Monads

Stephen A. Edwards

Columbia University

Fall 2023
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,

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

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.,

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

Prelude> import qualified Data.Map.Strict as Map
Prelude Map> myMap = Map.fromList ["One","Two"],("Two","Three"),
Prelude Map| ("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

```
infixl 1 >>=
class Applicative m => Monad m where
    (>>=) :: m a -> (a -> m b) -> m b -- "Bind"
    return :: a -> m a -- Wrap a result in the Monad
```

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

```
pure :: b -> f b -- Put value in context
fmap :: (a -> b) -> f a -> f b -- Apply function in context
(<*)) :: f (a -> b) -> f a -> f b -- Function itself is in context
">>=" :: (a -> f b) -> f a -> f b -- 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 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> :t it
it :: Maybe Char

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

```haskell
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
  Prelude| Right "Hello World"

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

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

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

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

\[
[(x,c) | x <- [1,2], c <- ['a','b'] ]
\]

each produce

\[
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
\]
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

Prelude Control.Monad> guard True :: []
[()]
Prelude Control.Monad> guard False :: []
[]
Prelude Control.Monad> guard True :: Maybe ()
Just ()
Prelude Control.Monad> guard False :: Maybe ()
Nothing
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 
\[
[1..50] \gg= \ \backslash x \rightarrow \\
guard \ (x \ `rem` 7 == 0) \gg \quad \text{-- Discard any returned ()} \\
return \ x \\
\]

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

\[ [ x \mid x \leftarrow [1..50], x \ `rem` 7 == 0 ] \]

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
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
import Control.Monad.Writer

logGCD :: Int -> Int -> Writer [String] Int
logGCD a b = do
    tell ["logGCD " ++ show a ++ " " ++ show b]
    if a == b then 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
```

```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
```
Control.Monad.{liftM, ap}: Monads as Functors

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

\[
(<*>) :: \text{Applicative } f \Rightarrow f (a \rightarrow b) \rightarrow f a \rightarrow f b \quad -- \text{“apply”}
\]

In Monad-land, these have alternative names

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

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

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

\[
liftM f m = do \ x <- m \quad -- \text{Get the argument from inside } m
\]

\[
\quad \text{return } (f \ x) \quad -- \text{Apply the argument to the function}
\]

\[
ap mf m = do \ f <- mf \quad -- \text{Get the function from inside } mf
\]

\[
\quad x <- m \quad -- \text{Get the argument from inside } m
\]

\[
\quad \text{return } (f \ x) \quad -- \text{Apply the argument to the function}
\]

Operations in a do block are ordered: ap evaluates its arguments left-to-right
**liftM and ap In Action**

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

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

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

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

Prelude> :set prompt "\n\n> :set prompt-cont "| "
> import Control.Monad.Writer
> :{ runWriter $ ap (writer ((+10), "first") ) (writer (42, "second")) 
> :}
(52, ["first", "second"])

(52, ["first", "second"])
Lots of Lifting: Applying two- and three-argument functions

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

```haskell
liftA2 ::
    Applicative f => (a -> b -> c) -> f a -> f b -> f c

liftA3 ::
    Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d
```

In Control.Monad,

```haskell
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c

liftM3 :: Monad m => (a -> b -> c -> d) -> m a -> m b -> m c -> m d
```

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

```haskell
Prelude Control.Monad> liftM2 (,) (Just 'a') (Just 'b')
Just ('a','b')
Prelude Control.Monad> liftM2 (,) Nothing (Just 'b')
Nothing
```
join is boring on a Monad like Maybe, where it merely strips off a “Just”

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

"Apply f to every object in m 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

```
sequence :: [m a] -> m [a]
sequence_ :: [m a] -> m ()
```

```
Prelude> sequence [print 1, print 2, print 3]
1
2
3
[(),(),()]
Prelude> sequence_ [putStrLn "Hello", putStrLn "World"]
Hello
World
```

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

```haskell
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()  -- Discard result
```

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

```haskell
p10 x = writer (x+10, ["saw "] ++ show x) :: Writer [String] Int
runWriter $ 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_`:

```haskell
mapM_ print ([1..3] :: [Int])
1
2
3
```

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

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

In foldM, the folding function operates and returns a result in a Monad:

foldM :: Monad m => (a -> b -> m a) -> a -> [b] -> m a

foldM f a1 [x1, x2, ..., xm] = do a2 <- f a1 x1
    a3 <- f a2 x2
    ...
    f a m x m

