1. The Tree Monad

The Tree type defined in the lecture can actually be interpreted as a monad—we can think of a value of type `Tree a` as a computation delivering a tree of results, such as one might use in a search algorithm, for example. Write the Monad instance for this type.

You can check your definition by copying the following into the file `Tree.hs`, and adding your definition to it.

```
module Tree where
import Control.Monad
import Test.QuickCheck
import MonadLaws

data Tree a = Leaf a | Branch (Tree a) (Tree a)
  deriving (Eq, Show)

instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = frequency [(2,liftM Leaf arbitrary),
                         (1,liftM2 Branch arbitrary arbitrary)]
  shrink (Branch l r) =
    [l,r]++map (Branch l) (shrink r)++map (`Branch`r) (shrink l)
  shrink (Leaf a) = map Leaf (shrink a)

prop_TreeLeftUnit  = prop_LeftUnit  :: PropLeftUnit Tree
prop_TreeRightUnit = prop_RightUnit :: PropRightUnit Tree
prop_TreeAssoc     = prop_Assoc     :: PropAssoc Tree
```

As well as a definition of the `Tree` type, this code defines how to generate random trees (`arbitrary`)[1], and how to simplify a tree in a failed test (`shrink`). It also defines property instances for testing the monad laws. These property instances use another module, `MonadLaws`, that you will need to copy into the file `MonadLaws.hs`.

```
module MonadLaws where
import Test.QuickCheck
import Test.QuickCheck.Function

type PropLeftUnit m = Integer -> Fun Integer (m Integer) -> Bool

prop_LeftUnit x (Fun _ f) = (return x >>= f) == f x

type PropRightUnit m = m Integer -> Bool

prop_RightUnit m = (m >>= return) == m

type PropAssoc m = m Integer ->
                   Fun Integer (m Integer) ->
                    Fun Integer (m Integer) ->
                    Bool
```

[1] Leaves are given a higher weight than branches to ensure that random generation terminates.

```
prop_Assoc m (Fun _ f) (Fun _ g) =
  ((m >>= f) >>= g)
  ==
  (m >>= \x -> f x >>= g)
```

Add your definition of the `Monad` instance for `Tree`, load `Tree.hs` into `ghci`, make sure it type-checks, and then test the laws.

I'm not familiar with Haskell—how do I do that?

Install the Haskell Platform. Once you've done so, on Windows it's convenient to use WinGHCi, which you will find in the Haskell Platform folder in the Start menu. On Linux/Mac just type ghci in the shell.

To load a file into ghci, type either

```
:l <filename without the .hs>
```

or, in WinGHCi, use the `File->Load`… menu item. Imported modules will also be loaded, and any compile-time errors will be reported.

Once you have loaded your code, you can evaluate expressions at the ghci prompt. To test the monad laws, just type

```
quickCheck prop_TreeLeftUnit
```

and so on.

2. The State Monad

Copy the following code into `State.hs`.

```
module State where

import Control.Monad
import Test.QuickCheck

newtype State s a = MkState {unState :: s -> (a,s)}

instance Monad (State s) where
  return x = MkState (\s -> (x,s))
  MkState f >>= g = MkState (\s -> let (a,s') = f s in
                                   unState (g a) s')

get :: State s s
get = undefined

put :: s -> State s ()
put s = undefined

(===) ::
  Eq a => State Integer a -> State Integer a -> Integer -> Bool
(f === g) s = unState f s == unState g s

prop_get_get =
  do x <- get
```

```
    y <- get
    return (x,y)
===
do x <- get
   return (x,x)
```

It defines the state monad presented in the lecture, but using a `newtype` definition rather than a type synonym, so that it is accepted by GHC. Instead of a `tick` operation to increment the state, it defines a `get` operation to read the state, and a `put` operation to write the state. But these definitions are incomplete—finish them.

It is a little awkward to generate random values of the `State` type (we would need to generate values containing a QuickCheck Fun, and convert them into the State type as needed), so instead we use QuickCheck to formulate and test some properties of `put` and `get`. The `(===)` operator checks that two values of type `State Integer a` deliver the same value and final state, when invoked in the same state. It can be used to define properties such as `prop_get_get`, which says that two consecutive `get`s are equivalent to a single one. Check that this property passes (using `quickCheck prop_get_get`), and add and test properties relating `put`s and `get`s in either order, and relating two `put`s to one `put`.

