```haskell
{-
Kimberly Keller
Monad Exercises
9/12/18
-}

import Control.Monad
import Data.List
import Data.Char

data Btree a = Leaf a | Fork (Btree a) (Btree a) deriving (Eq, Show)

test :: Btree Int
test = Fork (Leaf 1) (Leaf 2)

instance Functor Btree where
    fmap f (Leaf a) = Leaf (f a)
    fmap f (Fork lt rt) = Fork (fmap f lt) (fmap f rt)

iota' :: [Int] -> Btree Int
iota' [n] = Leaf n
iota' ns = Fork left right
            where
                left = iota' (fst zs)
                right = iota' (snd zs)
                y = (length ns) `div` 2
                zs = splitAt y ns

iota :: Int -> Btree Int
iota x = iota' [1..x]


instance  Applicative Btree where
    (<*>) = ap
    pure = return

instance Monad Btree where
    return = Leaf
    (>>=) (Leaf a) f =  f a
    (>>=) (Fork a b) f = (Fork ((>>=) a f) ((>>=) b f))

newtype Lulz a = Lulz {runLulz :: [[a]]} deriving (Eq, Show)

instance Functor Lulz where
    fmap f (Lulz a) = Lulz [ [f x | x <- ks] | ks <- a ]

-- This produces an n x n list of lists, but with the same list
repeated n times.
rho :: Int -> Lulz Int
rho n = Lulz [ [i | i<-[1..n]] | j <- [1..n] ]
```

```
{-
I wrote several incorrect, incomplete implementations of join for
Lulz.
You can see the thought progression as I worked my way toward writing
one that
uses recursion.
-}

-- Doesn't really work at all
joinLulz' :: Lulz (Lulz a) -> Lulz a
joinLulz' xs = Lulz (concat (fmap runLulz (concat (runLulz xs))))

-- Works specifically for test4
joinLulz :: Lulz (Lulz a) -> Lulz a
joinLulz xs = Lulz (fmap concat (runLulz ks)) where ks = (fmap
(concat . runLulz) xs)


instance  Applicative Lulz where
    (<*>) = ap
    pure = return

instance Monad Lulz where
    return a = Lulz [[a]]
    -- Definition of bind with my broken joinLulz
    -- (>>=) lulz f = joinLulz (fmap f lulz)
    -- Trying to define recursiely, but this doesn't compile
    -- (>>=) (Lulz a) f = Lulz ((>>=) (concatMap f a) f)


test1 :: Lulz Int
test1 = Lulz [[1,2], [1,2]]

test2 :: Lulz Int
test2 = Lulz [[1,2,3],
              [1,2,3],
              [1,2,3]]


test3 :: Lulz Char
test3 = Lulz [['a', 'b', 'c'], ['a', 'b', 'c'], ['a', 'b', 'c']]

test4 :: Lulz (Lulz Int)
test4 = Lulz [[Lulz [[1,2,3],
                     [1,2,3],
                     [1,2,3]]],
              [Lulz [[1,2,3],
                     [1,2,3],
                     [1,2,3]]],
```

```
               [Lulz [[1,2,3],
                      [1,2,3],
                      [1,2,3]]]]

test5 :: Lulz [Int]
test5 = (fmap (concat . runLulz) test4)

{-
Little Tests For Monad Laws (using joinLulz for definition of bind)
-}

test6 :: Lulz Int
test6 = (rho >=> (rho >=> rho)) 2

test7 :: Lulz Int
test7 = ((rho >=> rho) >=> rho) 2

test8 :: Lulz Int
test8 = (return >=> rho) 2

test9 :: Lulz Int
test9 = (rho >=> return) 2

test10 :: Btree Int
test10 = (iota >=> (iota >=> iota)) 2

test11 :: Btree Int
test11 = ((iota >=> iota) >=> iota) 2

test12 :: Btree Int
test12 = (return >=> iota) 2

test13 :: Btree Int
test13 = (iota >=> return) 2

{-
Test Cases Run in GHCI:

*Main> test6
Lulz {runLulz = [[1,1,1,2,1,2,1,1,2,1,2],[1,1,1,2,1,2,1,1,2,1,2]]}
*Main> test7
Lulz {runLulz = [[1,1,1,2,1,2,1,1,2,1,2],[1,1,1,2,1,2,1,1,2,1,2]]}
*Main> test6 == test7
True
*Main> test8
Lulz {runLulz = [[1,2,1,2]]}
*Main> test9
Lulz {runLulz = [[1,2],[1,2]]}
*Main> test8 == test9
False
```

```
*Main> rho 2
Lulz {runLulz = [[1,2],[1,2]]}
*Main> test10
Fork (Leaf 1) (Fork (Leaf 1) (Fork (Leaf 1) (Leaf 2)))
*Main> test11
Fork (Leaf 1) (Fork (Leaf 1) (Fork (Leaf 1) (Leaf 2)))
*Main> test10 == test11
True
*Main> test12
Fork (Leaf 1) (Leaf 2)
*Main> test13
Fork (Leaf 1) (Leaf 2)
*Main> test12 == test13
True
*Main> iota 2
Fork (Leaf 1) (Leaf 2)
-}
```