Assignment 8

due at 23:59 on Wed Nov 15 (80 points)

The dropbox link for this assignment is https://www.dropbox.com/request/YYZLvRMyt2lZcFrQrFvl

Start your `a08.hs` with these imports:

```
import qualified Data.Set as S
import qualified Data.Map as M
import Control.Monad (void)
import Data.Time.Calendar (Day, fromGregorian)
```

Folds

In class, we used the function `foldr` to traverse a list, accumulating a result. In that case, we used it to compute a frequency analysis using a map, but it's a very general tool. This is essentially the definition of `foldr`, but we'll call it `foldList` to avoid a name conflict:

```
foldList :: (a -> b -> b) -> b -> [a] -> b
foldList f z [] = z
foldList f z (x:xs) = f x (foldList f z xs)
```

Let's trace how this works when applied to the following arguments.

```
foldList (+) 10 [5,8,19]
```

So the argument `f` is bound to the operator `(+)`, and the argument `z` is bound to the number `10`. We pattern-match on the list in the usual way, so it will be simpler to trace if we rewrite the list `[5,8,19]` using the 'cons' operator notation: `5:8:19:[]`:

```
foldList (+) 10 (5:8:19:[]) =
(+) 5 (foldList (+) 10 (8:19:[])) =
(+) 5 ((+) 8 (foldList (+) 10 (19:[]))) =
(+) 5 ((+) 8 ((+) 19 (foldList (+) 10 []))) =
(+) 5 ((+) 8 ((+) 19 10)) =
(+) 5 ((+) 8 29) =
(+) 5 37 =
42
```

So it adds, starting with `10` in the base case, and then adding each element of the list.

1. Using this definition:

```
g :: Int -> Int -> Int
g x y = x + y*y
```

   Trace the evaluation of

```
foldList g 1 [2..4]
```

   step-by-step. Type your trace into comments in your `a08.hs` file.

2. In the previous example, the function g has type `Int -> Int -> Int`; but the most general type supported by `foldList` is `a -> b -> b`, so the first two arguments don't *have* to be the same type. So here's another example where we accumulate something different:

```
snoc :: Char -> String -> String
snoc c s = s ++ [c]
```

   Trace the evaluation of:

```
foldList snoc "z" "arbe"
```

3. Define `foldLeft` with this type, which applies the given function to the list elements, but the arguments are in a different order:

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

   So that when you apply it with g as before, you get:

```
ghci> foldLeft g 1 [2..4]
30
```

   That's because it should calculate

```
g (g (g 1 4) 3) 2
```

   instead of:

```
g 2 (g 3 (g 4 1))
```

Maps and joins

The `Data.Map` module [hackage documentation] defines a type `Map k a` for a key-value store. The type variable `k` represents the type of keys (which must be an instance of `Ord`), and `a` represents the type of values. Let's use this to store a list of contacts with user IDs and full names:

```
type UserId = Int
data Name = Name { firstName, lastName :: String }
  deriving Eq

instance Show Name where
  show n = firstName n ++ " " ++ lastName n

people :: M.Map UserId Name
people = M.fromList
  [ (13, Name "Alice"   "Arcila")
  , (21, Name "Ben"     "Berman")
  , (40, Name "Carol"   "Cornwall")
  , (58, Name "Panda"   "Patel")
  , (71, Name "Randall" "Rivera")
  ]
```

The `Map` type implements `Show` as long as its key and value types do, so you can see the entire map just by typing the variable name:

```
ghci> people
fromList [(13,Alice Arcila),(21,Ben Berman),(40,Carol Cornwall),
(58,Panda Patel),(71,Randall Rivera)]
```

but let's define a specialized function to produce cleaner output:

```
printMap :: (Show k, Show a) => M.Map k a -> IO ()
printMap = void . M.traverseWithKey (\k a ->
  putStrLn $ show k ++ ": " ++ show a)
```

Here is the result of `printMap`:

```
ghci> printMap people
13: Alice Arcila
21: Ben Berman
40: Carol Cornwall
58: Panda Patel
71: Randall Rivera
```

Perhaps we'll use the same set of integer keys to create other data stores, such as birth dates and Twitter handles:

```
birthdays :: M.Map UserId Day
birthdays = M.fromList
  [ (21, fromGregorian 1984 10 15)
  , (40, fromGregorian 1990  3 24)
  , (71, fromGregorian 1979 12  4)
  ]

type TwitterId = String

followers :: M.Map UserId TwitterId
followers = M.fromList
  [ (11, "@anndavis")
  , (13, "@aliccea")
  , (21, "@bb")
  , (31, "@liveness")
  , (58, "@trailblazer")
  ]
```

We're using the `Day` type from module `Data.Time.Calendar`. It represents a particular date but without any time. You construct them using `fromGregorian`, which takes year, month, then day as integers.

