Monads

Stephen A. Edwards

Columbia University

Fall 2020
Motivating Example: lookup3

The Monad Type Class
   The Maybe Monad
do Blocks
   The Either Monad
Monad Laws

The List Monad
   List Comprehensions as a Monad

The MonadPlus Type Class and guard

The Writer Monad

Some Monadic Functions: liftM, ap, join, filterM, foldM, mapM, sequence

Functions as Monads

The State Monad
   An Interpreter for a Simple Imperative Language
Motivating Example: Chasing References in a Dictionary

In Data.Map, 

\[
\text{lookup} :: \text{Ord } k \Rightarrow k \rightarrow \text{Map } k \ a \rightarrow \text{Maybe } a
\]

Say we want a function that uses a key to look up a value, then treat that value as another key to look up a third key, which we look up and return, e.g.,

\[
\text{lookup3} :: \text{Ord } k \Rightarrow k \rightarrow \text{Map.Map } k\ k \rightarrow \text{Maybe } k
\]

Prelude\> import qualified Data.Map.Strict as Map
Prelude Map\> myMap = Map.fromList [("One","Two"),("Two","Three"),
                                          ("Three","Winner")]
Prelude Map\> Map.lookup "One" myMap
Just "Two"
Prelude Map\> Map.lookup "Two" myMap
Just "Three"
Prelude Map\> Map.lookup "Three" myMap
Just "Winner"
A First Attempt

lookup3 :: Ord k => k -> Map.Map k k -> Maybe k  -- First try
lookup3 k1 m = case Map.lookup k1 m of
    Nothing  -> Nothing
    Just k2  -> case Map.lookup k2 m of
        Nothing  -> Nothing
        Just k3  -> Map.lookup k3 m

Too much repeated code, but it works.

*Main Map> lookup3 "Three" myMap
Nothing
*Main Map> lookup3 "Two" myMap
Nothing
*Main Map> lookup3 "One" myMap
Just "Winner"
What’s the Repeated Pattern Here?

Nothing -> Nothing
Just k2 -> case Map.lookup k2 m of ...

“Pattern match on a Maybe. Nothing returns Nothing, otherwise, strip out the payload from the Just and use it as an argument to a lookup lookup.”

lookup3 :: Ord k => k -> Map.Map k k -> Maybe k  -- Second try
lookup3 k1 m = (helper . helper . helper) (Just k1)
  where helper Nothing  = Nothing
         helper (Just k) = Map.lookup k m

This looks a job for a Functor or Applicative Functor...

class Functor f where
  fmap :: (a -> b) -> f a -> f b  -- Apply function to data in context
class Functor f => Applicative f where
  (<*>): f (a -> b) -> f a -> f b  -- Apply a function in a context

..but these don’t fit because our steps take a key and return a key in context.
Even Better: An “ifJust” Function

```haskell
ifJust :: Maybe k -> (k -> Maybe k) -> Maybe k
ifJust Nothing _  = Nothing -- Failure: nothing more to do
ifJust (Just k) f = f k    -- Success: pass k to the function

lookup3 :: Ord k => k -> Map.Map k k -> Maybe k
lookup3 k1 m = ifJust (Map.lookup k1 m)
               (
k2   -> ifJust (Map.lookup k2 m)
                       (
k3   -> Map.lookup k3 m))
```

It's cleaner to write `ifJust` as an infix operator:

```haskell
lookup3 :: Ord k => k -> Map.Map k k -> Maybe k
lookup3 k1 m = Map.lookup k1 m `ifJust`
               \k2   -> Map.lookup k2 m `ifJust`
               \k3   -> Map.lookup k3 m
```
The Monad Type Class: It’s All About That Bind

\[
\text{infixl 1 } \gg= \\
\text{class } \text{Applicative } m \Rightarrow \text{Monad } m \text{ where} \\
  (\gg=) :: m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b \quad \text{-- “Bind”} \\
\text{return} :: a \rightarrow m \ a \quad \text{-- Wrap a result in the Monad}
\]

