Chapter 18 : Monad : Applicative functors

*) Monads can be thought of as an another way of applying functions over structure, with a couple of additional features.

*) defn : class Applicative m ⇒ Monad m where

```
(>>=) :: m a → (a → m b) → m b

(>>) :: m a → m b → m b

return :: a → m a
```

*) Applicative m : Monad is stronger than Applicative & Functor, and we can derive Applicative & Functor in terms of Monad. Just as we can derive Functor in terms of Applicative.

↓

This means that we can write fmap using monadic operations.

↓

```
fmap f xs = xs >>= return . f
```

↓ example

```
>> fmap (+1) [1..3]
   [2, 3, 4]

>> [1..3] >>= return . (+1)
   [2, 3, 4]
```

|

↓

This happens to be a law. It is important to understand this chain of dependency:

Functor → Applicative → Monad.

i.e. when we implement an instance of Monad for a type, we necessarily have an Applicative and a Functor as well.

*) Core operations : three core ops, although only (>>=) needs to be defined for a minimally complete Monad instance.

(>>=) :: m a → (a → m b) → m b

(>>) :: m a → m b → m b

return :: a → m a : Same as pure. Takes a value & return it inside a structure, whether the structure is list, List or IO

↓

>> : Mr. Pointy (Some → sequencing operator)

sequences two actions while discarding any resulting value of the first action.

↓

>>= : bind operator, makes the Monad special.

(*) The novel part of Monad : Conventionally when we use monads, we use the bind fn. Sometimes directly, and sometimes indirectly via do syntax.

↓

```
<$> :: Functor f
    => (a → b) → f a → f b

<*> :: Applicative f
    => f (a → b) → f a → f b

>>= :: Monad f
    => f a → (a → f b) → f b
```

↓

The idea of mapping a fn. over a value while bypassing its surrounding structure is not unique to Monad.

↓

Can be demonstrated by fmapping a fn. of type $(a \to m\ b)$, to make it more like >>=.

↓

```
fmap :: Functor f
     => (a → f b) → f a → f (f b)
```

↓ example

```
>> let andOne x = [x, 1]
>> andOne 10
[10, 1]

>> :t fmap andOne [4, 5, 6]
fmap andOne [4, 5, 6] :: Num t => [[t]]

>> fmap andOne [4, 5, 6]
[[4, 1], [5, 1], [6, 1]]
```

↓

was clear from the type signature that we would reach

here, i.e. is an extra layer of structure. Now what if want to remove the extra layer of structure or to discard it in favour of one layer of that structure.

↓

can be done for lists using concat.

↓

```
concat $ fmap and one [4, 5, 6]
[4, 1, 5, 1, 6, 1]
```

↓

a less general type on concat:

```
concat :: [[a]] → [a]
```

↓

Monad in a sense is a generalisation of concat. The unique part of Monad is the following func.

```
import Control.Monad (join)

join :: Monad m ⇒ m (ma) → ma
```

↓

we can inject more structure with the std. fmap if needed, however the ability to flatten those two layers of structure into one, makes Monad special. And it's by putting join & mapping fn. together that we get the bind, also known as >>=

→ Exercise: bind in terms of fmap & join

```
bind :: Monad m => (a -> mb) -> ma -> mb
bind g x = join $ fmap g x

fmap :: (a -> b) -> fa -> fb.
        (a -> fb) -> fa -> f (fb)

join :: f (fb) -> f b
```

---

*) What monad is not ?

a) Impure. Monadic fns. are pure fns. IO is an abstract datatype, that allows for impure, or effectful, actions and it has a monad instance. But there's nothing impure about Monads.

b) An embedded lang. for imperative programming

↓

while monads are used for sequencing actions in a way that looks like imperative programming, there are also commutative monads that donot order actions.

c) A value. The typeclass describes a specific relationship between elements in a domain and define operations over them, same way we refer to monoids, functors or applicatives for that manner.

d) About strictness: Monadic ops of bind and return

are nonstrict. Some ops can be made strict within a specific instance.

*) Monad also lifts : includes some lift fns that are same as the ones we already seen in Applicative. Only present for compatability reasons, and were used before Applicatives were discovered.

↓

```
liftA :: Applicative f
      => (a -> b) -> f a -> f b

liftM :: Monad m
      => (a1 -> r) -> m a1 -> m r
```

↓

similarly for liftA2 & liftM2, same for liftA3 & liftM3.

↓

zipWith is liftA2 or liftM2 specialized to lists.

↓

```
>> :t zipWith
zipWith :: (a -> b -> c)
        -> [a] -> [b] -> [c]

>> zipWith (+) [3,4] [5,6]
[8,10]

>> liftA2 (+) [3,4] [5,6]
[8,9,9,10]
```

Different results, although same types, is because of the different monoids being used.

→ Do Syntax and monads: do syntax as introduced in modules chapter was being used in the context of IO as syntatic sugar that allowed us to easily sequence actions by feeding the result of one action as the i/p value to the next.

↓

```
(*>) :: Applicative f => fa -> fb -> fb

(>>) :: monad m => ma -> mb -> mb
```

(should in all cases do the same thing, just the underlying constraints are different)

↓

```
>> putStrLn "Hello, " >> putStrLn "World!"
Hello,
World!
```

↓ do disugared using (>>) & (*>)

```
sequencing :: IO ()
sequencing = do
  putStrLn "blah"
  putStrLn "another-thing"
```

```
sequencing' :: IO ()
sequencing' =
  putStrLn "blah" >>
  putStrLn "ano"
```

↓ variable binding ex

```
binding :: IO ()
binding = do
  name <- getLine
  putStrLn name
```

```
binding' :: IO ()
binding' = do
  getLine >>= putStrLn
```

(>>=) passes it directly to putStrLn

*) when fmap alone isn't enough : fmap putStrLn over getLine won't do anything.

putStrLn <$> getLine