is shown in Figure 2.

```
-- $Id: Add100ListTests.hs,v 1.2 2019/09/10 11:13:39 leavens Exp leavens $
module Add10ListTests where
import Testing
import Add100List  -- you have to put your solutions in module Add100List

version = "Add100ListTests $Revision: 1.2 $"
recursive_tests = (tests add100_list_rec)
comprehension_tests = (tests add100_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 add100_list_comp
test_comprehension :: IO()
test_comprehension = dotests version comprehension_tests

-- do test_recursive to test just add100_list_rec
test_recursive :: IO()
test_recursive = dotests version recursive_tests

tests :: ([Integer] -> [Integer]) -> [TestCase [Integer]]
tests f =
    [(eqTest (f []) "==" [])
    ,(eqTest (f (1:[])) "==" (101:[]))
    ,(eqTest (f (3:1:[])) "==" (103:101:[]))
    ,(eqTest (f [1,5,7,1,7]) "==" [101,105,107,101,107])
    ,(eqTest (f [7 .. 21]) "==" [107 .. 121])
    ,(eqTest (f [8,4,-2,3,1,10000000,10])
          "==" [108,104,98,103,101,10000100,110])
    ]
```

Figure 2: Tests for problem 3. In the definition of `tests`, the name `f` stands for one of the two functions you are to write.

To run our tests, use the `Add100ListTests.hs` file. To make that work, you have to put your code in a module Add100List, which will need to be in a file named Add100List.hs (or Add100List.lhs), in the same directory as Add100ListTests.hs. Your file Add100List.hs should thus start as follows.

```
module Add100List where
add100_list_rec :: [Integer] -> [Integer]
add100_list_comp :: [Integer] -> [Integer]
```

Then run our tests by running the main function in Add100ListTests.hs.

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

You will all lose points for not solving the problem as specified in each part, so be sure to solve the problem in each part in the way specified. We will take points off for inelegant code.

4. (10 points) [UseModels] In Haskell, write the function:

```
squareEvens :: [Integer] -> [Integer]
```

that takes a list of Integers, lst, and returns a list of Integers that is just like lst, except that each even element of lst is replaced by the square of that element. In your solution, you might find it helpful to use the built-in predicate even.

There are examples in Figure 3.

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: SquareEvensTests.hs,v 1.2 2019/09/10 12:38:09 leavens Exp leavens $
module SquareEvensTests where
import SquareEvens
import Testing
main = dotests "SquareEvensTests $Revision: 1.2 $" tests
tests :: [TestCase [Integer]]
tests = [eqTest (squareEvens []) "==" []
        ,eqTest (squareEvens [3]) "==" [3]
        ,eqTest (squareEvens [4,-4]) "==" [16,16]
        ,eqTest (squareEvens [4,3,4]) "==" [16,3,16]
        ,eqTest (squareEvens [1,2,3,4,5,6]) "==" [1,4,3,16,5,36]
        ,eqTest (squareEvens [3,10,3,5,600,0,-2,-1,-3])
                "==" [3,100,3,5,360000,0,4,-1,-3]
        ]
```

Figure 3: Tests for problem 4.

---

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

