Activity 12

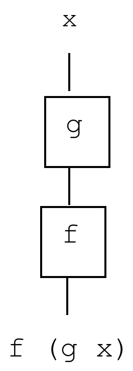
The topic of monads may seem scary but according to this blog the idea of Haskell monads is not that strange. Let's work through some of the ideas presented.

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 input the function g gets input x, the output of g is then the input of the function f:



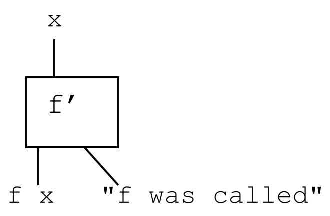
Try this out with the functions f and g that I have provided in the module Example12.
Use example of composition:

```
> f (g 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



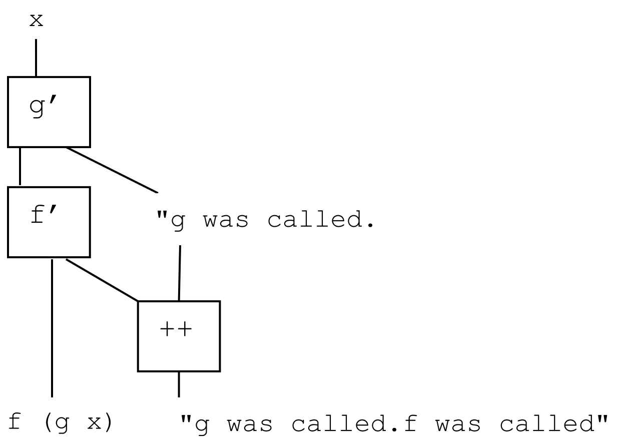
We can think of these as 'debuggable' functions.

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?

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 diagramatically:



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

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 from g' can't simply be plugged into the input to 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)
```

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))

> 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!

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,fx) = 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''
```

Let's look at a more complex example.

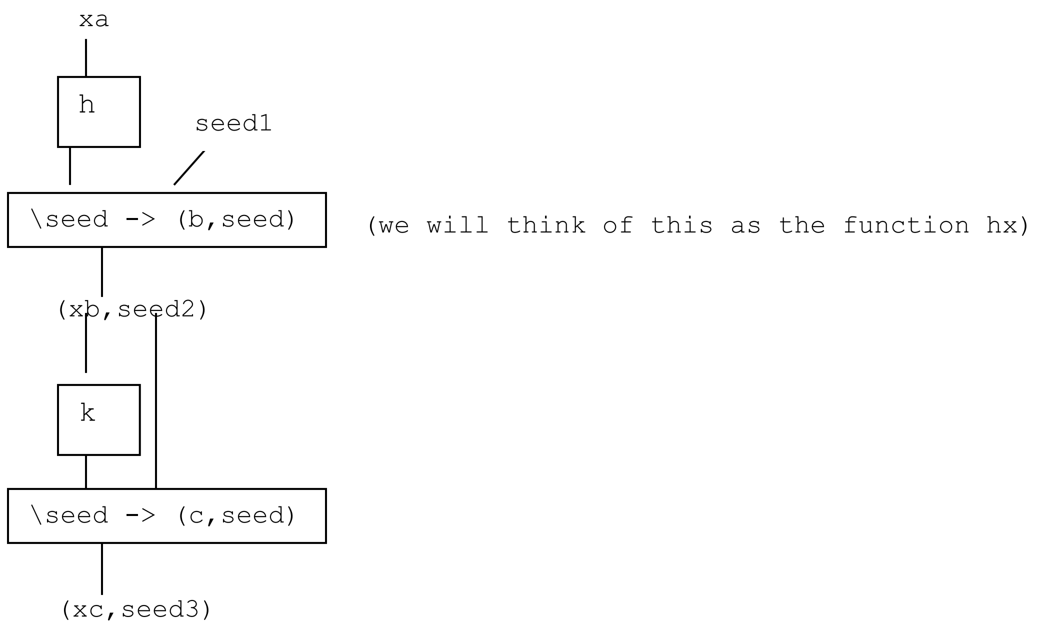
To generate a random value of type a, we would need a function that looks like:

```
type Seed = Int
random :: Seed → (a,Seed)
```

The idea is that in order to generate a random number you need a seed which gets scrambled in some way, and after you've generated the number you need to update the seed to a new value. The seed needs to be passed in and out explicitly - and that's what the signature of random describes. Note that this is similar to the debugging case above because we are returning extra data by using a pair. But this time we're passing in extra data too.

Now suppose we have a function which takes an input of type a and returns a random value of type b. The signature of such a function h would be h :: a -> Seed -> (b,Seed).

Let's look at the diagram for composing two such functions, h and k. What you have to remember is that h x is going to return a random value which is a function of type Seed -> (b,Seed).



So let's write the plumbing for the diagram:

```
bind :: (b -> Seed -> (c,Seed)) -> (Seed -> (b,Seed)) -> (Seed -> (c,Seed))
```

Which part of the signature of bind is the first parameter which will be the function k? ______________

Which part of the signature is the result of evaluating (h x)? Remember that when we evaluate (h x) this will result in returning a function (which we will call hx) ____________________

Which part of the signature is the third parameter for the original seed1? ___________________

Which part of the signature is the result should (xc,seed3)? _____________________

Ready to look at the definition of bind? Remember that bind provides the plumbing to attach the output of h, which is the function hx, and the function k. As you examine the definition, match up each piece with the diagram.

```
bind k hx = \seed1 -> let (xb,seed2) = hx seed1 in k xb seed2
```

Now we need the unit function so we can lift up the result of any function into one that can randomize.

```
unit :: a -> (Seed → (a,Seed))
unit x = \seed -> (x,seed)
```

Test out this second monad with bind' and unit' defined in Example12. You don't need to use the same seed values as used below:

```
>h' 2 123456789
> k' [2] 1234567789
> bind' k' (h' 2) 123456789
> bind' h' (unit' (m' 2)) 123456789
```

Now we will put this example in Haskell Monad syntax :

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

instance Monad Random where
    return x = MakeRandom(\seed -> (x,seed))
    MakeRandom(g) >>= f = MakeRandom(\seed-> let (xb seed') = g seed
                                                 MakeRandom(f') = f xb
                                             in f' seed')
```

In order to try this out we need a way to evaluate the function of type Seed -> (a,Seed) so we will define a function to do that:

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

Try this out to make sure it works with the functions h’’ and k’’ provided:

```
> apply ((h’’ 2) >>= k’’) 123456789
```

Also lets try lifting m’ by using return:

```
> apply (return (m’ 2) >>= h’’) 123456789
```

It's now time to step back and discern the general structure and purpose of monads.

Use the variable m to represent Debuggable or Random. In each case we are faced with the same problem. We're given a function `a -> m b` but we need to somehow apply this function to an object of type `m a` instead of one of type `a`. In each case we do so by defining a function called bind of type `(a -> m b) -> (m a -> m b)` and introducing a kind of identity function `unit :: a -> m a`.