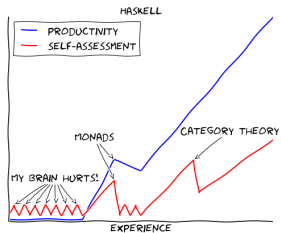


Monads and IO

Stephen A. Edwards

Columbia University

Fall 2026



Adding Tracing to the Calculator

The Monad Typeclass

I/O with the IO Monad

- putStrLn

- getLine

Monads Everywhere

- Maybe and Either as Monads

- Monad Laws

- The List Monad

- List Comprehensions as a Monad

Monads, Functors, and Applicative Functors

Monoids

The MonadPlus Type Class and guard

The Writer Monad

The State Monad



A Purely Functional Language

“All computation is the evaluation of mathematical functions”

⇒ referential transparency: if $f(x) = 3$, $f(x)$ can be replaced by 3

⇒ no (side-)effects: no assignments, no I/O

⇒ all dataflow is explicit

Recall the Calculator

```
data Op = Add | Sub | Mul
data Expr = BinOp Expr Op Expr | Neg Expr | Lit Int

eval :: Expr -> Int
eval (Lit n) = n
eval (Neg e) = negate $ eval e
eval (BinOp e1 op e2) = let
    e1' = eval e1
    e2' = eval e2 in
    case op of Add -> e1' + e2'
              Sub -> e1' - e2'
              Mul -> e1' * e2'
```

Let's Add a Tracing Facility

```
eval :: Expr -> (Int, [String])
```

-- Trace is a list of strings

Let's Add a Tracing Facility

```
eval :: Expr -> (Int, [String])      -- Trace is a list of strings  
eval (Lit n) = (n, ["Lit " ++ show n]) -- Base case: report literal
```

```
ghci> eval 3 -- Still using the instance Num Expr trick  
(3,["Lit 3"])
```

Let's Add a Tracing Facility

```
eval :: Expr -> (Int, [String])      -- Trace is a list of strings
eval (Lit n) = (n, ["Lit " ++ show n]) -- Base case: report literal
eval (Neg e) = let (e', t) = eval e in -- Recurse and get trace
  (negate e', t ++ ["Neg " ++ show e']) -- Return result; extend trace
```

```
ghci> eval (-(-3))
(3,["Lit 3","Neg 3","Neg -3"])
```

Let's Add a Tracing Facility

```
eval :: Expr -> (Int, [String])      -- Trace is a list of strings
eval (Lit n) = (n, ["Lit " ++ show n]) -- Base case: report literal
eval (Neg e) = let (e', t) = eval e in -- Recurse and get trace
  (negate e', t ++ ["Neg " ++ show e']) -- Return result; extend trace
eval (BinOp e1 op e2) = let
  (e1', t1) = eval e1                -- Recurse left
  (e2', t2) = eval e2 in             -- Recurse right
  (case op of Add -> e1' + e2'       -- Calculate result
           Sub -> e1' - e2'
           Mul -> e1' * e2',
   t1 ++ t2 ++ [show op ++ " " ++ show e1' ++ " " ++ show e2'])
```

```
ghci> eval $ (-2) * 3 + 4 * 5
(14,["Lit 2","Neg 2","Lit 3","Mul -2 3",
     "Lit 4","Lit 5","Mul 4 5","Add -6 20"])
```

```
type Traced a = (a, [String])
```

A value that's carrying a trace (list of strings) along with it

```
type Traced a = (a, [String])
```

```
andThen :: Traced a  
        -> (a -> Traced b)  
        -> Traced b
```

A value that's carrying a trace (list of strings) along with it

“Extract the value from a traced value; apply it to a function that produces a new traced value; and glue the two traces together”

```
type Traced a = (a, [String])
```

```
andThen :: Traced a  
        -> (a -> Traced b)  
        -> Traced b
```

```
andThen x f =  
  let (x1, t1) = x    in  
  let (x2, t2) = f x1 in  
  (x2, t1 ++ t2)
```

A value that's carrying a trace (list of strings) along with it

“Extract the value from a traced value; apply it to a function that produces a new traced value; and glue the two traces together”

```
type Traced a = (a, [String])
```

```
andThen :: Traced a  
        -> (a -> Traced b)  
        -> Traced b
```

```
andThen x f =  
    let (x1, t1) = x    in  
    let (x2, t2) = f x1 in  
    (x2, t1 ++ t2)
```

```
treturn :: a -> Traced a  
treturn x = (x, [])
```