```
 removeNth :: (Eq a) => Int -> a -> [a] -> [a]
```

that takes as arguments: a positive Int, n, an element, toRemove (of some equality type a), and a list, as, of type [a], and returns a list (of type [a]) that is just like the argument as, but which does not contain the nth occurrence (in as) of the element toRemove.

Your solution must *not* use any Haskell library functions. You may assume that n is strictly greater than 0.

There are test cases contained in RemoveNthTests.hs, which is shown in Figure 4 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: RemoveNthTests.hs,v 1.1 2015/01/21 16:33:59 leavens Exp $
module RemoveNthTests where
import Testing
import RemoveNth

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

tests :: [TestCase [Int]]
tests =
    [(eqTest (removeNth 1 3 []) "==" [])
    ,(eqTest (removeNth 1 3 (1:[])) "==" (1:[]))
    ,(eqTest (removeNth 1 1 (1:[])) "==" [])
    ,(eqTest (removeNth 1 3 (3:1:3:[])) "==" (1:3:[]))
    ,(eqTest (removeNth 2 3 (3:1:3:[])) "==" (3:1:[]))
    ,(eqTest (removeNth 3 3 (3:1:3:[])) "==" (3:1:3:[]))
    ,(eqTest (removeNth 1 3 (3:9:3:7:3:[])) "==" (9:3:7:3:[]))
    ,(eqTest (removeNth 3 3 (3:9:3:7:3:[])) "==" (3:9:3:7:[]))
    ,(eqTest (removeNth 2 1 (3:1:5:1:4:[])) "==" (3:1:5:4:[]))
    ,(eqTest (removeNth 1 7 (3:1:[])) "==" (3:1:[]))
    ,(eqTest (removeNth 1 7 [1,5,7,1,7]) "==" [1,5,1,7])
    ,(eqTest (removeNth 2 7 [1,5,7,1,7]) "==" [1,5,7,1])
    ,(eqTest (removeNth 4 9 [9,2,9,3,9,10,9,5,6]) "==" [9,2,9,3,9,10,5,6])
    ,(eqTest (removeNth 2 8 [8,8,8,8,8,8]) "==" [8,8,8,8,8])
    ,(eqTest (removeNth 17 8 [8,8,8,8,8,8]) "==" [8,8,8,8,8,8])
    ,(eqTest (removeNth 18 99 [8,8,8,8,8,8]) "==" [8,8,8,8,8,8])
    ,(eqTest (removeNth 3 8 [8,2,8,4,8,8,8,8,8]) "==" [8,2,8,4,8,8,8,8])
    ,(eqTest (removeNth 2 20 ([1 .. 50] ++ (reverse [1 .. 50])))
      "==" ([1 .. 50] ++ (reverse ([1 .. 19] ++ [21 .. 50]))))
    ]
```

Figure 4: Tests for problem 5.

6. (5 points) [Concepts] Is it possible to use a list comprehension to solve problem 5 in an easy, direct way? Briefly explain.

7. [UseModels] Complete the module `VecLists` found in the file `VecLists.hs` (provided in the `hw2-tests.zip` file), by writing function definitions in the indicated places that implement the functions: `scale`, `add`, and `sub`. This module represents VecLists by lists of Doubles. The functions you are to implement are as follows.

   (a) (5 points) The function

   ```
   scale :: Double -> VecList -> VecList
   ```

   takes a Double `y`, and a VecList, `v`, and returns a new VecList that is just like `v`, except that each coordinate is `y` times the corresponding coordinate in `v`.

   (b) (5 points) The function

   ```
   add :: VecList -> VecList -> VecList
   ```

   takes two VecLists and adds them together, so that each coordinate of the result is the sum of the corresponding coordinates of the argument VecLists. Your code should assume that the two VecList arguments have the same length.

   (c) (5 points) The function

   ```
   dotprod :: VecList -> VecList -> Double
   ```

   takes two VecLists and computes their dot product (or inner product), which is the sum of the products of the corresponding elements. Your code should assume that the two VecList arguments have the same length.

   There are test cases contained in `VecListsTests.hs`, which is shown in Figure 5 on the next page.

   To run our tests, use the `VecListsTests.hs` file. To make that work, edit your code into the provided file `VecLists.hs`. Our tests use the `FloatTesting` module from `hw2-tests.zip`, which is where `vecWithin` is defined.

   You can use `test_scale`, `test_add`, or `test_dotprod` to test individual functions. Then run all our tests by running the `main` function in `VectorsTests.hs`, and turn in both your code file and the output of our `main` test.