Bind, \(\gg=\), is the operator missing from the Functor and Applicative Functor type classes. It allows chaining context-producing functions

\[
\text{pure} :: b \rightarrow f \ b \quad \text{-- Put value in context} \\
\text{fmap} :: (a \rightarrow b) \rightarrow f \ a \rightarrow f \ b \quad \text{-- Apply function in context} \\
(\langle\ast\rangle) :: f (a \rightarrow b) \rightarrow f \ a \rightarrow f \ b \quad \text{-- Function itself is in context} \\
"\gg=" :: (a \rightarrow f \ b) \rightarrow f \ a \rightarrow f \ b \quad \text{-- Apply a context-producing func.}
\]
Actually, Monad is a little bigger

```haskell
infixl 1 >> >>=
class Monad m where
    -- The bind operator: apply the result in a Monad to a Monad producer
    (>>=) :: m a -> (a -> m b) -> m b
    
    -- Encapsulate a value in the Monad
    return :: a -> m a

    -- Like >>= but discard the result; often m () -> m b -> m b
    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y
    
    -- The default, which usually suffices

    -- Internal: added by the compiler to handle failed pattern matches
    fail :: String -> m a
    fail msg = error msg
```
Maybe is a Monad

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

instance Monad Maybe where  -- Standard Prelude definition
  return x = Just x          -- Wrap in a Just

  Just x >>= f = f x         -- Our “ifjust” function
  Nothing >>= _ = Nothing    -- “computation failed”

  fail _ = Nothing           -- fail quietly
The Maybe Monad in Action

Prelude> :t return "what?"
return "what?" :: Monad m => m [Char]

Prelude> return "what?" :: Maybe String
Just "what?"

Prelude> Just 9 >>= \x -> return (x*10)
Just 90

Prelude> Just 9 >>= \x -> return (x*10) >>= \y -> return (y+5)
Just 95

Prelude> Just 9 >>= \x -> Nothing >>= \y -> return (x+5)
Nothing

Prelude> Just 9 >> return 8 >>= \y -> return (y*10)
Just 80

Prelude> Just 9 >>= \_ -> fail "darn" >>= \x -> return (x*10)
Nothing
lookup3 using Monads

```
instance Monad Maybe where
    return x = Just x

Just x >>= f = f x       -- Apply f to last (successful) result
Nothing >>= _ = Nothing  -- Give up
```

```
lookup3 :: Ord k => k -> Map.Map k k -> Maybe k
lookup3 k1 m = Map.lookup k1 m >>= \k2 -> Map.lookup k2 m >>= \k3 -> Map.lookup k3 m
```

Or, equivalently,

```
lookup3 :: Ord k => k -> Map.Map k k -> Maybe k
lookup3 k1 m = Map.lookup k1 m >>= \k2 -> Map.lookup k2 m >>= \k3 -> Map.lookup k3 m
```
Monads and the do Keyword: Not Just For I/O

Monads are so useful, Haskell provides do notation to code them succinctly:

```haskell
lookup3 :: Ord k => 
  k -> Map.Map k k -> Maybe k
lookup3 k1 m = do
  k2 <- Map.lookup k1 m
  k3 <- Map.lookup k2 m
  Map.lookup k3 m
```

These are semantically identical. do inserts the >>=‘s and lambdas.

Note: each lambda’s argument moves to the left of the expression

```haskell
k2 <- Map.lookup k1 m
```

```haskell
Map.lookup k1 m >>= \k2 ->
```
Like an Applicative Functor

```
Prelude> (+) <$> Just 5 <*> Just 3
Just 8
Prelude> do
Prelude|   x <- Just (5 :: Int)
Prelude|   y <- return 3
Prelude|   return (x + y)
Just 8
Prelude> :t it
it :: Maybe Int
```

The Monad's type may change; "Nothing" halts and forces Maybe

```
Prelude> do
Prelude|   x <- return 5
Prelude|   y <- return "ha!"
Prelude|   Nothing
Prelude|   return x
Nothing
```

fail is called when a pattern match fails

```
Prelude> do
Prelude|   (x:xs) <- Just "Hello"
Prelude|   return x
Just 'H'
Prelude> do
Prelude|   (x:xs) <- Just []
Prelude|   return x
Nothing
```
Like Maybe, Either is a Monad

```
data Either a b = Left a | Right b  -- Data.Either

instance Monad (Either e) where
  return x = Right x

  Right x >>= f = f x  -- Right: keep the computation going

  Left err >>= _ = Left err  -- Left: something went wrong

Prelude> do
Prelude|   x <- Right "Hello"
Prelude|   y <- return "World"
Prelude|   return $ x ++ y
Right "Hello World"

Prelude> do
Prelude|   Right "Hello"
Prelude|   x <- Left "failed"
Prelude|   y <- Right $ x ++ "darn"
Prelude|   return y
Left "failed"
```
Monad Laws

Left identity: applying a wrapped argument with >>= just applies the function

\[ \text{return } x >>= f = f x \]

Right identity: using >>= to unwrap then return to wrap does nothing

\[ m >>= \text{return } = m \]

