Making Sense of Monads

Monday Morning Haskell - Monads Course

Lecture 1

Introduction

Course Materials

- Video Lectures
  - Explain basic concepts, walk through syntax

Course Materials

- Video Lectures
  - Explain basic concepts, walk through syntax
- Exercises
  - Practice your knowledge, pass unit tests
  - **(See PDF below this video)**

Course Materials

- Video Lectures
  - Explain basic concepts, walk through syntax
- Exercises
  - Practice your knowledge, pass unit tests
  - **(See PDF below this video)**
- Screencasts
  - Live demo of material from lecture

Course Structure

- Simpler Functional Structures
  - Monoids, Semigroups, Functors, Applicatives

Course Structure

- Simpler Functional Structures
  - Monoids, Semigroups, Functors, Applicatives
- Basic Monads

Course Structure

- Simpler Functional Structures
  - Monoids, Semigroups, Functors, Applicatives
- Basic Monads
- Reader, Writer, State Monads

Course Structure

- Simpler Functional Structures
    - Monoids, Semigroups, Functors, Applicatives
- Basic Monads
- Reader, Writer, State Monads
- IO Monad
    - Terminal, file system interactions

Course Structure

- Simpler Functional Structures
  - Monoids, Semigroups, Functors, Applicatives
- Basic Monads
- Reader, Writer, State Monads
- IO Monad
  - Terminal, file system interactions
- Functional Structure Laws

Course Structure

- Simpler Functional Structures
    - Monoids, Semigroups, Functors, Applicatives
- Basic Monads
- Reader, Writer, State Monads
- IO Monad
    - Terminal, file system interactions
- Functional Structure Laws
- Final Challenge (Parsing)

Lecture 2

Monoids and Semigroups

Intro to Monoids and Semigroups

- Every Monoid is a Semigroup
- Simpler than Monads
- A Semigroup is a type that can **build on itself**
- A Monoid also has an **Identity Element**

The Semigroup Typeclass

```haskell
class Semigroup a where
  -- AKA "mappend"
  (<>) :: a -> a -> a
```

Integer Addition

```
(+) :: Int -> Int -> Int

instance Semigroup Int where
  a <> b = a + b
```

Integer Multiplication

```
instance Semigroup Int where
  (<>) = (*)
```

The Monoid Typeclass

```
class Semigroup a => Monoid a where
  mempty :: a

a <> mempty == a
mempty <> a == a
```

Integer Addition

```
instance Semigroup Int where
  a <> b = a + b

instance Monoid Int where
  mempty = 0
```

Integer Multiplication

```
instance Semigroup Int where
  a <> b = a * b

instance Monoid Int where
  mempty = 1
```

List Instance

```
instance Semigroup [a] where
  (<>) = (++)

instance Monoid [a] where
  mempty = []
```

Conclusion

- Why do these abstractions help?
  - Help us to write **polymorphic** code

Lecture 3

Functors

Review

- **Monoids** and **Semigroups**
  - Type that can "build" on itself by **appending**
- **Functors** - first step towards monads!

Defining Functors

- A **container** of elements

Defining Functors

- A container of elements
- Can apply a **transformation** on those elements
- Transformation **preserves the internal structure**

Functor Typeclass

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Functor Typeclass

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Functor Typeclass

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Functor Typeclass

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Functor Typeclass

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$>) = fmap
```

Similarities to Map

```
fmap :: (a -> b) -> f a -> f b

map :: (a -> b) -> [a] -> [b]
```

Similarities to `map`

```
fmap :: (a -> b) -> f a -> f b

map :: (a -> b) -> [a] -> [b]

instance Functor [] where
  fmap = map
```

Maybe Instance

```
data Maybe a = Nothing | Just a
```

Maybe Instance

```
data Maybe a = Nothing | Just a

instance Functor Maybe where
  fmap _ Nothing = Nothing
  ...
```

Maybe Instance

```
data Maybe a = Nothing | Just a

instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just a) = Just (f a)
```

The Either Type

```
data Either a b = Left a | Right b
```

The Either Type

```
data Either a b = Left a | Right b

instance Functor Either where
  fmap _ (Left a)  = Left a
  fmap f (Right b) = Right (f b)
```

Conclusion

- A container of elements
- Can apply a **transformation** on those elements
- Transformation **preserves the internal structure**

Lecture 4

Applicative Functors

Functor Review

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Applicative Functors

```haskell
-- Allow application of function with a structure
class (Functor f) => Applicative f where
  ...
```

Applicative Functors

```
class (Functor f) => Applicative f where
  -- fmap :: (a -> b) -> f a -> f b
  (<*>) :: f (a -> b) -> f a -> f b
```

Applicative Functors

```
class (Functor f) => Applicative f where
  (<*>) :: f (a -> b) -> f a -> f b
  pure :: a -> f a
```

Maybe Instance

```
instance Applicative Maybe where
  pure = Just
  ...
```

Maybe Instance

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  _ <*> Nothing = Nothing
  ...
```

Maybe Instance

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  _ <*> Nothing = Nothing
  Just f <*> Just a = Just (f a)
```

Maybe Instance

```
>> let a = Just 5
>> let b = Just 7
>> pure (+) <*> a <*> b
Just 12
```

Maybe Instance

