Monads

Before we get to Monads, first a brief word about *type classes*. We saw in the example of the data type `Set a` that when we implemented this type with binary search trees we had to be sure that the general type `a` allowed the ordering operations (greater and less tests). To do this we specified `Ord a => Set a`. A type class, like `Ord` is just a collection of types that satisfy certain operations, in this case "$<$", "$=$", and "$>$".

`Monad` is a type class where all types in the class must implement the operations `return` and `>>=` (called the bind operator). What are these operations? Well, they can do different things for different monadic types, although they must satisfy some rules. It is easier to explain them by giving some examples.

Let us start with the monadic type `IO a`. An expression of type `IO a` is used to denote an action. In a functional language like Haskell, how do we implement an operation `getChar` which returns the latest character that the user has typed, or `putChar c` which prints the character `c` on the screen? The function `getChar` can not have the type `getChar :: Char` because this says that `getChar` is a function with no arguments so must be constant. We have to somehow specify that `getChar` has a side effect of interacting with the user. Similarly with `putChar`.

So the type `IO a` comes to the rescue with:

```
getChar :: IO Char

putChar :: Char -> IO ()
```

We can now define the operation

```
return :: a -> IO a
```

The command `return 42` returns the value 42 without consuming any input.

The operation `>>=` is defined as

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

The expression `p >>= q` first does `p`, returning a value `x` of type `a`, then does `q x`, returning a value `y` of type `b`. We will demonstrate this with the function `echo` in which the user inputs a character and then this character is echoed to the screen with `putChar`:

```
echo :: IO()
echo = getChar >>= putChar
```

Notice that the result of `getChar` is sent in as the argument to `putChar` which then results in something of type `IO()`. The type `IO()` means that an IO action is being perform that does not generate a value.

Now we can use these operations to write a function `readn` which reads in n characters typed by the user:

```
readn :: Int -> IO String
readn n = if n==0
           then return []
           else getChar >>= q where
                q c = readn (n-1) >>= r where
                      r cs = return (c : cs)
```

To read `n` characters we read a single character, which is passed into the function `q`. The function `q` reads `n-1` characters and passes the results to function `r`, which returns the entire string. This notation is rather clumsy so there is a simpler notation provided in Haskell. The `do` notation cleans up the code, but be aware this is just syntactic sugar for the code above.

```
readn :: Int -> IO String
readn n = if n==0
           then return []
           else do
                c <- getChar
                cs <- readn (n-1)
                return (c : cs)
```

The operations of `return` and `>>=` are used elegantly to deal with the side effects of IO.

Now let us turn to another example where monads are useful. We wish to generate random numbers (or really pseudo-random numbers) by starting with a seed value that gets scrambled in some way to produce another number. This process can then be repeated to generate the next random number. Here is one way to "scramble" the seed.

```
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
```

Now we can implement the roll of a single die:

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

We have to return both the result of the roll and the new seed to be used in the next roll. This is called a *state transformer* since it takes an initial state (the seed) and transforms it to a new state along with a result.

What if we want to sum the pips on a roll of two dice:

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

Notice that we had to thread the state (the seed) through the two rolls. The initial seed is used for the first roll, whose resulting seed in used for the second roll, whose resulting seed is returned with the sum. Instead of doing this, we could use `>>=` to thread the state for us.

We can define a type `Random a` which can be viewed as a value of type `a` which varies randomly. We can make this a monadic type as follows:

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

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

instance Monad Random where
--return :: a -> Random a
return x = MakeRandom(\seed -> (x,seed))

--(>>=) :: Random a -> (a -> Random b) -> Random b
m >>= g = MakeRandom(\seed0 ->
     let
          (result1,seed1) = apply m seed0
          (result2, seed2) = apply (g result1) seed1
     in (result2,seed2))
```

We have simply generalized the work that we did in `sumTwoDice` to define the bind operator and made the new type `Random a` a monadic type. Now we can redefine `sumTwoDice` to make use of the `return` and `>>=` operators.

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

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

This will be much easier to read if we use the `do` notation:

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

We can now test this out with an arbitrary seed

```
> apply sumTwoDice 123456789
(8,2053676357)
```