A value that's carrying a trace (list of strings) along with it

“Extract the value from a traced value; apply it to a function that produces a new traced value; and glue the two traces together”

Promote an ordinary result into a traced result

```
type Traced a = (a, [String])
```

```
andThen :: Traced a  
        -> (a -> Traced b)  
        -> Traced b
```

```
andThen x f =  
    let (x1, t1) = x    in  
    let (x2, t2) = f x1 in  
    (x2, t1 ++ t2)
```

```
treturn :: a -> Traced a  
treturn x = (x, [])
```

```
trace :: String -> Traced ()  
trace t = ((), [t])
```

A value that's carrying a trace (list of strings) along with it

“Extract the value from a traced value; apply it to a function that produces a new traced value; and glue the two traces together”

Promote an ordinary result into a traced result

Prepare to append a new string to a trace

```
type Traced a = (a, [String])
```

```
andThen :: Traced a  
        -> (a -> Traced b)  
        -> Traced b
```

```
andThen x f =  
  let (x1, t1) = x    in  
  let (x2, t2) = f x1 in  
  (x2, t1 ++ t2)
```

```
treturn :: a -> Traced a  
treturn x = (x, [])
```

```
trace :: String -> Traced ()  
trace t = ((), [t])
```

```
eval :: Expr -> Traced Int
```

```
type Traced a = (a, [String])

andThen :: Traced a
        -> (a -> Traced b)
        -> Traced b

andThen x f =
    let (x1, t1) = x    in
    let (x2, t2) = f x1 in
    (x2, t1 ++ t2)

treturn :: a -> Traced a
treturn x = (x, [])

trace :: String -> Traced ()
trace t = ((), [t])
```

```
eval :: Expr -> Traced Int

eval (Lit n) =
    trace ("Lit " ++ show n) `andThen`
    \_ -> treturn n
```

```
type Traced a = (a, [String])

andThen :: Traced a
         -> (a -> Traced b)
         -> Traced b

andThen x f =
  let (x1, t1) = x    in
  let (x2, t2) = f x1 in
  (x2, t1 ++ t2)

treturn :: a -> Traced a
treturn x = (x, [])

trace :: String -> Traced ()
trace t = ((), [t])
```

```
eval :: Expr -> Traced Int

eval (Lit n) =
  trace ("Lit " ++ show n) `andThen`
  \_ -> treturn n

eval (Neg e) = eval e `andThen`
  \e' -> trace ("Neg " ++ show e')
  `andThen` \_ -> treturn $ negate e'
```

```
type Traced a = (a, [String])

andThen :: Traced a
        -> (a -> Traced b)
        -> Traced b

andThen x f =
    let (x1, t1) = x    in
    let (x2, t2) = f x1 in
    (x2, t1 ++ t2)

treturn :: a -> Traced a
treturn x = (x, [])

trace :: String -> Traced ()
trace t = ((), [t])
```

```
eval :: Expr -> Traced Int

eval (Lit n) =
    trace ("Lit " ++ show n) `andThen`
    \_ -> treturn n

eval (Neg e) = eval e `andThen`
    \e' -> trace ("Neg " ++ show e')
    `andThen` \_ -> treturn $ negate e'

eval (BinOp e1 op e2) =
    eval e1 `andThen` \e1' ->
    eval e2 `andThen` \e2' ->
    trace (show op ++ " " ++ show e1' ++
           " " ++ show e2') `andThen` \_ ->
    treturn $ case op of Add -> e1' + e2'
                Sub -> e1' - e2'
                Mul -> e1' * e2'
```

```
infixl 1 >>= -- Low precedence  
           -- a >>= b >>= c means a >>= (b >>= c)  
  
class Monad m where  
    (>>=) :: m a -> (a -> m b) -> m b -- "Bind"  
    return :: a -> m a
```

A *Monad* is a group of types with a return function that wraps a value in the monad and a bind operator that applies a monadic function to a monadic value

Haskell

```
infixl 1 >>=  -- Low precedence
                -- a >>= b >>= c  means  a >>= (b >>= c)

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

A *Monad* is a group of types with a return function that wraps a value in the monad and a bind operator that applies a monadic function to a monadic value

"An operator so good, they used it as the logo"