```
>> let a = Just 5
>> let b = Just 7
>> pure (+) <*> a <*> b
Just 12
>> pure (+) <*> Nothing <*> b
Nothing
>> pure (+) <*> a <*> Nothing
Nothing
```

Using `fmap`

```
>> let a = Just 5
>> let b = Just 7
>> (+) <$> a <*> b
Just 12
```

List Applicative

```
>> let a = [1,2]
>> let b = [3,4]
>> pure (+) <*> a <*> b
???
```

List Applicative

```
>> let a = [1,2]
>> let b = [3,4]
>> pure (+) <*> a <*> b
[4,6]?
```

List Applicative

```
>> let a = [1,2]
>> let b = [3,4]
>> pure (+) <*> a <*> b
[4,5,5,6]
```

List Applicative

```
instance Applicative [] where
  pure a = [a]
  fs <*> as = [f a | f <- fs, a <- as]
```

List Applicative

```
instance Applicative [] where
  pure a = [a]
  fs <*> as = [f a | f <- fs, a <- as]
```

- Imagine list as a non-deterministic context
- We want every combination of options!

List Applicative

```
>> pure (+) <*> [1, 2] <*> [3, 4]
>> [1+, 2+] <*> [3, 4]
>> [1+3, 1+4, 2+3, 2+4]
>> [4, 5, 5, 6]
```

ZipList

```
>> let a = ZipList [1,2]
>> let b = ZipList [3,4]
>> pure (+) <*> a <*> b
ZipList {getZipList = [4,6]}
```

ZipList

```
>> let a = ZipList [1,2]
>> let b = ZipList [3,4,8,9]
>> pure (+) <*> a <*> b
ZipList {getZipList = [4,6]}
```

Conclusion

- Exercises - Learn applicative patterns!
- Monads are next!

Lecture 5

Monad Basics

Review

- Finally ready for monads!
- Not a big, scary concept!
- Just another structure with a typeclass

Monads

- **Context** in which a computation takes place
- Specifies how to **combine operations**
- e.g. Pass information as implicit parameters
- **Side effects**

Class Definition

```
class Applicative m => Monad m where
  ...
```

Class Definition

```
class Applicative m => Monad m where
  return :: a -> m a
  ...
```

Class Definition

```
class Applicative m => Monad m where
  return :: a -> m a
  -- AKA "bind"
  (>>=) :: m a -> (a -> m b) -> m b
```

Comparing Functions

```
(<$>) ::    (a -> b) -> f a -> f b
(<*>) :: f (a -> b) -> f a -> f b
(=<<) :: (a -> f b) -> f a -> f b
```

Maybe Monad

```
-- Computation might "fail"
-- Might produce a value, or might not
instance Monad Maybe where
  ...
```

Maybe Monad

```haskell
instance Monad Maybe where
  return = Just
  Nothing  >>= _ = Nothing
  (Just a) >>= f = Just (f a)
```

Maybe Monad

```haskell
canFail1 :: a -> Maybe b
canFail2 :: b -> Maybe c
canFail3 :: c -> Maybe d

finalValue :: a -> Maybe d
finalValue item =
  (return item) >>= canFail1 >>= canFail2 >>= canFail3
```

Maybe Monad

```haskell
canFail1 :: a -> Maybe b
canFail2 :: b -> Maybe c
canFail3 :: c -> Maybe d

finalValue :: a -> Maybe d
finalValue item =
  canFail1 item >>= canFail2 >>= canFail3
```

List Monad

```
instance Monad [] where
  return a = [a]
  xs >>= f = [y | x <- xs, y <- f x]
```

List Monad

```
makeMany1 :: a -> [b]
makeMany2 :: b -> [c]
makeMany3 :: c -> [d]

finalValue :: a -> [d]
finalValue item =
  makeMany1 item >>= makeMany2 >>= makeMany3
```

List Monad

```
makeMany1 :: Int -> [Int]
makeMany1 x = [2 * x, 3 * x]

makeMany2 :: Int -> [Int]
makeMany2 y = [y + 1, y + 2]

finalValue :: Int -> [Int]
finalValue item = return item >>= makeMany1 >>= makeMany2
```

List Monad

```
>> finalValue 4
>> [4] >>= makeMany1 >>= makeMany2
>> [8, 12] >>= makeMany2
>> [9, 10, 13, 14]
```

Review

- Try out monadic ideas!
- Bind operator has limitations.
- Can improve our syntax!

Lecture 6

Do-Syntax

Bind Operator

```
canFail1 :: a -> Maybe b
canFail2 :: b -> Maybe c
canFail3 :: c -> Maybe d

finalValue :: a -> Maybe d
finalValue item = canFail1 item >>= canFail2 >>= canFail3
```

Harder Example

```
func1 :: a -> Maybe b
func2 :: a -> Maybe c
func3 :: b -> c -> Maybe d

finalValue :: a -> Maybe d
finalValue x = ???
```

Harder Example

```
func1 :: a -> Maybe b
func2 :: a -> Maybe c
func3 :: b -> c -> Maybe d

finalValue :: a -> Maybe d
finalValue x = func1 x >>=
  (\b -> func2 x >>= (\c -> return (b, c))) >>=
  (\(b, c) -> func3 b c)
```

Do Syntax