Example: Sum a list of numbers and report progress

> runWriter $ foldM (\a x -> 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”

\a x -> writer (a+x, [(x,a)])
Control.Monad.filterM: Filter a List in Monad-land

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

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

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

> 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:

\[
\begin{align*}
\text{start with } x_1 & \quad \text{do not start with } x_1 \\
&
\left[\begin{array}{c}
[x_1, x_2, \ldots], \ldots, [x_1], [x_2, x_3, \ldots], \ldots, []
\end{array}\right]
\end{align*}
\]

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

*Main> powerset "abc"
["abc","ab","ac","a","bc","b","c",""]
Let’s perform this step (i.e., possibly prepending \(x\) and combining) using the list Monad. Recall \texttt{liftM2} applies Monadic arguments to a two-input function:

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

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

\[
\texttt{listM2 :: (Bool -> [Char] -> [Char]) -> [Bool] -> [[Char]] -> [[Char]]}
\]

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

\texttt{liftM2} makes the function “nondeterministic” by applying the function with every \texttt{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

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

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

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

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

Prelude> \( \text{filterM } (\_ \rightarrow [\text{True}, \text{False}]) \) "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 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,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,

\[
\text{newtype State } s \ a = \text{State} \{ \text{runState} :: s \rightarrow (a, s) \}
\]

\[
\text{instance Monad (State } s) \text{ where}
\]

\[
\text{return } x = \text{State } \_ \rightarrow (x, s)
\]

\[
\text{State } h \gg= f = \text{State } \_ \rightarrow \text{let } (a, s') = h s \text{ -- First step}
\]

\[
\text{State } g = f a \text{ -- Pass result}
\]

\[
\text{in } g s' \text{ -- Second step}
\]

\[
\text{get } = \text{State } \_ \rightarrow (s, s) \text{ -- Make the state the result}
\]

\[
\text{put } s = \text{State } \_ \rightarrow () \rightarrow ((), s) \text{ -- Set the state}
\]

\[
\text{modify } f = \text{State } \_ \rightarrow () \rightarrow ((), f s) \text{ -- Apply a state update function}
\]

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

a is the return value \hspace{1em} s is actually a state
import qualified Data.Map as Map

-- 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: The Eval Function Taking a Store

```haskell
eval :: Expr -> Store -> (Int, Store)  
eval (Lit n) s = (n, s)  -- Store unchanged  
eval (Add e1 e2) s = let (n1, s') = eval e1 s 
                        (n2, s'') = eval e2 s'  -- Sees eval e1 
                        in (n1 + n2, s'')  -- Sees eval e2  
eval (Var v) s = case Map.lookup v s of  -- Look up v 
                Just n -> (n, s)  
                Nothing -> error $ v ++ " undefined"  
eval (Asn v e) s = let (n, s') = eval e s 
                     in (n, Map.insert v n s')  -- Sees eval e  
eval (Seq es) s = foldl (\(_, ss) e -> eval e ss) (0, s) es
```

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

```haskell
eval :: Expr -> (Store -> (Int, Store))
eval (Lit n) = \s -> (n, s)  -- Store unchanged
eval (Add e1 e2) = \s -> let (n1, s') = eval e1 s
                       (n2, s'') = eval e2 s'
                       in (n1 + n2, s'')  -- Sees eval e1
                       -- Sees eval e2

eval (Var v) = \s ->
               case Map.lookup v s of
               Just n -> (n, s)
               Nothing -> error $ v ++ " undefined"

eval (Asn v e) = \s -> let (n, s') = eval e s
                   in (n, Map.insert v n s')  -- Sees eval e

eval (Seq es) = \s -> foldl (\(_, ss) e -> eval e ss) (0, s) es
```

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

eval :: Expr -> State Store Int

eval (Lit n) = return n  -- Store unchanged

eval (Add e1 e2) = do n1 <- eval e1
                       n2 <- eval e2  -- Sees eval e1
                       return $ n1 + n2  -- Sees eval e2

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

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

eval (Seq es) = foldM (\_ e -> eval e) 0 es  -- Ignore value

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

```haskell
a = 3;
b = a + 1;
a + (b = b + 1) + b

*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)]