DM580 – EXERCISE SHEET #9

(1) Write a function `maybePair :: Maybe a -> Maybe b -> Maybe (a,b)` in two ways: one that uses pattern matching on inputs, and one that uses `do`-notation. Which one do you prefer and why? The one that uses `do`-notation can be given a more general type, `Monad m => m a -> m b -> m (a,b)` – why is that?

(2) Write a function `seqM :: Monad m => [m a] -> m [a]` that sequences a list of monadic actions into a single monadic action.

(3) Write a function `(>->) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c` which combines two functions that return monad actions. For example, `readFile >-> putStrLn` should be the function that accepts a file name and prints the contents of that file to the screen.

(4) Recall the function given by

```
maybemap :: (a -> b) -> Maybe a -> Maybe b
maybemap f Nothing  = Nothing
maybemap f (Just x) = Just (f x)
```

from the lecture. Argue that `maybemap id = id` and `maybemap f . maybemap g = maybemap (f . g)`, thus making `Maybe` a valid instance of `Functor` with `fmap = maybemap`.

(5) Argue that `map id = id` and `map f . map g = map (f . g)`, thus making lists a valid instance of `Functor` with `fmap = map`. You may use that `map` can be defined as

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

Hint: In both cases you have to use induction on the length of the list. In the base case you have to show the property for the empty list. In the inductive case, you assume that the property holds for all lists of length n (such as `xs`), and show that it must also hold for all lists of length $n+1$ (such as `x:xs`).

(6) Recall the type of binary trees.

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

(a) Make `Tree` an instance of `Functor` and `Applicative` using your functions `treemap` and `treeapply` from Exercise Sheet #7 (if you didn't do this, now would be a good time to do it!). How should `pure` be defined?

(b) Make `Tree` an instance of `Monad` by defining `(>>=)`. Think about what the function should do before writing any code.

(7) A program is said to be *invertible* when inputs can always be uniquely determined from outputs. For example, the function `(+1)` is invertible (since, when the output is n, the input must've been $n-1$), while the function `(<3)` is not (since there are many different integers that are less than three, and many that are not).

A simple way to make a function invertible is to *log* all of its inputs (by storing them in a list off to the side), such that these are remembered. The logged inputs can then be returned alongside the outputs. The goal of this exercise is to develop a monad with an interface that enables this kind of logging. Define `Logger b a` to be the type of values of type `a` with a list of values of type `b` off to the side, i.e.,

```
newtype Logger b a = Logger { runLogger :: (a,[b]) }
```

(a) Make `Logger b` an instance of `Functor`, `Applicative`, and `Monad`. Before you start writing anything, think about how the functions involved in these type classes should manage the log.
Hint: To extract the value and log embedded in a value of type `Logger b a`, you can use the function `runLogger :: Logger b a -> (a,[b])` defined in the `newtype`-declaration.

(b) Write a function `writeToLog :: b -> Logger b ()` that writes a value to the log.

(c) Test out your implementation by making some example programs and running them. For example, with

```
add :: Int -> Int -> (Int, [Int])
add n m = runLogger $ do writeToLog n
                         writeToLog m
                         return (n+m)
```

it should be the case that `add 3 5 == (8,[3,5])`. What happens if you swap the order in which you write values to the log?