```
func1 :: a -> Maybe b
func2 :: a -> Maybe c
func3 :: b -> c -> Maybe d

finalValue :: a -> Maybe d
finalValue x = do
  y <- func1 x
  z <- func2 x
  func3 y z
```

Do Syntax

```
finalValue :: a -> Maybe d
finalValue x = do
  -- Observe the "get" operator <-
  y <- (func1 x :: Maybe b)
  z <- (func2 x :: Maybe c)
  func3 y z
```

Do Syntax

```
finalValue :: a -> Maybe d
finalValue x = do
  (y :: b) <- (func1 x :: Maybe b)
  (z :: c) <- (func2 x :: Maybe c)
  func3 y z
```

Do Syntax

```
finalValue :: a -> Maybe d
finalValue x = do
  (y :: b) <- (func1 x :: Maybe b)
  (z :: c) <- (func2 x :: Maybe c)
  (func3 y z :: Maybe d)
```

Concrete Example

```
safeSqrt :: Double -> Maybe Double
safeSqrt x = if x < 0
  then Nothing
  else Just (sqrt x)

safeDivide :: Double -> Double -> Maybe Double
safeDivide x y = if y == 0.0
  then Nothing
  else Just (x / y)
```

Concrete Example

```
divideSqrts :: Double -> Double -> Maybe Double
divideSqrts x y = do
  x' <- safeSqrt x
  y' <- safeSqrt y
  safeDivide x' y'
```

Concrete Example

```
divideSqrts :: Double -> Double -> Maybe Double
divideSqrts x y = do
  (x' :: Double) <- (safeSqrt x :: Maybe Double)
  (y' :: Double) <- (safeSqrt y :: Maybe Double)
  (safeDivide x' y' :: Maybe Double)
```

Concrete Example

```
divideSqrts :: Double -> Double -> Maybe Double
divideSqrts 16.0 4.0 = do
  x' <- safeSqrt 16.0
  y' <- safeSqrt 4.0
  safeDivide x' y'
```

Concrete Example

```
divideSqrts :: Double -> Double -> Maybe Double
divideSqrts 16.0 4.0 = do
  4.0 <- Just 4.0
  2.0 <- Just 2.0
  safeDivide 4.0 2.0
```

Concrete Example

```
divideSqrts :: Double -> Double -> Maybe Double
divideSqrts 16.0 4.0 = do
  4.0 <- Just 4.0
  2.0 <- Just 2.0
  Just 2.0 -- << Final Result!
```

Failure Example

```
divideSqrts :: Double -> Double -> Maybe Double
divideSqrts (-16.0) 4.0 = do
  x' <- safeSqrt (-16.0)
  y' <- safeSqrt 4.0
  safeDivide x' y'
```

Failure Example

```
divideSqrts :: Double -> Double -> Maybe Double
divideSqrts (-16.0) 4.0 = do
  ??? <- Nothing
  y' <- safeSqrt 4.0
  safeDivide x' y'
```

Failure Example

```
divideSqrts :: Double -> Double -> Maybe Double
divideSqrts (-16.0) 4.0 = do
  ??? <- Nothing
  Nothing
```

Let Statements

```
divideSqrts :: Double -> Double -> Maybe Double
divideSqrts x y = do
  x' <- safeSqrt x
  y' <- safeSqrt y
  let z' = y' - 2.0
  safeDivide x' z'
```

Let Statements

```
divideSqrts :: Double -> Double -> Maybe Double
divideSqrts 16.0 4.0 = do
  4.0 <- Just 4.0
  2.0 <- Just 2.0
  let z' = 0.0
  safeDivide 4.0 0.0
```

List Monad

```
makeMany1 x = [2 * x, 3 * x]

makeMany2 y = [y + 1, y + 2]

finalValue :: Int -> [Int]
finalValue x = do
   (y :: Int) <- (makeMany1 x :: [Int])
   (makeMany y :: [Int])
```

List Monad

```
makeMany1 x = [2 * x, 3 * x]

makeMany2 y = [y + 1, y + 2]

finalValue :: Int -> [Int]
finalValue 4 = do
  ??? <- [8, 12]
  [9, 10, 13, 14]
```

Conclusion

- Do-syntax makes it easy to write clean code!
- More complicated monads coming up!

Lecture 7

Reader and Writer Monads

Intro

- Explored `Maybe`, `Either`, and List as monads
- `Reader` and `Writer` have more specialized roles

Reader Monad

- Context of a global read-only value.
- Allows us to avoid external parameter passing.

Ask

```
ask :: Reader r r

...
```

Reader Monad

```
ask :: Reader r r

data Config = Config ...

readerAction :: Reader Config Int
readerAction = do
  (conf :: Config) <- ask
  ... -- Computations with conf
```

Asks

```
asks :: (r -> s) -> Reader r s

data Config = Config { configParam1 :: Int, ... }

readerAction :: Reader Config Int
readerAction = do
  (param1 :: Int) <- asks configParam1
  ... -- Computations with param1
```

Local

```
local :: (r -> r) -> Reader r s -> Reader r s

updateConfig :: Config -> Config
otherAction :: Reader Config Float

readerAction :: Reader Config Int
readerAction = do
  (result :: Float) <- local updateConfig otherAction
  ...
```

Reader and Side Effects

- Simpler implementation than Maybe, List
- No extra effects
  - (e.g. short-circuiting, multiplicity)
- Monad functions just pass the state

runReader