```
type Traced a = (a, [String])
```

```
andThen :: Traced a  
        -> (a -> Traced b)  
        -> Traced b
```

```
andThen x f =  
    let (x', t1) = x    in  
    let (x'', t2) = f x' in  
    (x'', t1 ++ t2)
```

```
treturn :: a -> Traced a  
treturn x = (x, [])
```

```
trace :: String -> Traced ()  
trace t = ((), [t])
```

```
eval :: Expr -> Traced Int
```

```
eval (Lit n) =  
    trace ("Lit " ++ show n) `andThen`  
    \_ -> treturn n
```

```
eval (Neg e) =  
    eval e `andThen`  
    \e' -> trace ("Neg " ++ show e')  
    `andThen` \_ -> treturn $ negate e'
```

```
eval (BinOp e1 op e2) =  
    eval e1 `andThen` \e1' ->  
    eval e2 `andThen` \e2' ->  
    trace (show op ++ " " ++ show e1' ++  
           " " ++ show e2') `andThen` \_ ->  
    treturn $ case op of Add -> e1' + e2'  
                    Sub -> e1' - e2'  
                    Mul -> e1' * e2'
```

```
newtype Traced a =  
    Tr (a, [String])  
deriving Show  
  
instance Monad Traced where  
    x >>= f =  
        let Tr (x', t1) = x in  
        let Tr (x'', t2) = f x' in  
        Tr (x'', t1 ++ t2)  
  
    return x = Tr (x, [])  
  
trace :: String -> Traced ()  
trace t = Tr ((), [t])
```

```
eval :: Expr -> Traced Int  
eval (Lit n) =  
    trace ("Lit " ++ show n) >>=  
    \_ -> return n  
eval (Neg e) =  
    eval e >>=  
    \e' -> trace ("Neg " ++ show e')  
    >>= \_ -> return $ negate e'  
eval (BinOp e1 op e2) =  
    eval e1 >>= \e1' ->  
    eval e2 >>= \e2' ->  
    trace (show op ++ " " ++ show e1' ++  
           " " ++ show e2') >>= \_ ->  
    return $ case op of Add -> e1' + e2'  
                    Sub -> e1' - e2'  
                    Mul -> e1' * e2'
```

```
newtype Traced a =  
  Tr (a, [String])  
deriving Show
```

```
instance Monad Traced where  
  x >>= f =  
    let Tr (x', t1) = x    in  
    let Tr (x'', t2) = f x' in  
    Tr (x'', t1 ++ t2)
```

```
return x = Tr (x, [])
```

```
trace :: String -> Traced ()  
trace t = Tr ((), [t])
```

```
eval :: Expr -> Traced Int
```

```
eval (Lit n) = do  
  trace ("Lit " ++ show n)  
  return n
```

```
eval (Neg e) = do  
  e' <- eval e  
  trace ("Neg " ++ show e')  
  return $ negate e'
```

```
eval (BinOp e1 op e2) = do  
  e1' <- eval e1  
  e2' <- eval e2  
  trace (show op ++ " " ++ show e1' ++  
    " " ++ show e2')  
  return $ case op of Add -> e1' + e2'  
                  Sub -> e1' - e2'  
                  Mul -> e1' * e2'
```



I/O



I/O in Haskell Uses the IO Monad

```
ghci> putStrLn "Hello World"
```

```
Hello World
```

```
ghci> :t putStrLn
```

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

```
ghci> :i IO
```

```
type IO :: * -> *
```

```
instance Monad IO
```

I/O in Haskell Uses the IO Monad

```
ghci> putStrLn "Hello World"
Hello World
```

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

```
ghci> :i IO
type IO :: * -> *
instance Monad IO
```

```
ghci> putStrLn "Hello" >=> \_ -> putStrLn "World"
Hello
World
```

I/O in Haskell Uses the IO Monad

```
ghci> putStrLn "Hello World"
Hello World
```

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

```
ghci> :i IO
type IO :: * -> *
instance Monad IO
```

```
ghci> :{
ghci| do putStrLn "Hello"
ghci|     putStrLn "World"
ghci| :}
Hello
World
```

Easy to change from tracing to printing