   We will take a point off for overly complicated or repeated code (in each part).

```
-- $Id: VecListsTests.hs,v 1.2 2019/09/10 13:35:17 leavens Exp leavens $
module VecListsTests where
import Testing
import FloatTesting -- contains vecWithin and defines ~=~
import VecLists  -- you have to put your solutions in module VecLists
version = "VecListsTests $Revision: 1.2 $"
-- do main to run our tests
main :: IO()
main = do startTesting version
          errs_scale <- run_test_list 0 scale_tests
          errs_add <- run_test_list errs_scale add_tests
          total_errs <- run_test_list errs_add dotprod_tests
          doneTesting total_errs
-- The following will test one function each
test_scale, test_add, test_dotprod :: IO()
test_scale = dotests "Testing scale $Revision: 1.2 $" scale_tests
test_add = dotests "Testing add $Revision: 1.2 $" add_tests
test_dotprod = dotests "Testing dotprod $Revision: 1.2 $" dotprod_tests

scale_tests :: [TestCase VecList]
scale_tests =
    [(vecWithin (scale 3.14 []) "~=~" [])
    ,(vecWithin (scale 10.0 [1.0, 2.0, 4.0]) "~=~" [10.0, 20.0, 40.0])
    ,(vecWithin (scale 5.3 [1.0 .. 10.0]) "~=~" [5.3, 10.6 .. 53.0])
    ,(vecWithin (scale 2.0 [1.0 .. 100.0]) "~=~" [2.0, 4.0 .. 200.0])
    ,(vecWithin (scale 3.5 [4.0]) "~=~" [3.5*4.0])
    ]
add_tests :: [TestCase VecList]
add_tests =
    [(vecWithin ([] `add` []) "~=~" [])
    ,(vecWithin ([0.0, 100.0, 200.0] `add` [1.0, 2.0, 4.0])
                "~=~" [1.0, 102.0, 204.0])
    ,(vecWithin ([1.0 .. 10.0] `add` [100.0 .. 109.0])
                "~=~" [101.0, 103.0 .. 119.0])
    ,(vecWithin ([1.0 .. 10.0] `add` [11.0 .. 20.0])
                "~=~" ([12.0, 14.0 .. 30.0]))
    ,(vecWithin ([3.5] `add` [10.0]) "~=~" [13.5])
    ,(vecWithin ([3.5,6.2,8.2,5.99] `add` [7.2,9.6,13.1,15.5]) "~=~"
                [10.7,15.8,21.299999999999997,21.490000000000002])
    ,(vecWithin ([-1.0] `add` [40.20]) "~=~" [39.20])
    ]
dotprod_tests :: [TestCase Double]
dotprod_tests =
    [(withinTest ([] `dotprod` []) "~=~" 0.0)
    ,(withinTest ([0.0, 100.0, 200.0] `dotprod` [1.0, 2.0, 4.0])
                "~=~" 1000.0)
    ,(withinTest ([1.0 .. 10.0] `dotprod` [100.0 .. 109.0])
                "~=~" 5830.0)
    ,(withinTest ([1.0 .. 10.0] `dotprod` [11.0 .. 20.0])
                "~=~" 935.0)
    ,(withinTest ([3.5] `dotprod` [4.7]) "~=~" (3.5*4.7))
    ,(withinTest ([3.5,1.0,10.1,599.25] `dotprod` [7.2,9.6,13.1,15.5]) "~=~" 9455.485)
    ,(withinTest ([-1.0] `dotprod` [40.20]) "~=~" (-40.2))
    ]
```

Figure 5: Tests for problem 7.

8. [UseModels] This problem will have you write two functions that deal with the application of binary relations to keys (i.e., the lookup of the values associated with a given key). In this problem binary relations are represented as lists of pairs, as described in the file `BinaryRelation.hs`:

```haskell
-- $Id: BinaryRelation.hs,v 1.3 2019/09/10 13:44:55 leavens Exp leavens $
module BinaryRelation where
-- Binary relations are represented as lists of pairs
type BinaryRelation a b = [(a,b)]
```

In a BinaryRelation, the first part of a pair is called a "key" and the second part of a pair is called a "value."

Your code for the following two functions should go in a module named `ApplyToList` that imports the `BinaryRelation` module. Thus it should start as follows.

```haskell
module ApplyToList where
import BinaryRelation
```

The functions you are to write are the following.

(a) (10 points) Using a list comprehension, write the function

```haskell
applyRel :: (Eq k) => k -> (BinaryRelation k v) -> [v]
```

When given a key value, `ky`, of some equality type `k`, and a BinaryRelation `pairs`, of type `(BinaryRelation k v)` the result is a list of values (of type `v`) that are the values from all the pairs whose key is `ky`. Note that values in the result appear in the order in which they appear in `pairs`.