Associative: applying \( g \) after applying \( f \) is like applying \( f \) composed with \( g \)

\[ (m >>= f) >>= g = m >>= (\lambda x \to f x >>= g) \]
The List Monad: “Nondeterministic Computation”

Intuition: lists represent all possible results

```haskell
instance Monad [] where
  return x = [x]  -- Exactly one result
  xs >>= f = concat (map f xs)  -- Collect all possible results from f
  fail _ = []  -- Error: “no possible result”
```

Prelude> [10,20,30] >>= \x -> [x-3, x, x+3]
[7,10,13,17,20,23,27,30,33]

“If we start with 10, 20, or 30, then either subtract 3, do nothing, or add 3, we will get 7 or 10 or 13 or 17 or ..., or 33”

```
[10,20,30] >>= \x -> [x-3, x, x+3]
  = concat (map (\x -> [x-3, x, x+3]) [10,20,30])
  = concat [[7,10,13],[17,20,23],[27,30,33]]
  = [7,10,13,17,20,23,27,30,33]
```
The List Monad

Everything needs to produce a list, but the lists may be of different types:

```
Prelude> [1,2] >>= \x -> ['a','b'] >>= \c -> [(x,c)]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

This works because -> is at a lower level of precedence than >>=

```
[1,2] >>= \x -> ['a','b'] >>= \c -> [(x,c)]
= [1,2] >>= (\x -> ([ 'a','b' ] >>= (\c -> [(x,c)]))
= [1,2] >>= (\x -> (concat (map (\c -> [(x,c)])) ['a','b']))
= [1,2] >>= (\x -> [(x,'a'),(x,'b')])
= concat (map (\x -> [(x,'a'),(x,'b')]) [1,2])
= concat [[(1,'a'),(1,'b')],[(2,'a'),(2,'b')]]
= [(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```
The List Monad, do Notation, and List Comprehensions

\[(1,2) \gg= \lambda x \rightarrow [\text{'a'}, \text{'b'}] \gg= \lambda c \rightarrow \text{return} \ (x, c)\]

\[(1,2) \gg= \lambda x \rightarrow [\text{'a'}, \text{'b'}] \gg= \lambda c \rightarrow \text{return} \ (x, c)\]

do \ x \leftarrow [1,2] \quad -- \text{Send 1 and 2 to the function that takes } x \text{ and}
\quad c \leftarrow [\text{'a'}, \text{'b'}] \quad -- \text{sends 'a' and 'b' to the function that takes } c \text{ and}
\quad \text{return} \ (x, c) \quad -- \text{wraps the pair } (x, c)\]

\[\{(x,c) | x \leftarrow [1,2], c \leftarrow [\text{'a'}, \text{'b'}]\}\]

each produce

\[
\begin{bmatrix}
(1, \text{'a'}), (1, \text{'b'}), (2, \text{'a'}), (2, \text{'b'})
\end{bmatrix}
\]
class Monad m => MonadPlus m where  -- In Control.Monad
  mzero :: m a          -- “Fail,” like Monoid’s mempty
  mplus :: m a -> m a -> m a  -- “Alternative,” like Monoid’s mappend

instance MonadPlus [] where
  mzero = []
  mplus = (++)

guard :: MonadPlus m => Bool -> m ()
guard True  = return ()  -- In whatever Monad you’re using
guard False = mzero       -- “Empty” value in the Monad
Using Control.Monad.guard as a filter

guard uses \texttt{mzero} to terminate a MonadPlus computation (e.g., Maybe, [], IO)
It either succeeds and returns () or fails. We never care about (), so use >>

\[
\begin{align*}
[1..50] & \triangleright= \backslash x \rightarrow \\
& \text{guard } (x \ `\text{rem}` 7 == 0) \triangleright \quad \text{-- Discard any returned ()} \\
& \text{return } x
\end{align*}
\]

\[
\begin{align*}
\text{do } x & \leftarrow [1..50] \\
& \text{guard } (x \ `\text{rem}` 7 == 0) \quad \text{-- No } \leftarrow \text{ makes for an implicit } \triangleright \\
& \text{return } x
\end{align*}
\]

\[
[ x | x \leftarrow [1..50], x `\text{rem}` 7 == 0 ]
\]

each produce

\[ [7,14,21,28,35,42,49] \]
The Control.Monad.Writer Monad

For computations that return a value and accumulate a result in a Monoid, e.g., logging or code generation. Just a wrapper around a (value, log) pair.