```
eval :: Expr -> Traced Int
eval (Lit n) = do
    trace ("Lit " ++ show n)
    return n
eval (Neg e) = do
    e' <- eval e
    trace ("Neg " ++ show e')
    return $ negate e'
eval (BinOp e1 op e2) = do
    e1' <- eval e1
    e2' <- eval e2
    trace $ show op ++ " " ++ show e1' ++ " " ++ show e2'
    return $ case op of
        Add -> e1' + e2'
        Sub -> e1' - e2'
        Mul -> e1' * e2'
```

Easy to change from tracing to printing

```
eval :: Expr -> IO Int
eval (Lit n) = do
    putStrLn ("Lit " ++ show n)
    return n
eval (Neg e) = do
    e' <- eval e
    putStrLn ("Neg " ++ show e')
    return $ negate e'
eval (BinOp e1 op e2) = do
    e1' <- eval e1
    e2' <- eval e2
    putStrLn $ show op ++ " " ++ show e1' ++ " " ++ show e2'
    return $ case op of
        Add -> e1' + e2'
        Sub -> e1' - e2'
        Mul -> e1' * e2'
```

Easy to change from tracing to printing

```
eval :: Expr -> IO Int
eval (Lit n) = do
    putStrLn ("Lit " ++ show n)
    return n
eval (Neg e) = do
    e' <- eval e
    putStrLn ("Neg " ++ show e')
    return $ negate e'
eval (BinOp e1 op e2) = do
    e1' <- eval e1
    e2' <- eval e2
    putStrLn $ show op ++ " " ++ show e1' ++ " " ++ show e2'
    return $ case op of
        Add -> e1' + e2'
        Sub -> e1' - e2'
        Mul -> e1' * e2'
```

```
ghci> eval $ (-2) * 3 + 4 * 5
Lit 2
Neg 2
Lit 3
Mul -2 3
Lit 4
Lit 5
Mul 4 5
Add -6 20
14
```

The IO Monad provides input, too

```
ghci> :t getLine
getLine :: IO String
```

hello2.hs:

```
main :: IO ()
main = do
    putStrLn "Hello. What is your name?" -- Print the string
    name <- getLine                       -- Read a line; bind result to name
    putStrLn $ "Hello, " ++ name
```

```
$ stack runhaskell hello2
Hello. What is your name?
Stephen
Hello, Stephen
```

let blocks may also appear in do blocks

let1.hs:

```
import Data.Char(toUpper)  -- Get the toUpper function from Data.Char

main = do                  -- The three kinds of syntax for do block statements:
  putStr "First Name? "    -- 1/3: expr
  fname <- getLine          -- 2/3: name <- expr
  putStr "Last Name? "
  lname <- getLine
  let fshout = map toUpper fname  -- 3/3: let decls
      lshout = map toUpper lname  -- in not used in do blocks
  putStrLn $ "WELCOME " ++ fshout ++ " " ++ lshout
```