(b) (10 points) Using recursion (that is, without using a list comprehension or library functions), write the function

```haskell
applyToList :: (Eq k) => [k] -> (BinaryRelation k v) -> [v]
```

When given a list of keys, `keys`, of some equality type `k`, and a BinaryRelation, `pairs`, the result is a list of values from all the pairs in the relation `pairs` whose key is one of the keys in `keys`. Note that the order of the answer is such that all the values associated with the first key in `keys` appear before any of the values associated with a later key, and similarly the values associated with other keys appear before later keys in `keys`. (Hint: You may use `applyRel` in your solution for this problem.)

There are test cases contained in `ApplyToListTests.hs`, which is shown in Figure 6 on the following page. That file imports `Relations.hs`, which is shown in Figure 7 on page 12.

To run our tests, use the `ApplyToListTests.hs` file. To make that work, you have to put your code in a module `ApplyToList`.

As specified on the first page of this homework, turn in both your code file and the output of your testing. Be sure to solve each part in the way specified in that part, or you will lose points. We will also take points off for overly complicated or repeated code.

```
-- $Id: ApplyToListTests.hs,v 1.2 2019/09/10 13:44:55 leavens Exp leavens $
module ApplyToListTests where
import Testing
import BinaryRelation
import Relations  -- some test inputs
import ApplyToList  -- you have to put your solutions in this module

version = "ApplyToListTests $Revision: 1.2 $"

-- do main to run our tests
main :: IO()
main = do startTesting version
          errs_wk <- run_test_list 0 applyRel_tests
          total_errs <- run_test_list errs_wk applyToList_tests
          doneTesting total_errs
-- do test_applyRel to test just applyRel
test_applyRel :: IO()
test_applyRel = dotests ("deleteWithValue " ++ version) applyRel_tests
-- do test_applyToList to test just applyToList
test_applyToList :: IO()
test_applyToList = dotests ("applyToList " ++ version) applyToList_tests

applyRel_tests :: [TestCase [String]]
applyRel_tests =
    [(eqTest (applyRel "Timbuktu" []) "==" [])
    ,(eqTest (applyRel "Ames" us_cities) "==" ["Iowa"])
    ,(eqTest (applyRel "Chicago" us_cities) "==" ["Illinois"])
    ,(eqTest (applyRel "bar" bar_stuff)
                 "==" ["mitzva", "stool", "tender", "keeper"])
    ,(eqTest (applyRel "salad" bar_stuff) "==" ["bar"])
    ,(eqTest (applyRel "foo" bar_stuff) "==" [])
    ]

applyToList_tests :: [TestCase [String]]
applyToList_tests =
    [(eqTest (applyToList ([]::[String]) ([]::(BinaryRelation String String))) "==" [])
    ,(eqTest (applyToList ["foo"] []) "==" [])
    ,(eqTest (applyToList ["foo","bar"] bar_stuff)
                 "==" ["mitzva", "stool", "tender", "keeper"])
    ,(eqTest (applyToList ["salad","bar"] bar_stuff)
                 "==" ["bar", "mitzva", "stool", "tender", "keeper"])
    ,(eqTest (applyToList ["Beijing", "Guangzhou", "Shenzhen", "Tokyo"] city_country)
                 "==" ["China", "China", "China", "Japan"])
    ,(eqTest (applyToList ["Buenos Aires", "Delhi", "Osaka"] city_country)
                 "==" ["Argentina", "India", "Japan"])
    ]
```

