Class 10: Monads!

April 2

Q: is there a simple way to remove "IO" from the output?

questions from homework

Q: how does IO actually work?

questions from homework

```
data IO a = IO (RealWorld -> (RealWorld, a))

(>>=) :: IO a -> (a -> IO b) -> IO b
(IO f) >>= k = IO (\state ->
  let (state', a) = f state
    in let IO g = k a
      in g state'
```

this is a rough approximation!

questions from the homework

monads by example

Maybe

```
lookup :: a -> [(a, b)] -> Maybe b

directory :: [(String, String)]
directory =
  [("Jill", "jill3@upenn.edu"),
   ("Jack", "jack4@other.edu"),
   (“Bob" , "malformed-email")]
```

example: email directory

```
getUsername :: String -> Maybe String
getUsername name =
  case lookup name directory of
    Nothing -> Nothing
    Just email ->
      case elemIndex '@' email of
        Nothing -> Nothing
        Just i -> Just (take i email)
```

example: email directory

```
getUsername name =
  case lookup name directory of
    Nothing -> Nothing
    Just email ->
      case elemIndex '@' email of
        Nothing -> Nothing
        Just i -> Just (take i email)
```

example: on further examination

```
getUsername name =
  case lookup name directory of   Maybe String
    Nothing -> Nothing
    Just email ->
      case elemIndex '@' email of
        Nothing -> Nothing
        Just i -> Just (take i email)
```

example: on further examination

```
getUsername name =
  case lookup name directory of   Maybe String
    Nothing -> Nothing
    Just email ->                 String ->
      case elemIndex '@' email of
        Nothing -> Nothing
        Just i -> Just (take i email)
```

example: on further examination

```
getUsername name =
  case lookup name directory of   Maybe String
    Nothing -> Nothing
    Just email ->                 String ->
      case elemIndex '@' email of          Maybe Int
        Nothing -> Nothing
        Just i -> Just (take i email)
```

example: on further examination

```
getUsername name =
  case lookup name directory of
    Nothing -> Nothing
    Just email ->
      case elemIndex '@' email of
        Nothing -> Nothing
        Just i -> Just (take i email)
```

```
                              String ->
```

example: on further examination

```
getUsername name =
  case lookup name directory of
    Nothing -> Nothing
    Just email ->
      case elemIndex '@' email of
        Nothing -> Nothing
        Just i -> Just (take i email)
```

```
String -> Maybe String
```

example: on further examination

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

(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
return :: a -> Maybe a
```

suspiciously similar types

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

the Monad type class

```
instance Monad Maybe where
  (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Nothing >>= _ = Nothing
  Just x >>= k = k x

  return :: a -> Maybe a
  return x = Just x
```

a Monad instance for Maybe

please follow along locally!

```
getUsername :: String -> Maybe String
getUsername name =
  lookup name directory >>=
    (\email ->
      elemIndex '@' email >>=
        (\i ->
          return (take i email)))
```

please follow along locally!

```
getUsername :: String -> Maybe String
getUsername name = do
  email <- lookup name directory
  i <- elemIndex '@' email
  return (take i email)
```

example: revisited

(exercise: safeThird)

List

```
cross [1, 2] [3] =
  [(1, 3), (2, 3)]
```

example: cartesian product

```
cross :: [a] -> [b] -> [(a, b)]
cross xs ys =
```

example: cartesian product

```
cross :: [a] -> [b] -> [(a, b)]
cross xs ys =
  ?map
    (\x ->



    )
  xs
```

example: cartesian product

```
cross :: [a] -> [b] -> [(a, b)]
cross xs ys =
  ?map
    (\x ->
      ?map
        (\y ->  )
        ys
    )
  xs
```

example: cartesian product

```
cross :: [a] -> [b] -> [(a, b)]
cross xs ys =
  concatMap
    (\x ->
      ?map
        (\y ->  )
        ys
    )
  xs
```

example: cartesian product

```
cross :: [a] -> [b] -> [(a, b)]
cross xs ys =
  concatMap
    (\x ->
      concatMap
        (\y -> [(x, y)])
        ys
    )
  xs
```

example: cartesian product

```
instance Monad [] where
  (>>=) :: [a] -> (a -> [b]) -> [b]
  xs >>= k = concatMap k xs

  return :: a -> [a]
  return x = [x]
```

a Monad instance for List

please follow along locally!

```
cross :: [a] -> [b] -> [(a, b)]
cross xs ys =
  xs >>=
    (\x ->
      ys >>=
        (\y -> return (x, y)))
```

example: revisited

please follow along locally!

```haskell
cross :: [a] -> [b] -> [(a, b)]
cross xs ys = do
  x <- xs
  y <- ys
  return (x, y)
```

example: revisited

```
move :: Position -> [Position]

move3 :: Position -> [Position]
move3 start = return state
  >>= move
  >>= move
  >>= move
```

another example

```haskell
evensUpTo20 :: [Int]
evensUpTo20 = do
  n <- [1 .. 20]
  guard (even n)
  return n
```

guard

```
evensUpTo20 :: [Int]
evensUpTo20 = do
  n <- [1 .. 20]
  guard (even n)
  return n

guard :: Bool -> [()]
guard True = [()]
guard False = []
```

guard

```haskell
cross :: [a] -> [b] -> [(a, b)]
cross xs ys = [(x, y) | x <- xs, y <- ys]

evensUpTo20 :: [Int]
evensUpTo20 = [n | n <- [1 .. 20], even n]
```

list comprehension

(exercise: factors)

```
Monad m =>
```

```
(>>) :: Monad m => m a -> m b -> m b
ma >> mb =
```

```
(>>) :: Monad m => m a -> m b -> m b
ma >> mb = mb
```

this is incorrect!

```
(>>) :: Monad m => m a -> m b -> m b
ma >> mb = ma >>= (\_ -> mb)
```

```
(>>) :: Monad m => m a -> m b -> m b
ma >> mb = ma >>= (\_ -> mb)
```

```
Just 3 >> Just 4 = ?
```

```
(>>) :: Monad m => m a -> m b -> m b
ma >> mb = ma >>= (\_ -> mb)
```

```
Nothing >> Just 4 = ?
```

```haskell
sequence :: Monad m => [m a] -> m [a]
sequence [] = return []
sequence (ma : mas) = do
  a <- ma
  as <- sequence mas
  return (a : as)
```

```
sequence :: Monad m => [m a] -> m [a]
sequence [] = return []
sequence (ma : mas) = do
  a <- ma
  as <- sequence mas
  return (a : as)
```

```
sequence [Just 3, Just 4] = ?
```

```
sequence :: Monad m => [m a] -> m [a]
sequence [] = return []
sequence (ma : mas) = do
  a <- ma
  as <- sequence mas
  return (a : as)
```

```
sequence [Just 3, Nothing] = ?
```

(exercise: mapM)