```
$ stack runhaskell let1
First Name? Stephen
Last Name? Edwards
WELCOME STEPHEN EDWARDS
```

Word Reverser Program → droW resreveR margorP

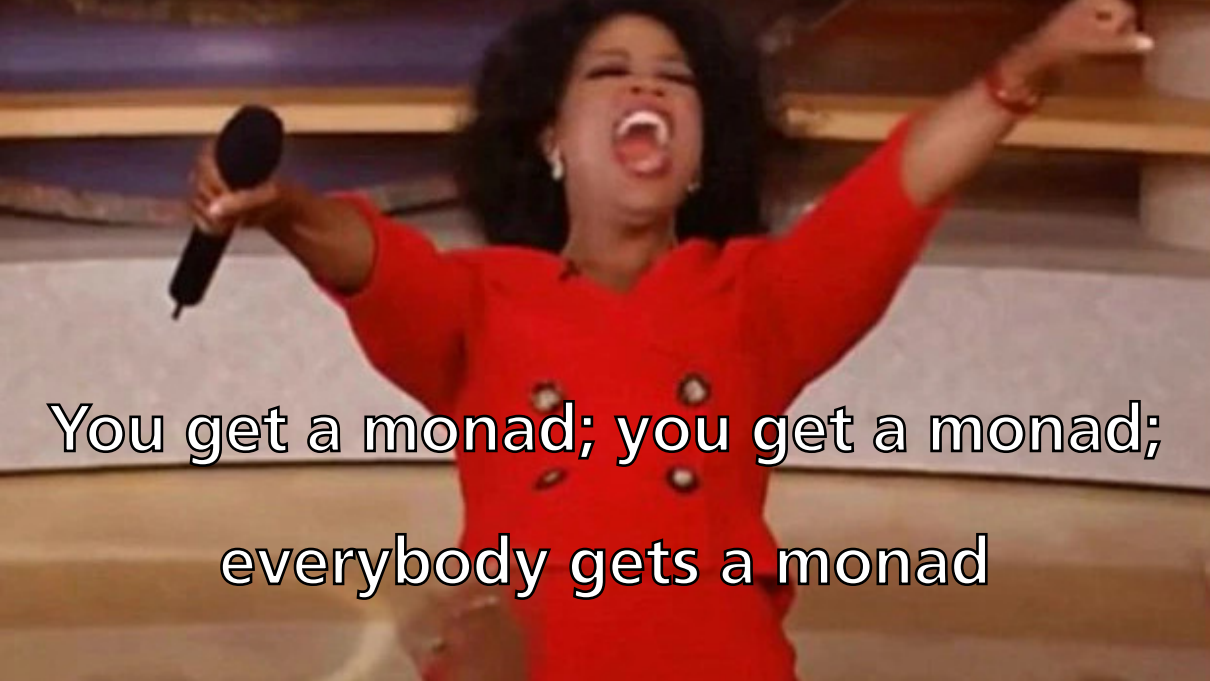
reverser.hs:

```
reverseWords :: String -> String
reverseWords = unwords . map reverse . words

main = do
  line <- getLine
  if null line then      -- if-then-else is an expression, so both
    return ()           -- branches must return the same thing but
  else do                -- return doesn't do quite what you think
    putStrLn $ reverseWords line
  main
```

```
$ stack runhaskell reverser
able elba stressed diaper looter debut deeps devil peels
elba able desserts repaid retool tubed speed lived sleep
tacocat deified civic radar rotor kayak aibohphobia
tacocat deified civic radar rotor kayak aibohphobia
```

Aibohphobia: Fear of palindromes



You get a monad; you get a monad;
everybody gets a monad

Maybe is a Monad: **Nothing** indicates failure

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

instance Monad Maybe where
  Just x  >>= f  = f x      -- Standard Prelude definition
  Nothing >>= _   = Nothing  -- Normal computation
                                -- Computation failed; stop

  return x = Just x        -- Wrap in a Just
```

The Maybe Monad in Action

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

```
ghci> return "what?" :: Maybe String  
Just "what?"
```

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

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

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

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

Either is also a Monad, similar to Maybe

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

```
instance Monad (Either e) where
```

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

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

```
  return x      = Right x
```

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

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

Monad Laws

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

```
return x >>= f    =    f x
```

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

```
m >>= return      =    m
```

Associative: applying `g` after applying `f` is like applying `f` composed with `g`

```
(m >>= f) >>= g    =    m >>= (\x -> f x >>= g)
```

The List Monad: "Nondeterministic Computation"

Intuition: lists represent all possible results

```
instance Monad [] where
```

```
  xs >>= f = concat (map f xs)  -- Collect all possible results from f
```

```
  return x = [x]                -- Exactly one result
```

```
ghci> [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 and List Comprehensions

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

```
ghci> [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] >>= \x -> ['a','b'] >>= \c -> return (x,c)
```

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

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

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

each produce

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

Monads are Functors

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

```
ghci> fmap (+1) (Just 41)  
Just 42  
ghci> fmap (+1) [10,100,41]  
[11,101,42]
```

fmap ("apply a function to arguments in a box") is called **liftM** in Monad-land:

```
liftM :: Monad m => (a -> b) -> m a -> m b  
liftM f m = do x <- m           -- Extract the argument  
              return (f x)      -- Apply f and wrap the result
```

```
ghci> Control.Monad.liftM (+1) [10,100,41]  
[11,101,42]
```

Applicative Functors: Putting Functions in a Box

```
infixl 4 <*>
```

```
class Functor f => Applicative f where
```

```
  pure  :: a -> f a
```

— Box something, e.g., a function

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

— Apply boxed function to a box