In Control.Monad.Writer,

```haskell
newtype Writer w a = Writer { runWriter :: (a, w) }

instance Monoid w => Monad (Writer w) where
  return x = Writer (x, mempty) -- Append nothing
  Writer (x, l) >>= f = let Writer (y, l') = f x in
    Writer (y, l `mappend` l') -- Append to log
```

*a* is the result value  
*w* is the accumulating log Monoid (e.g., a list) 

`runWriter` extracts the (value, log) pair from a `Writer` computation
The Writer Monad in Action

```haskell
import Control.Monad.Writer

logEx :: Int -> Writer [String] Int  -- Type of log, result
logEx a = do
    tell "logEx " ++ show a           -- Just log
    b <- return 42                    -- No log
    tell "b = " ++ show a
    c <- writer (a + b + 10, ["compute c"])  -- Value and log
    tell "c = " ++ show c
    return c

*Main> runWriter (logEx 100)
(152,["logEx 100","b = 100","compute c","c = 152"])
```
Verbose GCD with the Writer

```haskell
*Main> mapM_ putStrLn $ snd $ runWriter $ logGCD 9 3
logGCD 9 3
a > b
logGCD 6 9
a < b
logGCD 6 3
a > b
logGCD 3 6
a < b
logGCD 3 3
finished
```

```haskell
import Control.Monad.Writer

logGCD :: Int -> Int -> Writer [String] Int
logGCD a b = do
  tell ["logGCD " ++ show a ++ " " ++ show b]
  if a == b then do
    writer (a, ["finished"])
  else if a < b then do
    tell ["a < b"]
    logGCD a (b - a)
  else do
    tell ["a > b"]
    logGCD (a - b) a
```
Control.Monad.{liftM, ap}: Monads as Functors

\[ \text{fmap} \, f \quad \Rightarrow \quad (a \rightarrow b) \rightarrow f \ a \rightarrow f \ b \quad \text{-- a.k.a.} \quad <\$> \]

\[ (\langle\ast\rangle) \, : \, \text{Applicative} \; f \; \Rightarrow \; f \; (a \rightarrow b) \rightarrow f \; a \rightarrow f \; b \quad \text{-- “apply”} \]

In Monad-land, these have alternative names

\[ \text{liftM} \, : \, \text{Monad} \; m \; \Rightarrow \; (a \rightarrow b) \rightarrow m \; a \rightarrow m \; b \]

\[ \text{ap} \, : \, \text{Monad} \; m \; \Rightarrow \; m \; (a \rightarrow b) \rightarrow m \; a \rightarrow m \; b \]

and can be implemented with \( \gg= \) (or, equivalently, do notation)

\[ \text{liftM} \; f \; m \; = \; \text{do} \; x \leftarrow m \quad \text{-- Get the argument from inside } m \]
\[ \quad \text{return} \; (f \; x) \quad \text{-- Apply the argument to the function} \]

\[ \text{ap} \; mf \; m \; = \; \text{do} \; f \leftarrow mf \quad \text{-- Get the function from inside } mf \]
\[ \quad x \leftarrow m \quad \text{-- Get the argument from inside } m \]
\[ \quad \text{return} \; (f \; x) \quad \text{-- Apply the argument to the function} \]

Operations in a \textit{do} block are ordered: \( \text{ap} \) evaluates its arguments left-to-right
liftM and ap In Action

```haskell
liftM :: Monad m => (a -> b) -> m a -> m b
ap :: Monad m => m (a -> b) -> m a -> m b
```

```
Prelude> import Control.Monad
Prelude Control.Monad> liftM (map Data.Char.toUpper) getline
"HELLO"
```

Evaluate (+10) 42, but keep a log:

```haskell
Prelude> :set prompt "> "
> :set prompt-cont "| "
> import Control.Monad.Writer
> :{
| runWriter $
| ap (writer ((+10), ["first"])) (writer (42, ["second"]))
| :}
(52,["first","second"])
```
Lots of Lifting: Applying two- and three-argument functions

In Control.Applicative, applying a normal function to Applicative arguments:

\[
\text{liftA2 :: Applicative } \Rightarrow (a \rightarrow b \rightarrow c) \rightarrow f \ a \rightarrow f \ b \rightarrow f \ c
\]

\[
\text{liftA3 :: Applicative } \Rightarrow (a \rightarrow b \rightarrow c \rightarrow d) \rightarrow f \ a \rightarrow f \ b \rightarrow f \ c \rightarrow f \ d
\]

