CS119 – Module 14
Due Date: May 5

Purpose: Monads are a mind-blowing construct that allows us to use "state" within functional programs.

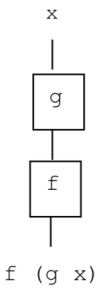
Knowledge: This lab will help you become familiar with the following content knowledge:

- Monad

Activity: With your group perform the following tasks and answer the questions. You will need to use files in the `lab14` directory. You will be reporting your answers back to the class in 30 minutes.

1. So consider this problem in a pure functional language: we have functions f and g that both map floats to floats. In Haskell, f and g might have types given by
   ```
   f,g :: Float -> Float
   ```

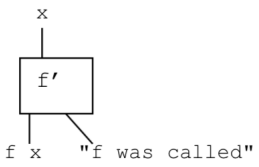
   Now suppose that we want to compose these functions as described in the diagram. The function g gets input x. The output of g is then the input of the function f:

   Try this out with the functions f and g that I have provided in the module Example14. Use example of composition:
   ```
   > f (g 2)
   ```

2. Now suppose we'd like to modify these functions to also output strings for debugging purposes. How can we modify the types of f and g to admit side effects? Well there really isn't any choice at all. If we'd like f' and g' to produce strings as well as floating point numbers as output, then the only possible way is for these strings to be returned alongside the floating point numbers. In other words, we need f' and g' to be of type:
   ```
   f',g' :: Float -> (Float,String)
   ```

   We can draw this diagrammatically as



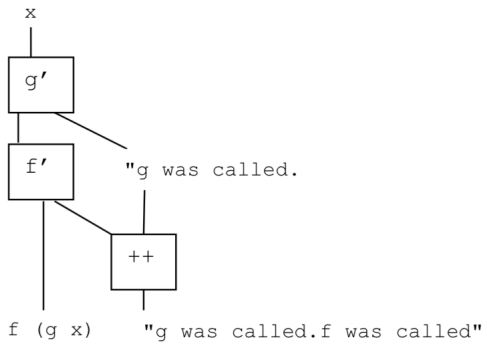
   We can think of these as 'debuggable' functions.

   But our debuggable functions aren't quite so straightforward to deal with. Try composing the functions f' and g' that have been provided. What happens when you try evaluating f'(g'2)? Why does that happen?

3. We need some additional plumbing to compose these functions. We'd like the strings returned by f' and g' to be concatenated into one longer debugging string (the one from f' after the one from g'). The plumbing could look something like the following, in which the `let` expression behaves in the same way as a `where` but kind of written in reverse. I am using the `let` rather than the `where` to emphasize the process.

```
h x = let (y,s) = g' x
          (z,t) = f' y
in (z,s++t)
```

   Here's how it looks diagrammatically:



   Try out the function h which is the composition of f' and g'.

4. This is hard work every time we need to compose two functions and if we had to implement this kind of plumbing all the way through our code it would be a pain. What we need is to define a higher order function to perform this plumbing for us. As the problem is that the output of g' can't simply be plugged into the input of f', we need to "upgrade" f'. So we introduce a function, `bind`, to do this. In other words we'd like

   ```
   bind :: (Float -> (Float,String)) -> ((Float,String) -> (Float,String))
   ```

   Suppose that `(gx,gs)` is the value returned by `(g x)`. We want it so that `bind f' (gx,gs)` returns the pair `(f (g x), "g was called,.f was called.")`

   Another way to look at it, by using partial application, is `bind` converts the function f' to a function which can easily be composed: `bind f' :: (Float,String) -> (Float,String)`

   The function bind must serve two purposes: it must (1) apply f' to the correct part of `(g' x)` and (2) concatenate the string returned by g' with the string returned by f'.

   Complete the function bind.

   ```
   bind f' (gx,gs) = let (fx,fs) = f' __________ in (__________ , __________)
   ```

   Use example:
   ```
   bind f' (g' 2)
   ```

5. Hurray! One function, `bind`, allows us to nicely compose any debuggable functions.

   But wait! What if I want to compose an ordinary function, like f, with a debuggable function, like g'? In order to do that I would need to somehow "lift" the ordinary function into a debuggable one, producing the empty string as a side effect since we don't know what the debug string should be.

   Consider the following definitions:

   ```
   unit :: Float -> (Float,String)
   unit x = (x, "")

   lift :: (Float -> Float) -> Float -> (Float -> String)
   lift f x = (unit (f x))
   ```

   Try:
   ```
   bind (lift f) (g' 2)
   ```

   In summary: the functions, `bind` and `unit`, allow us to compose debuggable functions in a straightforward way, and compose ordinary functions with debuggable functions in a natural way. You have defined your first monad!