3. Readers and Writers

The state monad lets us thread a state through a program, passing it in to each computation, and returning a new state after each one. Sometimes, though, we need only to do one of these things.

The `Reader s` monad lets us pass a state *into* a computation, but does not let us modify it:

```
newtype Reader s a = MkReader {unReader :: s -> a}

ask :: Reader s s
```

The Writer s monad lets us return state *from* a computation, but does not let us read it. Since we may write the state many times in a computation, we collect a *list* of state values:

```
data Writer s a = MkWriter [s] a

tell :: s -> Writer s ()
```

Write Monad instances for `Reader s` and `Writer s`, and definitions of `ask` and `tell`, and make sure they type-check.

The version of the Writer monad in the Haskell libraries is a little more general than this: is allows the state to be of any *monoid* type, rather than just lists; we need to be able to combine states (a binary operator) in `(>>=)`, we need an "empty" state to use in `return`, and the monad laws demand that these form a monoid.

4. The List Monad

Haskell's list type is also a monad—think of it as representing computations that return zero, one, or more answers. Try to define a suitable `Monad` instance for lists[2]:

```
instance Monad [] where
  return x = …
  xs >>= f = …
```

Sadly, you cannot compile this definition yourself, because a monad instance for lists is already defined. Instead, you will need to define an isomorphic list type `MyList`, and write your instance for that. Copy the following code into `MyList.hs`, add a suitable `Monad` instance, and test the monad laws.

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
module MyList where

import Control.Monad
import Test.QuickCheck
import MonadLaws

newtype MyList a = MkList {unList :: [a]}
  deriving (Eq, Show, Arbitrary)

prop_MyListLeftUnit  = prop_LeftUnit  :: PropLeftUnit MyList
prop_MyListRightUnit = prop_RightUnit :: PropRightUnit MyList
prop_MyListAssoc     = prop_Assoc     :: PropAssoc MyList

mymap f (MkList xs)     = MkList (map f xs)

flat :: MyList (MyList a) -> MyList a
flat (MkList xs) = MkList (concat (map unList xs))
```

This code defines the `MyList` type, inheriting test data generation from the underlying list type via "`deriving Arbitrary`"—the pragma at the top enables this useful extension to Haskell 98. It defines monomorphic properties for testing the monad laws, and then a pair of useful auxiliary functions for writing a `Monad` instance. Add your own `Monad` instance, and test the stated properties, now. Try varying your definitions, and see whether the monad laws then fail.

Once you have experimented a little, read the next section of the exercise.

It turns out that there *is* another plausible way to write a `Monad` instance for `MyList`. If you found the expected definition, then the `flat` function will have played a central role in the definition of `(>>=)`. This function just concatenates a list of lists to obtain a list of all the elements. But if we think of a list of lists as a matrix, then there is another natural way to extract a list of elements—we can just take the diagonal of the matrix! Add the following code to your file:

```
diag :: MyList (MyList a) -> MyList a
diag (MkList (MkList (x:xs):xss)) =
  mycons x (diag (MkList (map mydrop xss)))
```

---

[2] The parameterised list type in Haskell is written `[]`, so `[] a` is the same type as `[a]`.

```
diag _ = MkList []

mycons x (MkList xs)    = MkList (x:xs)

mydrop (MkList (x:xs)) = MkList xs
mydrop (MkList [])     = MkList []
```

Now, what if we replace the concatenation in `(>>=)` with the diagonal—do we get another, different monad?

It turns out that we have to modify the `return` function too; the right definition of `return` in this case should construct an *infinite* list.

```
return x = MkList (repeat x)
```

This in turn means that we may need to *compare* infinite lists as we test the monad laws. Of course, this will not terminate—so let us *replace* the definition of equality for `MyList` with an approximate equality that only compares the first 100 elements. This is good enough for testing the monad laws; we expect that any failures will be observable in far fewer than 100 elements. Add the following instance definition to your code, and remove the "`Eq,`" from the deriving clause.

```
instance Eq a => Eq (MyList a) where
  MkList xs == MkList ys = take 100 xs == take 100 ys
```

Complete a `Monad` instance using `diag`. Is the result really a monad?