In Control.Monad,

\[
\text{liftM2 :: Monad } \Rightarrow (a \rightarrow b \rightarrow c) \rightarrow m \ a \rightarrow m \ b \rightarrow m \ c
\]

\[
\text{liftM3 :: Monad } \Rightarrow (a \rightarrow b \rightarrow c \rightarrow d) \rightarrow m \ a \rightarrow m \ b \rightarrow m \ c \rightarrow m \ d
\]

Example: lift the pairing operator \((,\)\) to the Maybe Monad:

Prelude Control.Monad> \text{liftM2 (,) (Just 'a') (Just 'b')} 
Just ('a','b')

Prelude Control.Monad> \text{liftM2 (,) Nothing (Just 'b')} 
Nothing
join: Unwrapping a Wrapped Monad/Combining Objects

\[
\text{join} :: \text{Monad} \ m \Rightarrow m (m \ a) \rightarrow m \ a \quad \text{--- in Control.Monad}
\]

\[
\text{join } mm = \text{do } m <- mm \quad \text{--- Remove the outer Monad; get the inner one}
\quad m \quad \text{--- Pass it back verbatim (i.e., without wrapping it)}
\]

\textit{join} is boring on a Monad like Maybe, where it merely strips off a “Just”

\begin{verbatim}
Prelude Control.Monad> join (Just (Just 3))
Just 3
\end{verbatim}

For Monads that hold multiple objects, \textit{join} lives up to its name and performs some sort of concatenation

\begin{verbatim}
> join ["Hello", "Monadic", "World!"]
"Hello Monadic World!"
\end{verbatim}

\[
\text{join } (\text{liftM } f \ m) \text{ is the same as } m >>= f
\]

“Apply } f \text{ to every object in } m \text{ and collect the results in the same Monad”
sequence: “Execute” a List of Actions in Monad-Land

Change a list of Monad-wrapped objects into a Monad-wrapped list of objects

\[
\text{sequence} :: [\text{m a}] \to \text{m} [\text{a}]
\]

\[
\text{sequence}_\_ :: [\text{m a}] \to \text{m} ()
\]

Prelude> \text{sequence} [\text{print 1, print 2, print 3}]
1
2
3
[(),(),()]

Prelude> \text{sequence}_\_ [\text{putStrLn "Hello", putStrLn "World"}]
Hello
World

Works more generally on Traversable types, not just lists
mapM: Map Over a List in Monad-Land

\[
\text{mapM} :: \text{Monad } m \Rightarrow (a \rightarrow m b) \rightarrow [a] \rightarrow m [b]
\]

\[
\text{mapM} _ :: \text{Monad } m \Rightarrow (a \rightarrow m b) \rightarrow [a] \rightarrow m () \quad -- \text{Discard result}
\]

Add 10 to each list element and log having seen it:

\[
\text{> p10 x = writer } (x+10, ["saw " ++ show x]) :: \text{Writer } [\text{String}] \text{ Int}
\]

\[
\text{> runWriter } \$ \text{mapM p10 } [1..3]
\]

\[
([11,12,13],["saw 1","saw 2","saw 3"])
\]

Printing the elements of a list is my favorite use of mapM_:

\[
\text{> mapM_ print } ([1..3] :: [\text{Int}])
\]

\[
1
2
3
\]

Works more generally on Traversable types, not just lists
Control.Monad.foldM: Left-Fold a List in Monad-Land

\[ \text{foldl} :: \quad (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a \]

In \text{foldM}, the folding function operates and returns a result in a Monad:

\[ \text{foldM} :: \text{Monad} m =\Rightarrow (a \rightarrow b \rightarrow m a) \rightarrow a \rightarrow [b] \rightarrow m a \]

\[ \text{foldM} f a1 [x1, x2, \ldots, xm] = \text{do} \quad a2 <- f a1 x1 \\
\phantom{\text{do} \quad a2 <- f a1 x1} \quad a3 <- f a2 x2 \\
\phantom{\text{do} \quad a2 <- f a1 x1} \quad \ldots \\
\phantom{\text{do} \quad a2 <- f a1 x1} \quad f a m x m \]

Example: Sum a list of numbers and report progress

\[ > \text{runWriter} \$ \text{foldM} (\backslash a \ x \rightarrow \text{writer} (a+x, [(x,a)])) \ 0 \ [1..4] \]
\[ (10,[(1,0),(2,1),(3,3),(4,6)]) \]

“Add value \( x \) to accumulated result \( a \); log \( x \) and \( a \)”

\[ \backslash a \ x \rightarrow \text{writer} (a+x, [(x,a)]) \]
**Control.Monad.filterM**: Filter a List in Monad-land

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p = foldr (\x acc -> if p x then x : acc else acc) []
```