6. Haskell provides some Monad syntax for us:

```
newtype Debuggable a = Debug(a,String) deriving Show
instance Monad Debuggable where
  return x = Debug(x,"")
  Debug(gx,gs) >>= f' = let Debug(fx,fs) = f' gx in Debug(fx,gs++fs)
```

We defined a new type and declared it a Monad. In doing so we need to define `unit` and `bind`, which are now called `return` and `>>=`. These definitions are as before with the slight modification that bind is now an infix operator and the value returned by evaluating g' comes before f'.

Try this out to make sure it works with the functions g'' and f'' provided:
`> (g'' 2) >>= f''`

Also lets try lifting our original function by using `return`:
`> (return (g 2)) >>= f''`

Activity: With your group perform the following tasks and answer the questions. You will need to use file `Dice.hs` in the `lab14` directory. You will be reporting your answers back to the class in 30 minutes.

1. Suppose we want to simulate rolling a die which generates a random number from 1 to 6. A way to get a random number is to get a large number, which we call a "seed" and scramble it in some way. The scrambling gives us a random number but also needs to give us a new seed to be used for the next time we generate a random number.

   Consider the following:

   ```
   type Seed = Int
   randomNext :: Seed -> Seed
   randomNext rand = if newRand > 0
       then
          newRand
       else
          newRand + 2147483647
       where
          newRand = 16807*lo - 2836*hi
          (hi,lo) = rand `divMod` 127773

   rollDie1 :: Seed -> (Int,Seed)
   rollDie1 seed = ((seed `mod` 6) + 1, randomNext seed)
   ```

   The function `randomNext` is just scrambling the seed.
   Why does `rollDie1` return both an Int and a Seed?

2. You can view the seed as the "state" of our program and it has to be remembered as we continue to generate random numbers. We "remember" this state by threading it in and out of all of our functions. To see this in action look at the function:

   ```
   sumTwoDice1 :: Seed -> (Int,Seed)
   sumTwoDice1 seed0 =
      let
          (die1,seed1)=rollDie seed0
          (die2,seed2)=rollDie seed1
      in
          (die1+die2,seed2)
   ```

   Try it out and by supplying an initial seed (any number you want) and seeing the output as the sum of the two rolls and a new seed.

3. Now to turn our random numbers into a monad! Our random type is actually going to be a function (just like `rollDie` and the bind operator is going to look like what we did with `sumTwoDice`. We also have a function `apply` which executes the random function.

```
newtype Random a =MakeRandom(Seed -> (a,Seed))

apply :: Random a -> Seed -> (a,Seed)
apply (MakeRandom f) seed = f seed

instance Monad Random where
return x = MakeRandom(\seed->(x,seed))
m >>= g = MakeRandom(\seed0->
   let
      (result1,seed1) = apply m seed0
      (result2,seed2) = apply (g result1) seed1
   in (result2,seed2))
```

Let's see how this would be used:

```
rollDie :: Random Int
rollDie = MakeRandom(\seed -> ((seed `mod` 6) + 1, randomNext seed))

sumTwoDice :: Random Int
sumTwoDice = rollDie >>= (\die1->rollDie >>= (\die2 -> return (die1+die2)))
```

Remember that `>>=` reaches into the monad `rollDie` to get the value `die1` and all the while the monad threads the state for us!

You can try out `sumTwoDice` by using the `apply` function with some seed. For example,

```
apply sumTwoDice 123456789
```

Explain the result that you got.

4. Do you find the notation for `>>=` a bit difficult to read? You are not alone. For that reason Haskell provides some "syntactic sugar" which is just a different way to write the same code. An equivalent way to write `sumTwoDice` is:

```
sumTwoDice :: Random Int
sumTwoDice = do
               die1 <- rollDie
               die2 <- rollDie
               return (die1 + die2)
```

It looks like we are doing assignments but don't be fooled, we are really doing binds.

Complete the following assignment to be submitted for grading. It should be done individually but you can consult with a classmate to discuss your strategies or if you get an error message that you do not understand.

Complete the following assignment in the file `Dice.hs`.

Assignment 1:
Write a function `rollNDice :: Int -> Random [Int]` which rolls dice n times and returns a list of n results. Do this with `return` and `>>=`.

Criteria for Success: Test your function using the `apply` function and an arbitrary seed:

```
> apply (rollNDice 3) 123456789
```

The result should be a list of three dice rolls. Test this with different n and seed values.

Submit your file `Dice.hs` in Canvas for grading.