Where the user IDs match up in these three data sets, they indicate the same user. So user 21 (Ben Berman) was born on 1984-10-15, and is known on Twitter as '@bb'. But not every user has a known birth date. Not every user has a Twitter handle. And not every Twitter handle is matched to a full name!

What we have here is a classic database problem – that of **joining** data sets where there may be some keys missing from one set or another.

Your task is to define a set of 'join' functions with these signatures:

```
innerJoin :: Ord k => M.Map k a -> M.Map k b -> M.Map k (a,b)
leftJoin  :: Ord k => M.Map k a -> M.Map k b -> M.Map k (a, Maybe b)
rightJoin :: Ord k => M.Map k a -> M.Map k b -> M.Map k (Maybe a, b)
fullJoin  :: Ord k => M.Map k a -> M.Map k b -> M.Map k (Maybe a, Maybe b)
```

We can tell a lot from the types. These functions take two maps whose key types are the same (`k`), but whose value types can differ (`a` and `b`). They return a map where the values are matching pairs of `a` and `b`, but the matches account for missing keys in different ways.

An **inner** join only returns pairs for which the key is present in both maps:

```
ghci> printMap $ innerJoin people followers
13: (Alice Arcila,"@aliccea")
21: (Ben Berman,"@bb")
58: (Panda Patel,"@trailblazer")
```

so Carol is missing from this result because she has no Twitter handle, and '@anndavis' is missing because she doesn't have a name in `people`.

A **left** join returns a pair for each key in the *left* table, and uses `Nothing` (or NULL in a database system) if it's missing from the *right* table:

```
ghci> printMap $ leftJoin people followers
13: (Alice Arcila,Just "@aliccea")
21: (Ben Berman,Just "@bb")
40: (Carol Cornwall,Nothing)
58: (Panda Patel,Just "@trailblazer")
71: (Randall Rivera,Nothing)
ghci> printMap $ leftJoin followers people
11: ("@anndavis",Nothing)
13: ("@aliccea",Just Alice Arcila)
21: ("@bb",Just Ben Berman)
31: ("@liveness",Nothing)
58: ("@trailblazer",Just Panda Patel)
```

A **right** join is just the opposite:

```
ghci> printMap $ rightJoin birthdays followers
11: (Nothing,"@anndavis")
13: (Nothing,"@aliccea")
21: (Just 1984-10-15,"@bb")
31: (Nothing,"@liveness")
58: (Nothing,"@trailblazer")
ghci> printMap $ rightJoin followers birthdays
21: (Just "@bb",1984-10-15)
40: (Nothing,1990-03-24)
71: (Nothing,1979-12-04)
```

Finally, a **full** join returns a pair for each key that appears in *either* table. But that means that either side can be NULL.

```
ghci> printMap $ fullJoin people followers
11: (Nothing,Just "@anndavis")
13: (Just Alice Arcila,Just "@aliccea")
21: (Just Ben Berman,Just "@bb")
31: (Nothing,Just "@liveness")
```

```
40: (Just Carol Cornwall,Nothing)
58: (Just Panda Patel,Just "@trailblazer")
71: (Just Randall Rivera,Nothing)
```

To help you out, here are all the functions from `Data.Set` and `Data.Map` that I used in my solution, though you're free to user other functions if you wish.

- `M.empty` (doc) The empty map.
- `M.insert` (doc) Insert a new key and value in the map.
- `M.intersectionWith` (doc) Intersection of two maps, with a combining function.
- `M.keysSet` (doc) Return all keys in the map, as a set.
- `M.lookup` (doc) Lookup the value at a key in the map
- `M.mapWithKey` (doc) Map a function over all values in the map.
- `S.foldr` (doc) Fold the elements in the set using the given operator. Accumulates a result over the whole set, just like `foldList` does on lists.
- `S.union` (doc) The union of two sets.