```haskell
filterM :: Monad m => (a -> m Bool) -> [a] -> m [a]
filterM p = foldr (\x -> liftM2 (\k -> if k then (x:) else id) (p x)) (return []) (return [])
```

**filterM in action**: preserve small list elements; log progress

```haskell
isSmall :: Int -> Writer [String] Bool
isSmall x | x < 4   = writer (True, ["keep " ++ show x])
           | otherwise = writer (False, ["reject " ++ show x])
```

```haskell
> fst $ runWriter $ filterM isSmall [9,1,5,2,10,3]
[1,2,3]
> snd $ runWriter $ filterM isSmall [9,1,5,2,10,3]
["reject 9","keep 1","reject 5","keep 2","reject 10","keep 3"]
```
An Aside: Computing the Powerset of a List

For a list \([x_1, x_2, \ldots]\), the answer consists of two kinds of lists:

\[
\begin{bmatrix}
[x_1, x_2, \ldots], \ldots, [x_1],
[x_2, x_3, \ldots], \ldots, []
\end{bmatrix}
\]

- start with \(x_1\)
- do not start with \(x_1\)

\[
powerset :: [a] \rightarrow [[a]]
\]
\[
powerset [] = [[]] \quad \text{-- Tricky base case: } 2^\varnothing = \{\varnothing\}
\]
\[
powerset (x:xs) = \text{map} \ (x:) \ (\text{powerset} \ xs) ++ \text{powerset} \ xs
\]

\[
*\text{Main}*> \text{powerset} \ "abc"
\[
["abc","ab","ac","a","bc","b","c",""]
\]
The List Monad and Powersets

\[
\text{powerset} (x:xs) = \text{map} (x:) (\text{powerset} \ xs) ++ \text{powerset} \ xs
\]

Let’s perform this step (i.e., possibly prepending \(x\) and combining) using the list Monad. Recall \(\text{liftM2}\) applies Monadic arguments to a two-input function:

\[
\text{liftM2} :: \text{Monad} \ m \Rightarrow (a \to b \to c) \to m \ a \to m \ b \to m \ c
\]

So, for example, if \(a = \text{Bool}\), \(b \& c = [\text{Char}]\), and \(m\) is a list,

\[
\text{listM2} :: (\text{Bool} \rightarrow [\text{Char}] \rightarrow [\text{Char}]) \rightarrow [\text{Bool}] \rightarrow [[\text{Char}]] \rightarrow [[\text{Char}]]
\]

\[
> \text{liftM2} (\lambda k \rightarrow \text{if } k \text{ then ('a':) else id}) \ [\text{True, False}] \ [["bc", "d"]\n\ ["abc","ad","bc","d"]
\]

\(\text{liftM2}\) makes the function “nondeterministic” by applying the function with every \(\text{Bool}\) in the first argument, i.e., both \(k = \text{True}\) (include ‘\(a\’) and \(k = \text{False}\) (do not include ‘\(a\’) ), to every string in the second argument (\(["bc","d"]\) )
filterM Computes a Powerset: Like a Haiku, but shorter

\[ \text{foldr } f \ z \ [x_1, x_2, \ldots, x_n] = f \ x_1 \ (f \ x_2 \ (\ldots \ (f \ x_n \ z) \ldots)) \]

\[ \text{filterM } p \ = \ \text{foldr} \ (\lambda x \rightarrow \text{liftM2} \ (\lambda k \rightarrow \text{if } k \text{ then } (x:) \text{ else } \text{id}) \ (p \ x)) \ (\text{return } []) \]

\[ \text{filterM } p \ [x_1, x_2, \ldots, x_n] = \]
\[ \text{liftM2} \ (\lambda k \rightarrow \text{if } k \text{ then } (x_1:) \text{ else } \text{id}) \ (p \ x_1) \]
\[ \ (\text{liftM2} \ (\lambda k \rightarrow \text{if } k \text{ then } (x_2:) \text{ else } \text{id}) \ (p \ x_2) \]
\[ \ldots \]
\[ \ (\text{liftM2} \ (\lambda k \rightarrow \text{if } k \text{ then } (x_n:) \text{ else } \text{id}) \ (p \ x_n) \ (\text{return } [])) \ldots \]

If we let \( p \_ = [\text{True, False}] \), this chooses to prepend \( x_1 \) or not to the result of prepending \( x_2 \) or not to ... to \( \text{return } [] = [[]] \)

\[
\text{Prelude}\> \text{filterM} \ (\_ \rightarrow [\text{True, False}]) \ "abc"
\["abc","ab","ac","a","bc","b","c","""]
\]
Functions as Monads

Much like functions are applicative functors, functions are Monads that apply the same argument to all their constituent functions.