```
instance Applicative Maybe where
```

```
  pure = Just
```

— Put it in a “Just” box

```
  Nothing <*> _ = Nothing
```

— No function to apply

```
  Just f <*> m = fmap f m
```

— Apply function-in-a-box f

```
ghci> :t fmap (+) (Just 1)
```

```
fmap (+) (Just 1) :: Num a => Maybe (a -> a) — Function-in-a-box
```

```
ghci> fmap (+) (Just 1) <*> (Just 2)
```

```
Just 3
```

```
ghci> fmap (+) Nothing <*> (Just 2)
```

```
Nothing
```

— Nothing is a buzzkiller

Monads are Applicative Functors

```
infixl 4 <*>
```

```
class Functor f => Applicative f where
```

```
  pure  :: a -> f a           -- Box something, e.g., a function
```

```
  (<*>) :: f (a -> b) -> f a -> f b -- Apply boxed function to a box
```

In Applicative Functor-land, `<$>` is `fmap`. In Monad-land; `pure` is `return` and `<*>` (“apply a function in a box to an argument in a box”) is called `ap`

```
ap mf m    = do f <- mf      -- Get the function from inside mf
              x <- m          -- Get the argument from inside m
              return (f x)   -- Apply the argument to the function
```

```
ghci> Control.Monad.ap (return (+1000)) [10,50,100]
[1010,1050,1100]
```

```
ghci> Control.Monad.ap (return (+)) [10,50,100] <*> [0,1000]
[10,1010,50,1050,100,1100]
```

```
ghci> (+) <$> [10,50,100] <*> [0,1000]
[10,1010,50,1050,100,1100]
```

Monoids

Type classes present a common interface to types that behave similarly

A Monoid is a type with an associative binary operator and an identity value

E.g., * and 1 on numbers, ++ and [] on lists:

```
ghci> 4 * 1
4  — 1 is the identity on the right
ghci> 1 * 4
4  — 1 is the identity on the left
ghci> 2 * (3 * 4)
24
ghci> (2 * 3) * 4
24  — * is associative
ghci> 2 * 3
6
ghci> 3 * 2
6  — * happens to be commutative
```

```
ghci> "hello" ++ []
"hello"  — [] is the right identity
ghci> [] ++ "hello"
"hello"  — [] is the left identity
ghci> "a" ++ ("bc" ++ "de")
"abcde"
ghci> ("a" ++ "bc") ++ "de"
"abcde"  — ++ is associative
ghci> "a" ++ "b"
"ab"
ghci> "b" ++ "a"
"ba"  — ++ is not commutative
```

The Monoid Type Class

```
class Monoid m where
  mempty  :: a                -- The identity value
  mappend :: m -> m -> m     -- The associative binary operator

  mconcat :: [m] -> m        -- Apply the binary operator to a list
  mconcat = foldr mappend mempty -- Default implementation
```

Lists are Monoids:

```
instance Monoid [a] where
  mempty  = []
  mappend = (++)
```

```
ghci> mempty :: [a]
[]
ghci> "hello " `mappend` "world!"
"hello world!"
ghci> mconcat ["hello ", "pfp ", "world!"]
"hello pfp world!"
```

*****, 1 and +, 0 Can Each Make a Monoid**

newtype lets us build distinct Monoids for each

In Data.Monoid,

```
newtype Product a = Product { getProduct :: a }  
  deriving (Eq, Ord, Read, Show, Bounded)
```

```
instance Num a => Monoid (Product a) where  
  empty = Product 1  
  Product x `mappend` Product y = Product (x * y)
```

```
newtype Sum a = Sum { getSum :: a }  
  deriving (Eq, Ord, Read, Show, Bounded)
```

```
instance Num a => Monoid (Sum a) where  
  empty = Sum 0  
  Sum x `mappend` Sum y = Sum (x + y)
```

Product and Sum In Action

```
ghci> mempty :: Sum Int
```

```
Sum {getSum = 0}
```

```
ghci> mempty :: Product Int
```

```
Product {getProduct = 1}
```

```
ghci> Sum 3 `mappend` Sum 4
```

```
Sum {getSum = 7}
```

```
ghci> Product 3 `mappend` Product 4
```

```
Product {getProduct = 12}
```

```
ghci> mconcat [Sum 1, Sum 10, Sum 100]
```

```
Sum {getSum = 111}
```

```
ghci> mconcat [Product 10, Product 3, Product 5]
```

```
Product {getProduct = 150}
```

The Any (||, False) and All (&&, True) Monoids

In Data.Monoid,