```
runReader :: Reader r a -> r -> a
```

runReader

```
runReader :: Reader r a -> r -> a

useConfig :: Config -> Int
useConfig config = runReader readerAction config

readerAction :: Reader Config Int
...
```

Writer Monad

- Allows writing to a global, write-only state
- Takes single type parameter
- Write state must be a **Monoid**

Tell

```
tell :: (Monoid w) => w -> Writer w ()
```

Many Parameters

```
func1 :: (Int, String) -> (Int, String)
func2 :: (Int, String) -> (Int, String)
func3 :: (Int, String) -> (Int, String)
```

Many Parameters

```
func1 :: (Int, String) -> (Int, String)
func1 (prevCost, input) =
  if length input > 5
    then func2 (prevCost + 3, drop 3 input)
    else func3 (prevCost + 5, 'a' : input)
```

Using Writers

```
instance Monoid Int where
  ...

func1 :: String -> Writer Int String
func2 :: String -> Writer Int String
func3 :: String -> Writer Int String
```

Using a Writer

```
func1 :: String -> Writer Int String
func1 input =
  if length input > 5
    then do
      tell 3
      func2 $ drop 3 input
    else do
      tell 5
      func3 ('a' : input)
```

Using a Writer

```
-- No inputs, but 2 outputs!
runWriter :: Writer w a -> (a, w)

getCostAndFinalString :: String -> (String, Int)
getCostAndFinalString input = runWriter (func1 input)
```

Log Messages

```
func1 :: Int -> Writer [String] Int
func1 input = do
  tell ["Running func1"]
  ... -- Computations with input
```

Conclusion

- Get some practice with these!
- Next up: combining the ideas with `State` monad!

Lecture 8

State Monad

Review

- Reader and Writer monads
    - Implicit read-only and write-only states
- State monad - accessible and modifiable **global state**!

State Monad

```
data MyState = ...

stateAction :: State MyState Int
```

Reading our State

```haskell
data MyState = MyState { stateParam1 :: Int, ... }

get :: State s s

stateAction :: State MyState Int
stateAction = do
  (myState :: MyState) <- get
  ...
```

Reading our State

```
data MyState = MyState { stateParam1 :: Int, ... }

get :: State s s
gets :: (s -> a) -> State s a

stateAction :: State MyState Int
stateAction = do
  (myState :: MyState) <- get
  (param1 :: Int) <- gets stateParam1
  ...
```

Modifying our State

```
put :: s -> State s ()

stateAction :: State MyState Int
stateAction = do
  initialState <- get -- Retrieves old state
  put (MyState 5 ...) -- Modifies state
  newState <- get      -- Retrieves new state
  return $ stateParam1 newState -- returns 5
```

Modifying our State

```
modify :: (s -> s) -> State s ()

updateState :: MyState -> MyState

stateAction :: State MyState Int
stateAction = do
  initialState <- get -- Retrieves old state
  modify updateState  -- Modifies state
  newState <- get     -- Retrieves new state
  ...
```

Modifying our State

```
updatingAction :: State Int Int
updatingAction = do
  oldValue <- get
  modify (+1)
  return $ oldValue + 5
```

Modifying our State

```
stateAction :: State Int Int
stateAction = do
  initialValue <- get
  result1 <- updatingAction
  newValue <- get
  ...
```

Modifying our State

```
stateAction :: State Int Int
stateAction = do
  4 <- get
  9 <- updatingAction
  5 <- get
  ...
```

Running our State

```
runReader :: Reader r a -> r -> a
runWriter :: Writer w a -> (a, w)

runState :: State s a -> s -> (a, s)
```

Running our State

```
runState  :: State s a -> s -> (a, s)
execState :: State s a -> s -> s
evalState :: State s a -> s -> a
```

Object Oriented Programming

```
class MyObject {

  private int myInt;

  public void addInt(int a) {
    self.myInt += a;
  }
}
```

Object Oriented Programming

```
data MyObject = MyObject Int

addInt :: Int -> State MyObject ()
addInt a = modify (+ a)
```

Conclusion

- Haskell seems to have restrictions
    - Immutability, lack of global state
- We can still do anything from other languages!
    - But side effects must be encoded in the type system
- Next up: `IO` Monad!

Lecture 9

The IO Monad

Review

- Basic monads and specialized, stateful monads
- These only depend on their inputs
  - (Implicit and explicit)
- Thus they are "pure."

Introduction to IO

- The IO Monad can communicate with "the outside world."
  - Terminal, File System, OS, Network
- Much more prone to **runtime errors**
- So we want to **limit** where our program can use it
  - But we'll still need it *somewhere*!

Basic IO Functions

```
putStrLn :: String -> IO ()

print :: (Show a) => a -> IO ()

putStr :: String -> IO ()

getLine :: IO String
```

Basic IO Functions

```
fetchAndPrintName :: IO ()
fetchAndPrintName = do
  putStrLn "Hello! Please enter your name."
  input <- getLine
  putStrLn $ "Hello, " ++ input ++ "!"
  putStrLn "How many characters are in your name?"
  print (length input)
```

Basic IO Functions

```
Hello! Please enter your name.
Christopher
Hello, Christopher!
How many characters are in your name?
11
```

IO as a Functor

```
fetchAndPrintName :: IO ()
fetchAndPrintName = do
  putStrLn "Hello! Please enter your name."
  capitalName <- (map toUpper) <$> getLine
  putStrLn capitalName
```

IO as a Functor

```
Hello! Please enter your name.
Christopher
CHRISTOPHER!
```

Running IO?