```haskell
instance Monad ((->) r) where
  return x = \_ -> x -- Just produce x
  h >>= f = \w -> f (h w) w -- Apply w to h and f

import Data.Char

isIDChar :: Char -> Bool -- ((->) Char) is the Monad
isIDChar = do
  l <- isLetter -- The Char argument
  n <- isDigit -- is applied to
  underscore <- (=='_') -- all three of these functions
  return $ l || n || underscore -- before their results are ORed

*Main> map isIDChar "12 aB_"
[True,True,False,True,True,True,True]
```
The State Monad: Modeling Computations with Side-Effects

The Writer Monad can only add to a state, not observe it. The State Monad addresses this by passing a state to each operation. In Control.Monad.State,

```haskell
newtype State s a = State { runState :: s -> (a, s) }

instance Monad (State s) where
  return x = State $ \s -> (x, s)
  State h >>= f = State $ \s -> let (a, s') = h s  -- First step
                             State g = f a             -- Pass result
                             in g s'                    -- Second step

get      = State $ \s -> (s, s)  -- Make the state the result
put s    = State $ _  -> ((), s) -- Set the state
modify f = State $ \s -> ((), f s) -- Apply a state update function
```

State **is not a state**; it more resembles a state machine’s **next state function**

a is the return value       s is actually a state
Example: An Interpreter for a Simple Imperative Language

```haskell
import qualified Data.Map as Map

type Store = Map.Map String Int  -- Value of each variable

-- Representation of a program (an AST)

data Expr = Lit Int  -- Numeric literal: 42
          | Add Expr Expr  -- Addition: 1 + 3
          | Var String     -- Variable reference: a
          | Asn String Expr -- Variable assignment: a = 3 + 1
          | Seq [Expr]     -- Sequence of expressions: a = 3; b = 4;

p :: Expr

p = Seq [ Asn "a" (Lit 3)  -- a = 3;
           , Asn "b" (Add (Var "a") (Lit 1))  -- b = a + 1;
           , Add (Add (Var "a") bpp)           -- a + (b = b + 1) + b;
           (Var "b") ]