```
newtype Any = Any { getAny :: Bool }  
    deriving (Eq, Ord, Read, Show, Bounded)
```

```
instance Monoid Any where  
    mempty = Any False  
    Any x `mappend` Any y = Any (x || y)
```

```
newtype All = All { getAll :: Bool }  
    deriving (Eq, Ord, Read, Show, Bounded)
```

```
instance Monoid All where  
    mempty = All True  
    All x `mappend` All y = All (x && y)
```

Any and All

```
ghci> mempty :: Any
Any {getAny = False}
ghci> mempty :: All
All {getAll = True}

ghci> getAny $ Any True `mappend` Any False
True
ghci> getAll $ All True `mappend` All False
False

ghci> mconcat [Any True, Any False, Any True]
Any {getAny = True}
ghci> mconcat [All True, All True, All False]
All {getAll = False}
```

Yes, *any* and *all* are easier to use

Ordering as a Monoid

```
data Ordering = LT | EQ | GT
```

In Data.Monoid,

```
instance Monoid Ordering where
  mempty = EQ
  LT `mappend` _ = LT
  EQ `mappend` y = y
  GT `mappend` _ = GT
```

Application: an *lcomp* for strings ordered by length then alphabetically, e.g.,

```
lcomp :: String -> String -> Ordering
```

```
"b"      `lcomp` "aaaa"    = LT  --- b is shorter
```

```
"bbbbbb" `lcomp` "a"       = GT  --- bbbbbb is longer
```

```
"avenger" `lcomp` "avenged" = LT  --- Same length: r is after d
```

lcomp

```
lcomp :: String -> String -> Ordering
lcomp x y = case length x `compare` length y of
    LT -> LT
    GT -> GT
    EQ -> x `compare` y
```

A little too operational; *mappend* is exactly what we want

```
lcomp :: String -> String -> Ordering
lcomp x y = (length x `compare` length y) `mappend`
    (x `compare` y)
```

Maybe the Monoid

```
instance Monoid a => Monoid (Maybe a) where
  mempty = Nothing
  Nothing `mappend` m      = m
  m      `mappend` Nothing = m
  Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

```
ghci> Nothing `mappend` Just "pfp"
Just "pfp"
```

```
ghci> Just "fun" `mappend` Nothing
Just "fun"
```

```
ghci> :m +Data.Monoid
```

```
ghci> Just (Sum 3) `mappend` Just (Sum 4)
Just (Sum {getSum = 7})
```

```
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
```

```
ghci> guard True :: [()]
[()]
ghci> guard False :: [()]
[]
ghci> guard True :: Maybe ()
Just ()
ghci> guard False :: Maybe ()
Nothing
```

Using Control.Monad.guard as a filter

guard uses 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] >>= \x ->  
  guard (x `rem` 7 == 0) >>  -- Discard any returned ()  
  return x
```

```
do x <- [1..50]  
  guard (x `rem` 7 == 0)      -- No <- makes for an implicit >>  
  return x
```

```
[ x | x <- [1..50], x `rem` 7 == 0 ]
```

each produce

```
[7,14,21,28,35,42,49]
```

The Writer Monad

An implementation of the tracing pattern: the ability to accumulate a result in order while performing computation (e.g., logging, code generation).

Control.Monad.Writer has something like

```
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

tell :: w -> Writer w () -- Log something
tell w = Writer ((), w)
```

runWriter is a trick for extracting the (value, log) pair from a Writer computation

```
import Control.Monad.Writer

eval :: Expr -> Writer [String] Int

eval (Lit n) = do
    tell ["Lit " ++ show n]
    return n

eval (Neg e) = do
    e' <- eval e
    tell ["Neg " ++ show e']
    return $ negate e'

eval (BinOp e1 op e2) = do
    e1' <- eval e1
    e2' <- eval e2
    tell [show op ++ " " ++ show e1' ++
          " " ++ show e2']
    return $ case op of Add -> e1' + e2'
                        Sub -> e1' - e2'
                        Mul -> e1' * e2'
```