```
runIO :: IO a -> ??? -> a
```

Running IO?

~~`runIO :: IO a -> ??? -> a`~~

Running IO?

- The IO Monad is a *different* kind of context.
  - Its side effects are limitless
- We can't allow *any* function to call into IO.
  - It would defeat the purpose of separating it!

The `main` function

- If our code starts in a pure function, it can **never** call IO!
- So our starting point **must** be an `IO` function

The `main` function

- If our code starts in a pure function, it can **never** call IO!
- So our starting point **must** be an `IO` function
- `main :: IO ()`
  - Starting point for all our code
- Can call into pure code, or more IO code
- All IO code must form a chain back to `main`!

The `main` function

- A Haskell module with `main` can be run via GHC
- Stack organizes things through "executables"
  - Each has a designated `Main` module with `main :: IO ()`

File System Functions

```haskell
type FilePath = String

getCurrentDirectory :: IO FilePath

getHomeDirectory :: IO FilePath

(</>) :: FilePath -> FilePath -> FilePath

listDirectory :: FilePath -> IO [FilePath]
```

File System Functions

```
-- Retrieve arguments passed to program!
getArgs :: IO [String]

>> my-exec --name Christopher --password 1234

getArgs -> ["--name", "Christopher", "--password", "1234"]
```

Conclusion

- Syntactically, `IO` is just another monad!
- Next up: reading and writing files!

Lecture 10

Reading and Writing Files

Review

- `IO` Monad basics - reading and writing to terminal
- Some file system operations
- What about getting information from files?

Reading Files

```
readFile :: FilePath -> IO String

main :: IO ()
main = do
  fileContents <- readFile "myfile.txt"
  ...
```

Using Handles

```
openFile :: FilePath -> IOMode -> IO Handle

main :: IO ()
main = do
  fileHandle <- openFile "myfile.txt" ReadMode
  ...
```

Using Handles

```
hGetLine     :: Handle -> IO String
hGetContents :: Handle -> IO String

main :: IO ()
main = do
  fileHandle <- openFile "myfile.txt" ReadMode
  line1 <- hGetLine fileHandle      -- Reads first line
  line2 <- hGetLine fileHandle      -- Reads second line
  rest  <- hGetContents fileHandle -- Reads rest of file
  ...
```

Splitting up Lines

```
lines :: String -> [String]
unlines :: [String] -> String

main :: IO ()
main = do
  fileHandle <- openFile “myfile.txt” ReadMode
  fileLines :: [String] <- lines <$> hGetContents fileHandle
  ...
```

Closing Handles

```haskell
hClose :: Handle -> IO ()

main :: IO ()
main = do
  fileHandle <- openFile "myfile.txt" ReadMode
  line1 <- hGetLine fileHandle
  hClose fileHandle -- Safely closes the file
```

File Output

```
main :: IO ()
main = do
  line1 <- getLine
  putStrLn line1
```

File Output

```
main :: IO ()
main = do
  line1 <- getLine
  handle <- openFile "outputfile.txt" WriteMode
  putStrLn line1
```

File Output

```haskell
hPutStrLn :: Handle -> String -> IO ()

main :: IO ()
main = do
  line1 <- getLine
  handle <- openFile "outputfile.txt" WriteMode
  hPutStrLn handle line1
  hClose handle
```

Write Mode vs. Append

- **WriteMode** will **ERASE** anything in an existing file
  - (This happens after you write to it the first time, not opening the file)
- **AppendMode** adds content to the end of an existing file

Strictness

```haskell
-- Works lazily!
readFile :: FilePath -> IO String

-- inputfile.txt and outputfile.txt will be open
-- at the same time!
main :: IO ()
main = do
  originalFileContents <- readFile "inputfile.txt"
  handle <- openFile "outputfile.txt" WriteMode
  hPutStrLn handle originalFileContents
  hClose handle
```

Strictness

```
import qualified System.IO.Strict as S

-- inputfile.txt and outputfile.txt will be NOT open
-- at the same time!
main :: IO ()
main = do
  -- Contents are read strictly!
  originalFileContents <- S.readFile "inputfile.txt"
  handle <- openFile "outputfile.txt" WriteMode
  hPutStrLn handle originalFileContents
  hClose handle
```

Terminal Handles

```haskell
stdin  :: Handle
stdout :: Handle
stderr :: Handle

main :: IO ()
main = do
  hPutStrLn stdout "Please enter your name."
  input <- hGetLine stdin
  hPutStrLn stdout ("Hello, " ++ input ++ "!")
```

Creating a Directory

```
createDirectoryIfMissing :: Bool -> FilePath -> IO ()

main :: IO ()
main = do
  dir <- getCurrentDirectory
  createDirectoryIfMissing False (dir </> "new_directory")
```

File Existence and Manipulation

