

# COMP90048: Workshop 9

Eleanor McMurtry, University of Melbourne



Monads for great good

# Setting the stage

- Consider the function  $\text{head} :: [a] \rightarrow a$ . What if you give it an empty list?

# Setting the stage

- Consider the function `head :: [a] -> a`. What if you give it an empty list?

```
Prelude> head []
```

```
*** Exception: Prelude.head: empty list
```

# Setting the stage

- **Better:**

```
maybeHead :: [a] -> Maybe a
```

```
maybeHead [] = Nothing
```

```
maybeHead (x:_) = Just x
```

# Setting the stage

- **Example:** you are parsing a string and if it's a non-negative number, find its square root.

```
maybeRead :: Read a => String -> Maybe a
```

```
maybeSqrt :: Floating a => a -> Maybe a
```

```
maybeSqrt x
```

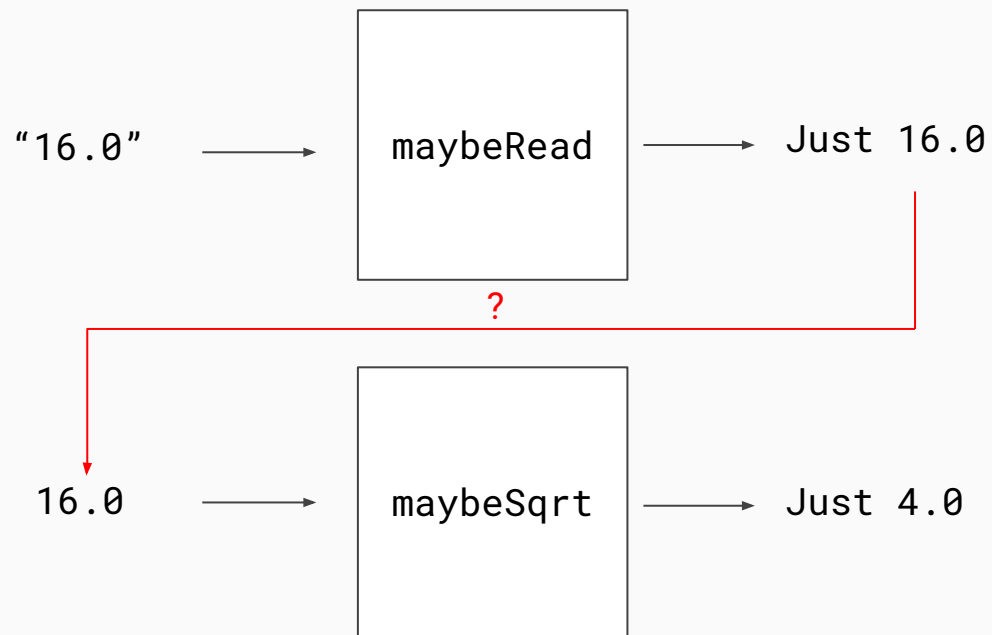
```
  | x >= 0      = Just (sqrt x)
```

```
  | otherwise = Nothing
```

# Setting the stage

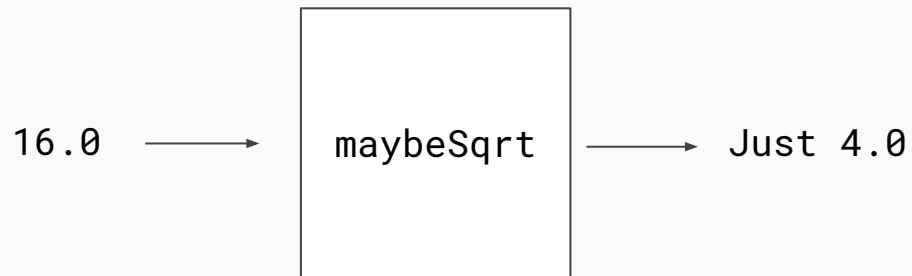
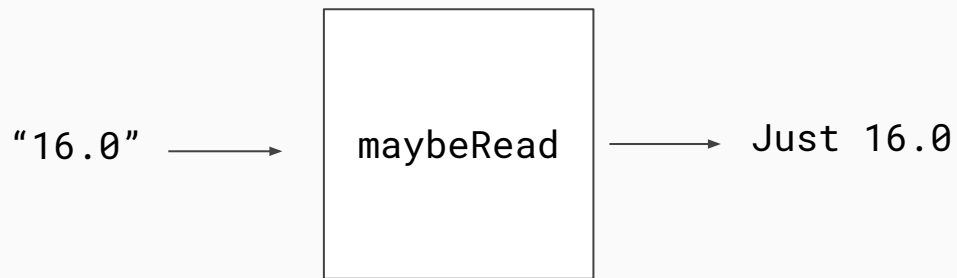