Figure 6: Tests for problem 8.

```
-- $Id: Relations.hs,v 1.3 2019/09/11 13:01:08 leavens Exp leavens $
module Relations where
import BinaryRelation
bar_stuff :: BinaryRelation String String
us_cities :: BinaryRelation String String
city_country :: BinaryRelation String String
city_population :: BinaryRelation String Int
city_areakm2 :: BinaryRelation String Int
country_population :: BinaryRelation String Int
bar_stuff = [("bar","mitzva"),("bar","stool"), ("bar","tender"),("salad","bar"),("bar","keeper")]
us_cities = [("Chicago","Illinois"),("Miami","Florida"),("Ames","Iowa")
            ,("Orlando","Florida"),("Des Moines","Iowa")]
-- The following data are from Wikipedia.org, accessed August 19, 2013
city_country =
    [("Beijing","China"),("Buenos Aires","Argentina"),("Cairo","Egypt"),("Delhi","India")
    ,("Dhaka","Bangladesh"),("Guangzhou","China"),("Istanbul","Turkey"),("Jakarta","Indonesia")
    ,("Karachi","Pakistan"),("Kinshasa","Democratic Republic of the Congo"),("Kolkata","India")
    ,("Lagos","Nigeria"),("Lima","Peru"),("London","United Kingdom"),("Los Angeles","United States")
    ,("Manila","Philippines"),("Mexico City","Mexico"),("Moscow","Russia"),("Mumbai","India")
    ,("New York City","United States"),("Osaka","Japan"),("Rio de Janeiro","Brazil")
    ,("Sao Paulo","Brazil"),("Seoul","South Korea"),("Shanghai","China"),("Shenzhen","China")
    ,("Tehran","Iran"),("Tianjin","China"),("Tokyo","Japan")]
city_population =
   [("Tokyo", 37239000),("Jakarta", 26746000),("Seoul", 22868000)
   ,("Delhi", 22826000),("Shanghai", 21766000),("Manila", 21241000)
   ,("Karachi", 20877000),("New York City", 20673000),("Sao Paulo", 20568000)
   ,("Mexico City", 20032000),("Beijing", 18241000),("Guangzhou", 17681000)
   ,("Mumbai", 17307000),("Osaka", 17175000),("Moscow", 15788000)
   ,("Cairo", 15071000),("Los Angeles", 15067000),("Kolkata", 14399000)
   ,("Buenos Aires", 13776000),("Tehran", 13309000),("Istanbul", 12506000)
   ,("Lagos", 12090000),("Rio", 10183000),("London", 9576000)
   ,("Lima", 9400000),("Kinshasa", 9387000),("Tianjin", 9277000)
   ,("Chennai", 9182000),("Chicago", 9104000),("Bengaluru", 9044000)
   ,("Bogota", 9009000)]
city_areakm2 = -- area is measured in square km
    [("Tokyo", 8547) ,("Jakarta", 2784) ,("Seoul", 2163)
    ,("Delhi", 1943) ,("Shanghai", 3497) ,("Manila", 1437)
    ,("Karachi", 803) ,("New York City", 11642) ,("Sao Paulo", 3173)
    ,("Mexico City", 2046) ,("Beijing", 3497) ,("Guangzhou", 3173)
    ,("Mumbai", 546) ,("Osaka", 3212) ,("Moscow", 4403)
    ,("Cairo", 1658) ,("Los Angeles", 6299) ,("Kolkata", 1204)
    ,("Bangkok", 2331) ,("Dhaka", 324) ,("Buenos Aires", 2642)
    ,("Tehran", 1360) ,("Istanbul", 1347) ,("Shenzhen", 1748)
    ,("Lagos", 907) ,("Rio de Janeiro", 2020) ,("Paris", 2845)
    ,("Nagoya", 3820) ,("London", 1623) ,("Lima", 648)
    ,("Kinshasa", 583) ,("Tianjin", 1684) ,("Chennai", 842)
    ,("Chicago", 6856) ,("Bengaluru", 738) ,("Bogota", 414)]
country_population = -- from Wikipedia.org access August 21, 2013
    [("China",1359470000),("India",1232830000),("United States",316497000),("Indonesia",237641326)
    ,("Brazil",193946886),("Pakistan",184013000),("Nigeria",173615000),("Bangladesh",152518015)
    ,("Russia",143400000),("Japan",127350000),("Mexico",117409830),("Philippines",98234000)]
```

Figure 7: Test data for relation problems, the file `Relations.hs`.

9. [UseModels] In this problem you will implement 4 functions that operate on the type `BinaryRelation`, which is defined in the file `BinaryRelation.hs`:

```haskell
-- $Id: BinaryRelation.hs,v 1.3 2019/09/10 13:44:55 leavens Exp leavens $
module BinaryRelation where
-- Binary relations are represented as lists of pairs
type BinaryRelation a b = [(a,b)]
```

Your code should be written in a module named `BinaryRelationOps`, which should import the `BinaryRelation` module, and thus should start as follows.

```haskell
module BinaryRelationOps where
import BinaryRelation
```

You are to write the following functions:

(a) (5 points) The function

```haskell
first :: (BinaryRelation a b) -> [a]
```

projects a binary relation on its first column. That is, it returns a list of all the keys of the relation (in their original order).