```
doesFileExist         :: FilePath -> IO Bool
doesDirectoryExist    :: FilePath -> IO Bool
getModificationTime   :: FilePath -> IO UTCTime
setModificationTime   :: FilePath -> UTCTime -> IO ()
```

Conclusion

- Now you know many different IO tasks!
- User interaction, file manipulation, etc.
- Next up: combining monads!

Lecture 11

Monad Transformers

Introduction

- Seen many monads so far!
- What if want to use 2 monads at once?
- Monad transformers let us do this!

Motivating Example

- Entering Registration Info at Terminal
- Terminal Requires IO monad
- But failed operations should short-circuit, so we also want `Maybe` monad.

Monad Transformers

```
-- "m" is always another monad
Maybe     --> MaybeT m
Reader r  --> ReaderT r m
State s   --> StateT s m

-- No Transformer for IO
```

Monad Transformers

```
maybeIOAction :: MaybeT IO Int

readerMaybeAction :: ReaderT Config Maybe Int
```

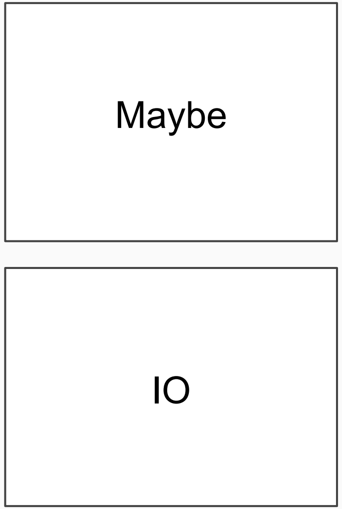
Monad Transformers

```
maybeIOAction :: MaybeT IO Int

readerMaybeAction :: ReaderT Config Maybe Int

type LongMonad a =
  StateT MyState (WriterT LogType (ReaderT Config IO)) a
```

Monad Transformer Stack



Monad Transformer Stack

Registration Example

```
readEmail :: MaybeT IO String
readPassword :: MaybeT IO String
readAge :: MaybeT IO Int

runRegistration :: MaybeT IO User
runRegistration = do
  email <- readEmail
  password <- readPassword
  age <- readAge
  return $ User email password age
```

Run Functions

```
runState  :: State  s a   -> s ->   (a, s)

runStateT :: StateT s m a -> s -> m (a, s)
```

Run Maybe

```haskell
runMaybeT :: MaybeT m a -> m (Maybe a)
```

Run Functions

```
type LongMonad a =
  StateT MyState (WriterT LogType (Reader Config)) a

runLongM :: LongMonad a -> Config -> MyState ->
  (a, MyState, LogType)
runLongM action initialConfig initialState = (res, state, log)
  where
    writerAction          = runStateT action initialState
    readerAction          = runWriterT writerAction
    ((res, state), log)   = runReader readerAction initialConfig
```

Registration Example

```
runRegistration :: MaybeT IO User
...

main :: IO
main = do
  maybeUser <- runMaybeT runRegistration
  case maybeUser of
    Nothing -> ...
    Just user -> ...
```

Lifting

```
runRegistration :: MaybeT IO User
runRegistration = do
  -- Fails!
  putStrLn "Please enter your info!"  -- < IO
  email <- readEmail                  -- < MaybeT IO
  password <- readPassword
  age <- readAge
  return $ User email password age
```

Lift

```
lift :: m a -> t m a

runRegistration :: MaybeT IO User
runRegistration = do
  -- Succeeds!
  lift $ putStrLn "Please enter your info!"
  email <- readEmail
  ...
```

Lift

Maybe	`lift getLine :: MaybeT IO String`
IO	`getLine :: IO String`

Lift!

Lift

State Int	`lift putStrLn :: State Int IO ()`	Lift!
IO	`putStrLn :: IO ()`	

Programming Patterns

- Monad Transformers are **really important**
- `StateT IO` **pattern**
- `Reader/Writer/State` **pattern**

Conclusion

- Practice Practice Practice!
- Next up: Functional Structure Laws!

Lecture 12

Functional Structure Laws

Introduction

- Many different structures
- What is a "valid" structure?
- Each structure follows mathematical **laws**
- Other programmers expect your structures to follow these!
- Abstract, but intuitive

Monoid Identity Law

```
a <> mempty = a
mempty <> a = a

instance Monoid Int where
  mempty = 0
  mappend = (+)

2 <> 0 = 2
0 <> 2 = 2
```

Monoid Identity Law

```
a <> mempty = a
mempty <> a = a

instance Monoid Int where
  mempty = 1 -- Bad idea!
  mappend = (+)

2 <> 1 = 3
1 <> 2 = 3
```

Monoid Associativity Law

```
(a <> b) <> c = a <> (b <> c)

instance Monoid Int where
  mempty = 0
  mappend = (+)

((3 + 4) + 5) + 6 = 18
(3 + 4) + (5 + 6) = 18
3 + ((4 + 5) + 6) = 18
```

Commutativity

```
instance Semigroup [a] where
  mappend = (++)

instance Monoid [a] where
  mempty = []

