1 Monad

Monads are applicative functors. A functor maps a function over a structure. An applicative maps a function contained in a structure over some other structures and then combine two layers like mappend. (functor →) So **monad** is a way of applying functions over structure.

```haskell
class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
```

`(>>=)`: bind
`(>>)`: Mr.Pointy: sequencing operator.
A monad is not

- Impure.

  `IO` is an abstract datatype that allows impure, effectful, actions and it has `Monad` instance.

- Imperative programming language. While used for *sequencing actions* that looks like imperative ones, there are commutative monads that do not order actions. (`Reader`)

- About strictness.

  The monadic operations of `bind` and `return` are nonstrict. Some operations can be made strict within a specific instance.

Not require math, category theory.

Do syntax and Monad

We can evaluate `IO` actions multiple times.

```haskell
twoBinds :: IO ()
twoBinds = do
  putStrLn "name pls:"
  name <- getLine
  putStrLn "age pls:"
  age <- getLine
  putStrLn ("y helo thar: "
            ++ name ++ " who is: "
            ++ age ++ " years old.")

twoBinds' :: IO ()
twoBinds' =
  putStrLn "name pls:" >>
  getLine >>=
  \name ->
  putStrLn "age pls:" >>
  getLine >>=
  \age ->
  putStrLn ("y helo thar: "
            ++ name ++ " who is: "
            ++ age ++ " years old.")
```

Maybe Monad

```haskell
mkSphericalCow' :: String
                -> Int
                -> Int
                -> Maybe Cow
mkSphericalCow' name' age' weight' = do
  nammy <- noEmpty name'
  agey <- noNegative age'
  weighty <- noNegative weight'
  weightCheck (Cow nammy agey weighty)
```

Why can't we do this with `Applicative`? Because our `weightCheck` function depends on the prior existence of a `Cow` value and returns more monadic structure in its return type `Maybe Cow`.

- With the `Maybe Applicative`, each `Maybe` computation fails or succeeds independently of each other. We are lifting functions that are also `Just` or `Nothing` over `Maybe` values.
- With the `Maybe Monad`, computations contributing to the final result can choose to return `Nothing` based on previous computations.

When it fails: `Nothing >>= _ = Nothing`, the `bind` function will leave the entire rest of the computation produce a `Nothing` value.

```
Prelude> Nothing >>= undefined
Nothing
Prelude> Just 1 >>= undefined
*** Exception: Prelude.undefined
```

Either Monad

```
-- m ~ Either e
(>>=) :: Monad m
      =>        m a
      -> (a ->        m b)
      ->        m b
(>>=) :: Either e a
      -> (a -> Either e b)
      -> Either e b
```

The problem is that we can't make a `Monad` instance for `Validation` that accumulates the errors like the Applicative does. Instead, it would be identical to `Either`'s one.

2 Monad laws

- **Identity laws.** `return` should be neutral and not perform any computation.

```
-- right identity
m >>= return     =  m

-- left identity
return x >>= f   =  f x
```

- **Associativity.**

```
(m >>= f) >>= g  =  m >>= (\x -> f x >>= g)
```

Regrouping the functions should not have any impacts on the final result, same as the associativity of `Monoid`.

3 Definition

Definition : Monad

Functorially applying a function which produces more structure and using `join` to reduce the nested structure.

A *monadic function* is one which generates more structure after having been lifted over monadic structure.

```
(.)
  :: (b ->   c) -> (a ->   b) -> a ->   c

(>=>)
  :: Monad m
  => (a -> m b) -> (b -> m c) -> a -> m c
```

bind is `>>=`