(b) (5 points) The function

```haskell
second :: (BinaryRelation a b) -> [b]
```

projects a binary relation on its second column. That is, it returns a list of all the values of the relation (in their original order). (Note that the resulting list may have duplicates even if the original relation had no duplicate tuples.)

(c) (10 points) The function

```haskell
select :: ((a,b) -> Bool) -> (BinaryRelation a b) -> (BinaryRelation a b)
```

takes a predicate and a binary relation and returns a list of all the tuples in the relation that satisfy the predicate (in their original order). Note that the predicate is a function that takes a single pair as an argument. For those pairs for which it returns **True**, the `select` function should include that pair in the result.

(d) (10 points) The function

```haskell
compose :: Eq b => (BinaryRelation a b) -> (BinaryRelation b c)
                 -> (BinaryRelation a c)
```

takes two binary relation and returns their relational composition, that is the list of pairs (a, c) such that there is some pair (a, b) in the first argument binary relation and a pair (b, c) in the second relation argument.

There are test cases contained in `BinaryRelationOpsTests.hs`, which is shown in Figure 8 on the following page. To make this work your code must be in a module named `BinaryRelationOps`.

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.

We will take points off for overly complex or repeated code.

```haskell
-- $Id: BinaryRelationOpsTests.hs,v 1.4 2019/09/10 13:52:48 leavens Exp $
module BinaryRelationOpsTests where
import Testing; import BinaryRelation; import Relations
import BinaryRelationOps  -- you have to put your solutions in this module
version = "BinaryRelationOpsTests $Revision: 1.4 $"
main :: IO() -- do main to run all our tests
main = do startTesting version
          pj1_errs <- run_test_list 0 first_tests
          pj2_errs <- run_test_list pj1_errs second_tests
          select_errs <- run_test_list pj2_errs select_tests
          total_errs <- run_test_list select_errs compose_tests
          doneTesting total_errs
-- do test_f to test just the function named f
test_first, test_second, test_select, test_compose :: IO()
(test_first, test_second, test_select, test_compose) =
   (runts first_tests, runts second_tests, runts select_tests, runts compose_tests)
   where runts :: Show a => [TestCase [a]] -> IO() -- prevents type errors
         runts = dotests version
first_tests :: [TestCase [String]]
first_tests =
    [(eqTest (first []) "==" [])
    ,(eqTest (first bar_stuff) "==" ["bar", "bar", "bar", "salad", "bar"])
    ,(eqTest (first city_country)
      "==" ["Beijing","Buenos Aires","Cairo","Delhi","Dhaka","Guangzhou","Istanbul","Jakarta","Karachi"
           ,"Kinshasa","Kolkata","Lagos","Lima","London","Los Angeles","Manila","Mexico City","Moscow"
           ,"Mumbai","New York City","Osaka","Rio de Janeiro","Sao Paulo","Seoul","Shanghai"
           ,"Shenzhen","Tehran","Tianjin","Tokyo"])]
second_tests :: [TestCase [String]]
second_tests =
    [(eqTest (second []) "==" [])
    ,(eqTest (second bar_stuff) "==" ["mitzva", "stool", "tender", "bar", "keeper"])
    ,(eqTest (second city_country)
      "==" ["China","Argentina","Egypt","India","Bangladesh","China","Turkey","Indonesia","Pakistan"
           ,"Democratic Republic of the Congo","India","Nigeria","Peru","United Kingdom","United States"
           ,"Philippines","Mexico","Russia","India","United States","Japan","Brazil","Brazil"
           ,"South Korea","China","China","Iran","China","Japan"])]
select_tests :: [TestCase (BinaryRelation String String)]
select_tests =
    [(eqTest (select (\(x,y) -> length x > length y) []) "==" [])
    ,(eqTest (select (\(x,y) -> length x <= length y) us_cities)
      "==" [("Chicago","Illinois"),("Miami","Florida"),("Ames","Iowa"),("Orlando","Florida")])
    ,(eqTest (select (\(_,y) -> y == "Iowa") us_cities) "==" [("Ames","Iowa"),("Des Moines","Iowa")])
    ,(eqTest (select (\(x,y) -> x == "Tokyo" && y == "Japan") city_country) "==" [("Tokyo","Japan")])
    ,(eqTest (select (\(c:city,y:country) -> c == y) city_country)
      "==" [("Mexico City","Mexico"),("Seoul","South Korea")])]
compose_tests :: [TestCase (BinaryRelation String Int)]
compose_tests =
    [(eqTest (compose [] country_population) "==" [])
    ,(eqTest (compose bar_stuff [("stool",3),("tender",16)]) "==" [("bar",3),("bar",16)])
    ,(eqTest (compose city_country country_population)
      "==" [("Beijing",1359470000),("Delhi",1232830000),("Dhaka",152518015)
           ,("Guangzhou",1359470000),("Jakarta",237641326),("Karachi",184013000)
           ,("Kolkata",1232830000),("Lagos",173615000),("Los Angeles",316497000)
           ,("Manila",98234000),("Mexico City",117409830),("Moscow",143400000)
           ,("Mumbai",1232830000),("New York City",316497000),("Osaka",127350000)
           ,("Rio de Janeiro",193946886),("Sao Paulo",193946886),("Shanghai",1359470000)
           ,("Shenzhen",1359470000),("Tianjin",1359470000),("Tokyo",127350000)])]
```