where bpp = Asn "b" (Add (Var "b") (Lit 1))
```

Example program:
Example: The Eval Function Taking a Store

\[
\begin{align*}
\text{eval} & : \text{Expr} \to \text{Store} \to (\text{Int}, \text{Store}) \\
\text{eval} (\text{Lit } n) & \quad s = (n, s) \quad -- \text{Store unchanged} \\
\text{eval} (\text{Add } e_1 e_2) & \quad s = \begin{cases} 
\text{let } (n_1, s') = \text{eval } e_1 s \\
(n_2, s'') = \text{eval } e_2 s' 
\end{cases} \quad -- \text{Sees eval } e_1 \\
& \quad \text{in } (n_1 + n_2, s'') \quad -- \text{Sees eval } e_2 \\
\text{eval} (\text{Var } v) & \quad s = \begin{cases} 
\text{case } \text{Map.lookup } v s \text{ of} \\
\text{Just } n & \to (n, s) \\
\text{Nothing} & \to \text{error }$ v ++ " undefined"
\end{cases} \\
\text{eval} (\text{Asn } v e) & \quad s = \begin{cases} 
\text{let } (n, s') = \text{eval } e s \\
\text{in } (n, \text{Map.insert } v n s') \quad -- \text{Sees eval } e
\end{cases} \\
\text{eval} (\text{Seq } es) & \quad s = \text{foldl } (\_\_ s s) \to \text{eval } e s s) (0, s) es
\end{align*}
\]

The fussy part here is “threading” the state through the computations
Example: The Eval Function in Uncurried Form

\[
\text{eval :: \text{Expr} \rightarrow \text{Store} \rightarrow (\text{Int}, \text{Store})}
\]

\[
\text{eval (Lit n) = \lambda s \rightarrow (n, s)} \quad \text{-- Store unchanged}
\]

\[
\text{eval (Add e1 e2) = \lambda s \rightarrow \text{let} \ (n1, s') = \text{eval e1 } s
\]

\[
\text{\hspace{1cm} (n2, s'') = \text{eval e2 } s'
\]

\[
\text{\hspace{2cm} in (n1 + n2, s'')} \quad \text{-- Sees eval e1}
\]

\[
\text{\hspace{2cm} Sees eval e2}
\]

\[
\text{eval (Var v) = \lambda s \rightarrow }
\]

\[
\text{\hspace{1cm} case Map.lookup v s of}
\]

\[
\text{\hspace{2cm} Just n \rightarrow (n, s)} \quad \text{-- Look up v}
\]

\[
\text{\hspace{2cm} Nothing \rightarrow \text{error } $v ++ " undefined"}$
\]

\[
\text{eval (Asn v e) = \lambda s \rightarrow \text{let} \ (n, s') = \text{eval e } s
\]

\[
\text{\hspace{2cm} in (n, Map.insert v n s')} \quad \text{-- Sees eval e}
\]

\[
\text{eval (Seq es) = \lambda s \rightarrow \text{foldl} (\lambda(_, ss) e \rightarrow \text{eval e ss}) (0, s) es}
\]

The parentheses around \text{Store -> (Int, Store)} are unnecessary.
Example: The Eval Function Using the State Monad

$$\text{eval} :: \text{Expr} \rightarrow \text{State Store Int}$$

$$\text{eval} (\text{Lit n}) = \text{return } n \quad \text{-- Store unchanged}$$

$$\text{eval} (\text{Add e1 e2}) = \text{do } n1 <- \text{eval e1}$$
$$\quad \quad \quad n2 <- \text{eval e2} \quad \text{-- Sees eval e1}$$
$$\quad \quad \quad \text{return } \$ n1 + n2 \quad \text{-- Sees eval e2}$$

$$\text{eval} (\text{Var v}) = \text{do } s <- \text{get}$$
$$\quad \quad \quad \text{case Map.lookup v s of} \quad \text{-- Get the store}$$
$$\quad \quad \quad \quad \text{Just n } \rightarrow \text{return } n \quad \text{-- Look up v}$$
$$\quad \quad \quad \quad \text{Nothing } \rightarrow \text{error } \$ v ++ " undefined"$$

$$\text{eval} (\text{Asn v e}) = \text{do } n <- \text{eval e}$$
$$\quad \quad \quad \text{modify } \$ \text{Map.insert v n} \quad \text{-- Sees eval e}$$
$$\quad \quad \quad \text{return } n \quad \text{-- Assigned value}$$

$$\text{eval} (\text{Seq es}) = \text{foldM } (\_ e \rightarrow \text{eval e}) 0 \text{ es} \quad \text{-- Ignore value}$$

The >>= operator threads the state through the computation
The Eval Function in Action: runState, evalState, and execState

\[
\begin{align*}
a &= 3; \\
b &= a + 1; \\
a + (b = b + 1) + b
\end{align*}
\]

*Main> :t runState (eval p) Map.empty
runState (eval p) Map.empty :: (Int, Store)  -- (Result, State)

*Main> :t evalState (eval p) Map.empty
evalState (eval p) Map.empty :: Int  -- Result only
*Main> evalState (eval p) Map.empty
13

*Main> :t execState (eval p) Map.empty
execState (eval p) Map.empty :: Store  -- State only
*Main> Map.toList $ execState (eval p) Map.empty
["a",3,"b",5]
Harnessing Monads

```haskell
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show

A function that works in a Monad can harness any Monad:

```haskell
mapTreeM :: Monad m => (a -> m b) -> Tree a -> m (Tree b)
mapTreeM f (Leaf x) = do x' <- f x
  return $ Leaf x'
mapTreeM f (Branch l r) = do l' <- mapTreeM f l
  r' <- mapTreeM f r
  return $ Branch l' r'
```

```haskell
toList :: Tree a -> [a]
toList t = execWriter $ mapTreeM (\x -> tell [x]) t -- Log each leaf
```

```haskell
foldTree :: (a -> b -> b) -> b -> Tree a -> b
foldTree f s0 t = execState (mapTreeM (\x -> modify (f x)) t) s0
```

```haskell
sumTree :: Num a => Tree a -> a
sumTree t = foldTree (+) 0 t -- Accumulate values using stateful fold
```
Harnessing Monads

*Main> simpleTree = Branch (Leaf (1 :: Int)) (Leaf 2)
*Main> toList simpleTree
[1,2]
*Main> sumTree simpleTree
3
*Main> mapTreeM (\x -> Just (x + 10)) simpleTree
Just (Branch (Leaf 11) (Leaf 12))
*Main> mapTreeM print simpleTree
1
2
*Main> mapTreeM (\x -> [x, x+10]) simpleTree
[Branch (Leaf 1) (Leaf 2),
 Branch (Leaf 1) (Leaf 12),
 Branch (Leaf 11) (Leaf 2),
 Branch (Leaf 11) (Leaf 12)]