This week: Monads!
Monads

- Have already seen an example of a monad
  - IO monad
- But similar concepts can be used for a lot of completely unrelated tasks
- Monads are useful "general interfaces" to a wide variety of computational tasks
Monads

- Monads can act as generalized "containers"
  - e.g. List monad
- or as generalized "transformers" or "actions"
  - e.g. IO monad, State monad
- and many other things as well
- Don't get hung up on one viewpoint
  - all are valid
Category theory

- The word "Monad" comes from a branch of mathematics known as category theory.
  - However, we won't deal with category theory here.
  - If you're interested in this, I can talk more about this off-line.
  - CT is relevant but not strictly necessary to understand Haskell monads.
Monads

- Haskell defines a **Monad** type class like this:

```haskell
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    return :: a -> m a
    fail   :: String -> m a
```
Monads

What does this mean?

```haskell
class Monad m where
  (>>>=)  :: m a -> (a -> m b) -> m b
  (>>>)   :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a
```
Monads

- Let's ignore (>>>) and fail for now

```haskell
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a
```
Effects

- To explain further, we need to talk about the notion of functions with "effects"
- "Effects" may include input/output (IO monad), manipulating local or global state (State monad), raising exceptions (Error monad), possible failure (Maybe monad), or returning multiple values (List monad)
- or other possibilities!
Functions and effects (1)

- There are many kinds of "functions" or function-like actions that we might want to do that have effects beyond mapping specific inputs to specific outputs.
A normal function has the signature \( a \to b \), for some types \( a \) and \( b \).

If such a function also had some kind of "effect" (call it \( E \)), then we might write this as:

\[ a \to [E] \to b \]

I'll refer to functions with effects as "monadic functions".
A normal function of type \( a \rightarrow b \) can be composed with a function of type \( b \rightarrow c \) to give a function of type \( a \rightarrow c \).

How would we compose a function with effects (monadic function) with another such function?

How do we compose \( a \rightarrow [E1] \rightarrow b \) with \( b \rightarrow [E2] \rightarrow c \) to give a function \( a \rightarrow [E1,E2] \rightarrow c \)?
Haskell represents functions with effects i.e. $a \rightarrow [E] \rightarrow b$ as having the type $a \rightarrow E \ b$ where $E$ is some kind of a monad (like $\text{IO}$).

- We'll write $m$ instead of $E$ from now on.

- So we need to figure out how to compose functions of type $a \rightarrow m \ b$ with functions of type $b \rightarrow m \ c$ to get functions of type $a \rightarrow m \ c$. 
Functions and effects (5)

- Being able to compose functions with effects is critical, because we want to be able to build larger effectful functions by composing smaller effectful functions.

- Example: chaining together functions that read input from the terminal (in the \texttt{IO} monad) to functions that write output to the terminal (in the \texttt{IO} monad).
Functions and effects (6)

- We want to compose functions with types
  - \( f1 :: a \to m b \)
  - \( f2 :: b \to m c \)

- to get a function with type \( a \to m c \)

- We can pass a value of type \( a \) to \( f1 \) to get a value of type \( m b \)

- Then we need to somehow take the \( m b \) value, unpack a value of type \( b \) and pass it to \( f2 \) to get the final \( m c \) value
Functions and effects (7)

- How do we take the `m b` value, unpack a value of type `b` and pass it to `f2` to get the final `m c` value?

- The answer is specific to every monad
  - For `IO` it's kind of "magical"; the system takes care of it

- This is why there is the `>>= ` function in the `Monad` type class, with the type signature
  
  ```haskell
  m a -> (a -> m b) -> m b
  ```
Functions and effects (8)

- **Note**: the type signature:
  - \( m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b \)

- is the same as:
  - \( m \ b \rightarrow (b \rightarrow m \ c) \rightarrow m \ c \)
  - (just change the type variable names)

- so this is indeed what we want
The bind operator:

\[(\gg\gg=) \:: \cdot m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b\]

is thus a kind of "monadic apply operator" which takes a "monadic value" (of type \(m\ a\)), unpacks a value of type \(a\) somehow, and feeds it to the "monadic function" (of type \(a \rightarrow m\ b\)) to get the final monadic value (of type \(m\ b\))
The bind operator:

\[(>>=) : : m a \rightarrow (a \rightarrow m b) \rightarrow m b\]

is part of the Monad type class, so it has a separate (overloaded) definition for every instance of the Monad type class such as IO, State, Error, Maybe, List, etc.
Monad definition again

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

- Note that instances of `Monad` (i.e. `m`) must be polymorphic type constructors
  - `m` is a type constructor, `m a` is a type
- Whereas instances of `Eq`, `Ord` etc. are just regular types (not type constructors)
Monad definition again

- N.B. \texttt{IO} is a type constructor, so \texttt{IO} can substitute for \texttt{m} here:

```haskell
instance Monad IO where
  (>>=) :: IO a -> (a -> IO b) -> IO b
  (definition omitted)

return :: a -> IO a
  (definition omitted)
```
Monad laws