Figure 8: Tests for problem 9. These tests use the relations defined in `Relations.hs` (see Figure 7 on page 12).

10. (5 points) [Concepts] [UseModels] Consider the data type `Number` defined below.

```
module Number where
data Number = Zero | One | Two | Three | Four
```

In Haskell, write the polymorphic function

```
rotate :: Number -> (a,a,a,a,a) -> (a,a,a,a,a)
```

in a module named `Rotate`, which takes a Number, `amt`, and a 5-tuple of elements of some type, `(v,w,x,y,z)`, and returns a triple that is circularly rotated to the right by the number of steps indicated by the English word that corresponds to `amt`. That is, when `amt` is `Zero`, then `(v,w,x,y,z)` is returned unchanged; when `amt` is `One`, then `(z,v,w,x,y)` is returned; when `amt` is `Two`, then `(y,z,v,w,x)` is returned; when `amt` is `Three`, then `(x,y,z,v,w)` is returned; finally, when `amt` is `Four`, then `(w,x,y,z,v)` is returned. There are examples in Figure 9.

We will take points off for using strings instead of values of type `Number`, for using `==` on values of type `Number` (which is a type error), for treating tuples as lists, for other type errors, and for incorrect code.

```
-- $Id: RotateTests.hs,v 1.3 2019/09/10 14:51:35 leavens Exp leavens $
module RotateTests where
import Testing
import Number
import Rotate -- your code should go in this module
main = dotests "RotateTests $Revision: 1.3 $" tests
tests :: [TestCase Bool]
tests =
    [assertTrue ((rotate Zero (1,2,3,4,5)) == (1,2,3,4,5))
    ,assertTrue ((rotate One (1,2,3,4,5)) ==  (5,1,2,3,4))
    ,assertTrue ((rotate Two (1,2,3,4,5)) ==  (4,5,1,2,3))
    ,assertTrue ((rotate Three (1,2,3,4,5)) ==(3,4,5,1,2))
    ,assertTrue ((rotate Four (1,2,3,4,5)) == (2,3,4,5,1))
    ,assertTrue ((rotate Two ("jan","feb","mar","apr","may")) == ("apr","may","jan","feb","mar"))
    ,assertTrue ((rotate Three ("jan","feb","mar","apr","may")) == ("mar","apr","may","jan","feb"))
    ,assertTrue ((rotate Four ("jan","feb","mar","apr","may")) == ("feb","mar","apr","may","jan"))
    ,assertTrue ((rotate Zero (True,False,True,False,True)) == (True,False,True,False,True))
    ,assertTrue ((rotate One (True,False,True,False,True)) == (True,True,False,True,False)) ]
```

Figure 9: Tests for problem 10.