-- NOT Commutative!
[True] ++ [False] = [True, False]
[False] ++ [True] = [False, True]
```

Functor Laws

- Identity Law
    - `fmap id = id`

Functor Laws

- Identity Law
  - `fmap id = id`
- Composition Law
  - `(.) :: (b -> c) -> (a -> b) -> (a -> c)`

Functor Laws

- Identity Law
  - `fmap id = id`
- Composition Law
  - `(.) :: (b -> c) -> (a -> b) -> (a -> c)`
  - `fmap (f . g) = fmap f . fmap g`

Applicative Laws

- Identity Law
  - `pure id <*> v = v`

Applicative Laws

- Identity Law
  - `pure id <*> v = v`
- Composition Law
  - `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`

Applicative Laws

- Identity Law
  - `pure id <*> v = v`
- Composition Law
  - `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
- Homomorphism Law
  - `pure f <*> pure x = pure (f x)`
- Interchange Law
  - `u <*> pure y = pure ($ y) <*> u`

Monad Laws

- Left Identity Law
  - `return a >>= f = f a`
- Right Identity Law
  - `m >>= return = m`

Monad Laws

- Left Identity Law
  - `return a >>= f = f a`
- Right Identity Law
  - `m >>= return = m`
- Associativity Law
  - `(m >>= f) >>= g = m >>= (\x -> f x >>= g)`

Conclusion

- Many laws, but a few core concepts
- **Identity** functions don't change anything
- Applying functions **shouldn't change structure**
- **Order** of **application** shouldn't matter
- Intuitive functions will follow laws
- Next up: Final Challenge!

Lecture 13

Parsing with Megaparsec

Introduction

- Time for the final challenges!
- Parse files using `Megaparsec`
- Lots of things to learn about parsing
- But it's good practice for monads!

Parsing

- Translate file contents into program structure
- Parsing is inherently **stateful**
- Produce certain outputs as we consume the string
- Might have to **backtrack**
- Usually a **monadic** process

Parsec Monad

```
data Parsec e s a = ...
```

Parsec Monad

```
data Parsec e s a = ...

data ParsecT e s m a = ...
```

Running the Parser

```
runParser :: Parsec e s a -> String -> s
  -> Either (ParseErrorBundle e s) a
```

Running the Parser

```
runParser :: Parsec e s a -> String -> s
  -> Either (ParseErrorBundle e s) a

runParserT :: ParsecT e s m a -> String -> s
  -> m (Either (ParseErrorBundle e s) a)
```

Making an Alias

```
data Parsec e s a = ...

type Parser a = Parsec Void String a

type RParser a = ReaderT Config Parser a
```

Making an Alias

```
data ParsecT e s m a = ...

type ParserT a = Parsec Void String (Reader Config) a
```

Parsing Strings

```
string :: String -> Parser String

-- Case Insensitive
string' :: String -> Parser String
```

Parsing Strings

```
hello :: Parser String
hello = string' "Hello"

>> runParser hello "" "Hello"
Right "Hello"
>> runParser hello "" "hello"
Right "hello"
```

Parsing Strings

```
hello :: Parser String
hello = string' "Hello"

>> runParser hello "" "He"
Left ...
>> runParser hello "" "Hello, world!"
Left ...
```

Parsing Unknown Characters

```
letterChar :: Parser Char

>> runParser letterChar "" "H"
Right 'H'
>> runParser letterChar "" "a"
Right 'a'
>> runParser letterChar "" " "
Left ...
>> runParser letterChar "" "5"
Left ...
```

`some` and `many`

```
some :: Parser a -> Parser [a]
many :: Parser a -> Parser [a]

word :: Parser String
word = some letterChar

>> runParser word "" "Good"
Right "Good"
>> runParser word "" "Hello, world!"
Right "Hello"
```

some and many

```
>> runParser (some letterChar) "" "1 Hi"
Left ...
>> runParser (many letterChar) "" "1 Hi"
Right ""
>> runParser (some letterChar) "" ""
Left ...
>> runParser (many letterChar) "" "Hello, World"
Right "Hello"
```

Combing Parsers

```
parseFeatureName :: Parser String
parseFeatureName = do
  _ <- string "Feature: "
  word

>> runParser parseFeatureName "" "Feature: Parsing"
Right "Parsing"
>> runParser parseFeatureName "" "Parsing"
Left ...
```

Combing Parsers

```
parseFeatureName :: Parser String
parseFeatureName = do
  _ <- string "Feature: "
  word

>> runParser parseFeatureName "" "Feature: Parsing"
Right "Parsing"
>> runParser parseFeatureName "" "Feature:Parsing\n"
Left ...
```

Combing Parsers

```
parseFeatureName :: Parser String
parseFeatureName = do
  string "Feature:"
  hspace
  result <- word
  hspace
  newline
  return result

>> runParser parseFeatureName "" "Feature:Parsing\n"
Right "Parsing"
```

Combing Parsers

```
-- file.txt
Feature: parse
Feature: run
Feature: test
Feature: analyze
```

```
parseFeatures :: Parser [String]
parseFeatures = many parseFeatureName
```

```
>> runParser parseFeatures "file.txt" "..."
Right ["parse", "run", "test", "analyze"]
```

Backtracking

```
data ParseElement = Feature String | Case String

parseFeature :: Parser ParseElement
parseCase :: Parser ParseElement

>> runParser parseFeature "" "Feature: Parsing\n"
Right (Feature "Parsing")
>> runParser parseCase "" "Case: Test\n"
Right (Case "Test")
```

Backtracking