- Haskell's monads must obey these laws:
  1) \((\text{return } x) \gg= f \equiv f x\)
  2) \(m x \gg= \text{return} \equiv m x\)
  3) \((m x \gg= f) \gg= g \equiv m x \gg= (\lambda x \rightarrow f x \gg= g)\)

- (1) and (2) are sorta-kind identity laws
- (3) is sorta-kind an associative law
- (here, \(m x\) is a value of type \(m x\))
Note

3) \( (mx >>= f) >>= g \) ==
\[
mx >>= (\lambda x . f x >>= g)
\]

- Can write this as:

   3) \( (mx >>= (\lambda x . f x)) >>= g \) ==
\[
mx >>= (\lambda x . (f x >>= g))
\]

- Slightly more intuitive
Monad laws (2)

- Monad laws just ensure that composing of monadic functions behaves properly.
- Can re-write them in terms of the monadic composition operator $\gg\gg$, which we haven't seen before.
- $(\gg\gg) :: (a \to m b) \to (b \to m c) \to (a \to m c)$
- (This can be found in the module `Control.Monad`, if you're curious)
Monad laws (3)

- In terms of \((\Rightarrow\Rightarrow)\), and monadic functions
  - \(mf :: a \rightarrow m b\)
  - \(mg :: b \rightarrow m c\)
  - \(mh :: c \rightarrow m d\)

- the monad laws become:
  - 1) \(\text{return} \Rightarrow\Rightarrow mf = mf\) (left identity)
  - 2) \(mf \Rightarrow\Rightarrow \text{return} = mf\) (right identity)
  - 3) \(mf \Rightarrow\Rightarrow (mg \Rightarrow\Rightarrow mh) = (mf \Rightarrow\Rightarrow mg) \Rightarrow\Rightarrow mh\) (associativity)
Monad laws (4)

- Haskell doesn't (and can't) enforce the monad laws!
  - it's not that powerful (not a theorem prover!)
- It's up to the designer of every Monad instance to make sure that these laws are valid
- This often strongly determines why a particular monad has the definitions it does for return and (>>=) (especially return)
- >>>= is the "bind" operator
- What does this do, again?
- x >>>= f
- >>>= "unpacks" component of type \( a \) from a value of type \( m \ a \)
- and applies function \( f \) to it to get value of type \( m \ b \) (since \( f :: a \rightarrow m \ b \))
(monadic composition) can trivially be defined in terms of >>=

\( f_1 \gg f_2 = \lambda x \to (f_1 \ x \gg f_2) \)

So \( \gg \gg \) (monadic application) is the important concept
can also be defined in terms of $\ggg$:

$$a \ggg b = a \ggg \_ \rightarrow b$$

This is the default

Used when "contents" or "return value" of monad not needed for next operation

*Example:* `putStr :: String -> IO ()`

- `()` "result" of monad isn't needed for further operations
instance Monad Maybe where
    (Just x) >>= f   =  f x
    Nothing >>= f   =  Nothing
    return           =  Just

instance Monad [] where
    lst >>= f = concat (map f lst)
    return x = [x]

    -- and IO monad is mostly built-in
So the list polymorphic type is a monad
And the *Maybe* polymorphic type is also a monad
Big deal... what does this buy us?
**Maybe monad (1)**

- `Maybe` type:

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

- Can be used to represent computations that may fail

- Can use monadic infrastructure to chain together computations that can fail in a nice way
instance Monad Maybe where
    (Just x) >>= f   =  f x
Nothing     >>= f   =  Nothing
return       =  Just

- Meaning?
- **Nothing** stays **Nothing** even through **>>=** operator
- x unpacked from **Just x** and given to f
We'll work through an example involving a population of sheep.

This will be a good opportunity to learn more about lamb-das:
- (Thanks to John Wagner for that observation!)
- Hopefully, nothing ba-a-a-d will happen.
Maybe monad (3)

data Sheep = ... 
father :: Sheep -> Maybe Sheep
father = ...
mother :: Sheep -> Maybe Sheep
mother = ...

maternalGrandfather :: Sheep -> Maybe Sheep

maternalGrandfather s =
    case (mother s) of
        Nothing -> Nothing
        Just m  -> father m
Maybe monad (5)

```haskell
mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s =
    case (mother s) of
        Nothing -> Nothing
        Just m -> case (father m) of
            Nothing -> Nothing
            Just gf -> father gf
```

- As functions get more complex, this gets uglier and uglier due to nested `case` statements
"Use the monadic way, Luke!"
-- Obi-wan Curry

maternalGrandfather s =
  (return s) >>= mother >>= father

mothersPaternalGrandfather s =
  (return s) >>= mother >>= father >>= father
Or with syntactic sugar:

```haskell
maternalGrandfather s =
  do m <- mother s
     father m

mothersPaternalGrandfather s =
  do m <- mother s
     f <- father m
     father f
```
**do notation (1)**

- `maternalGrandfather s =
  do m <- mother s
  father m`

- is equivalent to:

- `maternalGrandfather s =
  mother s >>= \m ->
  father m`
do notation (2)

```haskell
mothersPaternalGrandfather s =
  do m <- mother s
     f <- father m
     father f
```

is equivalent to:

```haskell
fathersMaternalGrandmother s =
  mother s >>= \m ->
     father m >>= \f ->
     father f
```
do notation (3)

Note: parse:

\[
\text{mothersMaternalGrandmother } s = \\
\quad \text{mother } s >>= \ m \rightarrow \\
\quad \quad \text{father } m >>= \ f \rightarrow \\
\quad \quad \quad \text{father } f
\]

as:

\[
\text{mothersMaternalGrandmother } s = \\
\quad \text{mother } s >>= (\ m \rightarrow \\
\quad \quad \text{father } m >>= (\ f \rightarrow \\
\quad \quad \quad \text{father } f))
\]
Moral

- Monadic form will keep computations involving `Maybe` types manageable
  - no matter how deeply nested the computations get
- Code is more readable, more maintainable, much less prone to stupid errors
List monad (1)

- Lists can be used to represent functions that can have multiple possible results
  - or no results (empty list)
- Simple example:
  - Take two numbers
  - For each, generate list of numbers within 1 of original number
  - Add two such "fuzzy numbers" together
List monad (2)

- Recall...

```haskell
instance Monad [] where
    lst >>= f = concat (map f lst)
    return x = [x]
```

- Meaning?
- Let's work through an evaluation
List monad (3)

```haskell
fuzzy :: Int -> [Int]
fuzzy n = [n-1, n+1]
addFuzzy :: [Int] -> [Int] -> [Int]
addFuzzy f1 f2 = do n1 <- f1
                     n2 <- f2
                     return (n1 + n2)
(fuzzy 10) `addFuzzy` (fuzzy 20)  
→ [28, 30, 30, 32]
```
List monad (4)

- desugared version:

\[
\text{addFuzzy \ ((fuzzy\ 10)\ (fuzzy\ 20)) = addFuzzy\ [9,\ 11]\ [19,\ 21] = [9,\ 11]\ >>=\ (\n1 -> [19,\ 21]\ >>=\ (\n2 -> return\ (n1 + n2))))}
\]
List monad (5)

\[ [9, 11] >>= (\n1 \rightarrow [19, 21] >>= (\n2 \rightarrow return (n1 + n2)) \)

\[ [9, 11] >>= (\n1 \rightarrow [19, 21] >>= (\n2 \rightarrow [n1 + n2])) \] -- def'n of return
List monad (6)

\[ [9, 11] \triangleright \triangleright = (\ \lambda n1 \rightarrow
\quad \quad [19, 21] \triangleright \triangleright = (\ \lambda n2 \rightarrow
\quad \quad \quad [n1 + n2])\)\]

\[ \rightarrow \]

\[ [9, 11] \triangleright \triangleright = (\ \lambda n1 \rightarrow
\quad \quad \text{concat} (\text{map} (\ \lambda n2 \rightarrow [n1 + n2])
\quad \quad \quad [19, 21]))\)

\[ \text{-- def'n of (\triangleright \triangleright=) }\]
List monad (7)

\[ [9, 11] >>= (\n1 ->
    \n1 + n2])
\]

\[ [9, 11] >>= (\n1 ->
    \n1 + 19, \n1 + 21]) \]

\[ [9, 11] >>= (\n1 ->
    \n1 + 19, \n1 + 21]) \]
List monad (8)

\[ [9, 11] >>= (\n1 ->
    concat [[n1 + 19], [n1 + 21]]) \]

\rightarrow

\[ [9, 11] >>= (\n1 ->
    [n1 + 19, n1 + 21]) \]

\rightarrow

concat (map (\n1 -> [n1 + 19, n1 + 21])
      [9, 11])
List monad (9)

concat (map (\n1 -> [n1 + 19, n1 + 21]) [9, 11])

\n\n\n\n[28, 30, 30, 32]

- And we're done!
List monad (10)

- Even better:

```haskell
addFuzzy f1 f2 =
    let vals = do n1 <- f1
                 n2 <- f2
                 return (n1 + n2)
    in [minList vals, maxList vals]
    where minList = foldl1 min
             maxList = foldl1 max

(fuzzy 10) `addFuzzy` (fuzzy 20)  \rightarrow  [28, 32]
```
List monad (11)

- List monadic computations are also isomorphic to list comprehensions

- Can add filters to do-notation:
  ```
do  x  <-  [1..6]
    y  <-  [1..6]
    if  x  +  y  ==  7
      then  return  (x,  y)
    else  []
  -->  [(1, 6),  (2, 5),  (3, 4),
       (4, 3),  (5, 2),  (6, 1)]
```
References

- "All About Monads" by Jeff Newbern
  - Very in-depth discussion, examples of many different monads

- "Yet Another Monad Tutorial" by me
  - 8-part series (so far!)
  - Incredibly detailed
Next week

- More about monads
  - State monads (very important)
  - MonadZero and MonadPlus type classes