11. (10 points) [UseModels] This problem concerns lists of words, which are strings that do not contain blanks.

```haskell
import Prelude hiding (Word)
type Word = String
```

Your task is to write a function

```haskell
txt2sms :: [Word] -> [Word]
```

that takes a list of words `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`,
- the two words `I will` are replaced by the word `I'll`,
- the word `to` is replaced by the word `2`, and
- `great` is replaced by `gr8`.

This list is complete (for this problem).

The examples in Figure 10 on the following page are written using the `Testing` module supplied with the homework. They r also found in our testing file `Txt2smsTests.hs` which u can get from webcourses (in the zip file attached to problem 1). Be sure 2 turn in both ur code and the output of our tests on webcourses.

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 2 form the result list, like: `case w of ...`.

```
-- $Id: Txt2smsTests.hs,v 1.1 2019/09/11 12:27:00 leavens Exp leavens $
module Txt2smsTests where
import Testing
import Txt2sms

main = dotests "Txt2smsTests $Revision: 1.1 $" tests

tests :: [TestCase [String]]
tests =
    [(eqTest (txt2sms []) "==" [])
    ,(eqTest (txt2sms ["got", "to", "go","too","late"])
      "==" ["got", "2", "go","too","late"])
    ,(eqTest (txt2sms ["you","you","you","you"]) "==" ["u","u","u","u"])
    ,(eqTest (txt2sms ["you","know","I","will","see","you","soon"])
      "==" ["u","know","I'll","cya","soon"])
    ,(eqTest (txt2sms ["by","the","way","you","must","see","my","girlfriend","she","is","great"])
      "==" ["btw","u","must","see","my","gf","she","is","gr8"])
    ,(eqTest (txt2sms (["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 (txt2sms ["by","the","way","I","will","be","right","back"])
      "==" ["btw","I'll","brb"])
    ]
```

Figure 10: Tests for problem 11.

Iteration

12. (10 points) [UseModels]

    In Haskell, using tail recursion, write a polymorphic function

    ```
    listMax :: (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 maximal element from `lst`. That is, the result should be an element of `lst` that is not smaller than any other element of `lst`.

    Although you are allowed to use the standard `max` function, your code must *not* use any other library functions.

    In your code, you can assume that the argument list is non-empty and finite. There are test cases contained in `ListMaxTests.hs`, which is shown in Figure 11.

    Note: your solution *must use tail recursion*. You will lose points if your solution is not tail recursive. We will also take points off for repeated code or unnecessary or hard to follow code.

```
-- $Id: ListMaxTests.hs,v 1.2 2019/09/11 12:41:54 leavens Exp leavens $
module ListMaxTests where
import Testing
import ListMax

main = do startTesting "ListMaxTests $Revision: 1.2 $"
          errs <- run_test_list 0 tests_ints
          total <- run_test_list errs tests_chars
          doneTesting total

tests_ints :: [TestCase Int]
tests_ints =
    [(eqTest (listMax (1:1:1:1:1:[])) "==" 1)
    ,(eqTest (listMax (26:[])) "==" 26)
    ,(eqTest (listMax (1:[])) "==" 1)
    ,(eqTest (listMax (1:26:[])) "==" 26)
    ,(eqTest (listMax (26:1:[])) "==" 26)
    ,(eqTest (listMax (1:2:3:4:1:3:5:26:27:[])) "==" 27)
    ,(eqTest (listMax (4:0:2:0:[])) "==" 4)
    ,(eqTest (listMax (86:99:12: -3:[])) "==" 99)
    ,(eqTest (listMax (100000:8600000:12222: -999999:[])) "==" 8600000)
    ]

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

Figure 11: Tests for `listMax`.

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.

13. (10 points) [UseModels] In Haskell, using tail recursion, write a polymorphic function

    ```
    findIndex :: (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 occurrence of `sought` in `lst`. However, if `sought` does not occur in `lst`, it returns `-1`.

    Your code must *not* use any library functions and must be tail recursive. You will lose points if your code is not tail recursive.

    In your code, you can assume that the argument list is finite. There are test cases contained in `FindIndexTests.hs`, which is shown in Figure 12.

```
-- $Id: FindIndexTests.hs,v 1.1 2019/09/11 12:48:37 leavens Exp leavens $
module FindIndexTests where
import FindIndex
import Testing

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

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

Figure 12: Tests for `FindIndex`.

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: 160.

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.