# Setting the stage



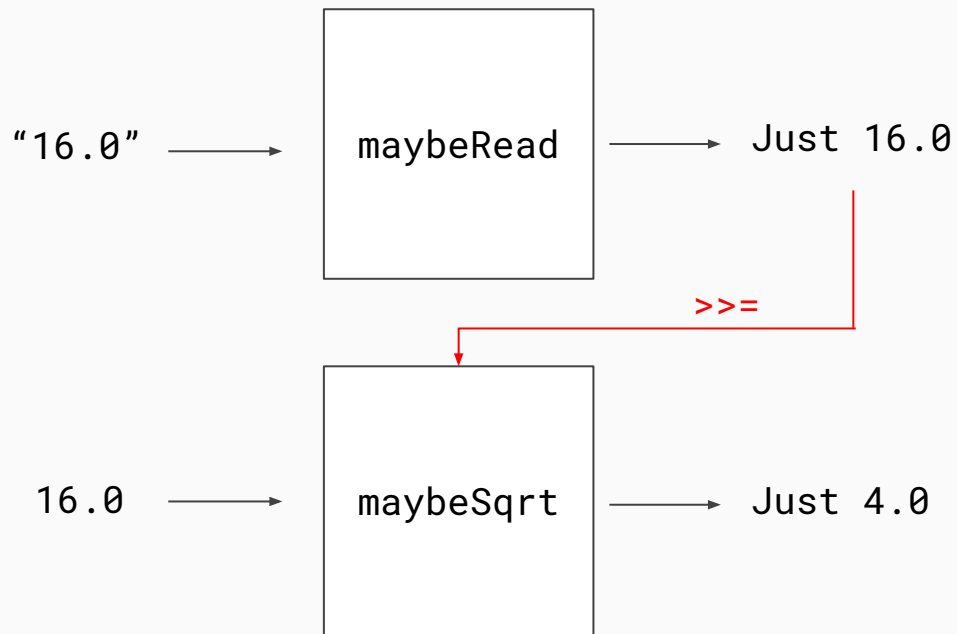


# Setting the stage



```
readAndSqrt :: String -> Maybe Double
readAndSqrt str = case maybeRead str of
  Nothing -> Nothing
  Just x   -> maybeSqrt x
```

# Introducing monadic bind!



```
readAndSqrt :: String -> Maybe Double  
readAndSqrt str =  
    maybeRead str >>= maybeSqrt
```

# Introducing monadic bind!

- The **bind** operation combines functions that operate on **monads**.

$(\gg=) :: \text{Monad } m \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

# Introducing monadic bind!

- The **bind** operation combines functions that operate on **monads**.

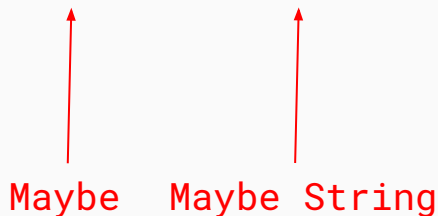
$(\gg=) :: \text{Monad } m \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

  
Maybe

# Introducing monadic bind!

- The **bind** operation combines functions that operate on **monads**.


$(\gg=) :: \text{Monad } m \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

  
Maybe    Maybe String

# Introducing monadic bind!

- The **bind** operation combines functions that operate on **monads**.

$(>>=) :: \text{Monad } m \Rightarrow m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b$

  
Maybe      Maybe String      maybeSqrt

# Introducing monadic bind!

- The **bind** operation combines functions that operate on **monads**.


$(>>=) :: \text{Monad } m \Rightarrow m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

  
Maybe    Maybe String    maybeSqrt    Maybe Double

# Introducing monadic bind!

- The **bind** operation combines functions that operate on **monads**.

$(>>=) :: \text{Monad } m \Rightarrow m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b$

  
Maybe    Maybe String    maybeSqrt



# What is a monad?

- A monad should be thought of as “a type with a context”.
  - `Maybe a` is the type `a` with the context “may be missing”

# What is a monad?

- A monad should be thought of as “a type with a context”.
  - `Maybe a` is the type `a` with the context “may be missing”
  - `[a]` is the type `a` with the context “additional data may follow”

# What is a monad?

- Monads define two basic operations:
  - $(\gg=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$   
(monadic bind)
  - $return :: a \rightarrow m\ a$   
("lift" a value into the monad)

# Monads encapsulate side effects

- Haskell normally prevents side effects. Monads allow side effects *in strict contexts*.


```
echo :: IO ()  
echo = getLine >>=  
      putStrLn
```

# Monads encapsulate side effects

- Haskell normally prevents side effects. Monads allow side effects *in strict contexts*.

```
echo :: IO ()  
echo = getLine >>=  
        putStrLn
```

`getLine :: IO String`



# Monads encapsulate side effects

- Haskell normally prevents side effects. Monads allow side effects *in strict contexts*.

```
echo :: IO ()  
echo = getLine >>=  
      putStrLn
```

*Annotations:*

- `getLine :: IO String` (points to `getLine`)
- `putStrLn :: String -> IO ()` (points to `putStrLn`)

# More side effects

```
greeter :: IO ()
greeter = putStr "First name: " >>
  getLine >>= \first ->
  putStr "Last name: " >>
  getLine >>= \last ->
  putStrLn ("Hello, " ++ first
    ++ " " ++ last ++ "!")
```

# More side effects

```
greeter :: IO ()
greeter = putStr "First name: " >>
  getLine >>= \first ->
  putStr "Last name: " >>
  getLine >>= \last ->
  putStrLn ("Hello, " ++ first
            ++ " " ++ last ++ "!")
```

ignore return value





# More side effects

```
greeter :: IO ()
```

```
greeter = putStr "First name: " >>
```

```
  getLine >>= \first ->
```

```
  putStr "Last name: " >>
```

```
  getLine >>= \last ->
```

```
  putStrLn ("Hello, " ++ first  
            ++ " " ++ last ++ "!" )
```

ignore return value

we'll use the return  
value later

# do notation

```
greeter :: IO ()
greeter = do
    putStr "First name: "
    first <- getLine
    putStr "Last name: "
    last  <- getLine
    putStrLn $ "Hello, " ++ first ++ " " ++ last ++ "!"
```

Practice, practice, practice!