```
-- file.txt
Case: parse         <-- parseFeature -> Left ...
Feature: run        <-- parseCase       -> Left ...
Feature: test
Case: analyze
```

Backtracking

```
-- file.txt
Case: parse          <-- try parseFeature -> Left ...
Feature: run
Feature: test
Case: analyze
```

Backtracking

```
-- file.txt
Case: parse          <-- try parseCase -> Right (Case "parse")
Feature: run
Feature: test
Case: analyze
```

Backtracking

```
-- file.txt
Case: parse         <-- try parseCase -> Right (Case "parse")
Feature: run        <-- try parseFeature -> Right (Feature "run")
Feature: test
Case: analyze
```

`try` and alternative `(<|>)`

```
(<|>) :: m a -> m a -> m a

try :: Parser a -> Parser a


elementParser :: Parser ParseElement
elementParser = try parseFeature <|> try parseCase

parseFile :: Parser [ParseElement]
parseFile = many elementParser
```

Parsing the File

```
-- file.txt
Case: parse
Feature: run
Feature: test
Case: analyze
```

```haskell
parseFile :: Parser [ParseElement]
parseFile = many elementParser
```

```
>> runParser parseFile "file.txt" "..."
Right [ Case "parse", Feature "run"
      , Feature "test", Case "analyze"]
```

Without `try`

```
-- This will not work!
elementParser :: Parser ParseElement
elementParser = parseFeature <|> parseCase

parseFile :: Parser [ParseElement]
parseFile = many elementParser
```

Without `try`

```
-- file.txt
Case: parse         <-- parseFeature -> Left ...
Feature: run
Feature: test
Case: analyze
```

Without `try`

```
-- file.txt
ase: parse          <-- parseCase -> Left ...
Feature: run
Feature: test
Case: analyze
```

JSON Data

```
"Hello"

3012

True

Null
```

JSON Data

```
["Hello", 3012, [True, False], Null]

{
  "string": "hello",
  "number": 3012,
  "booleans": [True, False],
  "mapping": {"other_string": "World"}
}
```

JSON Type

```
data Value =
  String Text |
  Number Scientific |
  Bool Bool |
  Null |
  Array (Vector Value) |
  Object (KeyMap Value)
```

Conclusion

- Lots of material!
- Challenge: Write a JSON parser
- Next: One more data format to learn

Lecture 14

GEDCOM Data Format

Introduction

- One more challenge exercise!
- Geneological Data Communication format (GEDCOM)
- Data describes family trees and life events

GEDCOM Data

```
-- basic.ged
0 @I1@ INDI
1 NAME John /Smith/
1 SEX M
1 FAMS @F1@
0 @F1@ FAM
1 HUSB @I1@
```

Individual Lines

```
0 @I1@ INDI
...
```

Individual Lines

```
0 @I1@ INDI
1 NAME James /Smith/
1 SEX M
...
```

Individual Lines

```
0 @I1@ INDI
1 NAME James /Smith/
1 SEX M
1 FAMC @F1@
1 FAMS @F2@
```

Family Lines

```
0 @F2@ FAM
...
```

Family Lines

```
0 @F2@ FAM
1 HUSB @I1@
1 WIFE @I2@
1 CHIL @I3@
1 MARR
```

Haskell Data Types

```
data Individual = Individual
  { givenName    :: String
  , familyName   :: String
  , sex          :: SexOption
  , spouseFamily :: Maybe String
  , childFamily  :: Maybe String
  }

data SexOption = Male | Female
```

Haskell Data Types

```
data Family = Family
  { familyHusband  :: Maybe String
  , familyWife     :: Maybe String
  , familyChildren :: [String]
  , spousesMarried :: Bool
  }
```

Haskell Data Types

```
data FamilyTree = FamilyTree
  { individuals :: Map String Individual
  , families    :: Map String Family
  }
```

Conclusion

- Exercises have some outlines already
- Design the **monad stack**
- Will want at least one other monad!

Lecture 15

Conclusion

Congratulations!

You're done with Making Sense of Monads!

Course Structure

- **Simpler Functional Structures**
  - **Monoids, Semigroups, Functors, Applicatives**
- Basic Monads
- Reader, Writer, State Monads
- IO Monad
  - Terminal, file system interactions
- Functional Structure Laws

Course Structure

- Simpler Functional Structures
  - Monoids, Semigroups, Functors, Applicatives
- **Basic Monads**
- Reader, Writer, State Monads
- IO Monad
  - Terminal, file system interactions
- Functional Structure Laws

Course Structure

- Simpler Functional Structures
  - Monoids, Semigroups, Functors, Applicatives
- Basic Monads
- **Reader, Writer, State Monads**
- IO Monad
  - Terminal, file system interactions
- Functional Structure Laws

Course Structure

- Simpler Functional Structures
  - Monoids, Semigroups, Functors, Applicatives
- Basic Monads
- Reader, Writer, State Monads
- **IO Monad**
  - **Terminal, file system interactions**
- Functional Structure Laws

Course Structure

- Simpler Functional Structures
  - Monoids, Semigroups, Functors, Applicatives
- Basic Monads
- Reader, Writer, State Monads
- IO Monad
  - Terminal, file system interactions
- **Functional Structure Laws**