```
ghci> runWriter $ eval $
ghci|      (-2) * 3 + 4 * 5
(14,["Lit 2","Neg 2","Lit 3",
    "Mul -2 3","Lit 4",
    "Lit 5","Mul 4 5",
    "Add -6 20"])
```

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

```
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:

```
> 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_:

```
> 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 am xm
```

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 [])
```

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])
```

```
ghci> fst $ runWriter $ filterM isSmall [9,1,5,2,10,3]
[1,2,3]
ghci> 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 $[x_1, x_2, \dots]$, the answer consists of two kinds of lists:

$$\left[\underbrace{[x_1, x_2, \dots], \dots, [x_1]}_{\text{start with } x_1}, \underbrace{[x_2, x_3, \dots], \dots, []}_{\text{do not start with } x_1} \right]$$

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

```
ghci> powerset "abc"  
["abc", "ab", "ac", "a", "bc", "b", "c", ""]
```

The List Monad and Powersets

```
powerset (x:xs) = map (x:) (powerset xs) ++ powerset xs
```

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

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

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

```
listM2 :: (Bool -> [Char] -> [Char]) -> [Bool] -> [[Char]] ->
        [[Char]]
```

```
ghci> liftM2 (\k -> if k then ('a':) else id) [True, False] ["bc", "d"]
["abc", "ad", "bc", "d"]
```

`liftM2` makes the function “nondeterministic” by applying the function with every `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

```
foldr f z [x1,x2,..,xn] = f x1 (f x2 ( ... (f xn z) ... ))

filterM p = foldr (\x -> liftM2 (\k -> if k then (x:)
                                else id) (p x)) (return [])

filterM p [x1,x2,..xn] =
  liftM2 (\k -> if k then (x1:) else id) (p x1)
  (liftM2 (\k -> if k then (x2:) else id) (p x2)
    ..
    (liftM2 (\k -> if k then (xn:) else id) (p xn) (return []))) ..)
```

If we let `p _ = [True, False]`, this chooses to prepend `x1` or not to the result of prepending `x2` or not to ... to return `[] = [[]]`

```
ghci> filterM (\_ -> [True, False]) "abc"
["abc","ab","ac","a","bc","b","c",""]
```

Adding side-effects to our calculator

```
data Op = Add | Sub | Mul
  deriving Show

data Expr = BinOp Expr Op Expr
  | Neg Expr
  | Lit Int
  | Var String
  | Asn String Expr
  | Seq [Expr]
  deriving Show
```

```
infixl 1 #      -- Sequencing
(#) :: Expr -> Expr -> Expr
Seq e1 # e2     = Seq (e1 ++ [e2])
e1      # e2     = Seq ([e1, e2])

infixl 2 <==    -- Assignment
(<==) :: Expr -> Expr -> Expr
(Var v) <== e   = Asn v e
_       <== _   = error "var?"
```

```
ghci> a = Var "a" ; b = Var "b"
ghci> a <== 3      #    b <== a + 1      #    a * (b <== b + 1) + b
Seq [Asn "a" (Lit 3),
     Asn "b" (BinOp (Var "a") Add (Lit 1)),
     BinOp (BinOp (Var "a") Mul
                (Asn "b" (BinOp (Var "b") Add (Lit 1)))) Add (Var "b")]
```

The store and **doop**

We need something to hold the value of each variable.
Simple, inefficient solution: an association list

```
type Store = [(String, Int)]
```

```
ghci> st = [("a",10), ("b",20)] :: Store
ghci> lookup "a" st           -- Fetch a variable's value
Just 10
ghci> st' = ("a", 15) : st -- Update a variable's value
ghci> lookup "a" st'
Just 15
```

Helper function for evaluating operators:

```
doop :: Op -> Int -> Int -> Int
doop Add = (+)
doop Sub = (-)
doop Mul = (*)
```

Implementing eval: threading state

```
eval :: Expr -> Store -> (Int, Store)

eval (Lit n) s =      (n, s)

eval (Neg e) s =      let (e', s') = eval e s
                      in (negate e', s')

eval (BinOp e1 op e2) s =      let (e1', s1) = eval e1 s
                                (e2', s2) = eval e2 s1
                                in (doop op e1' e2', s2)

eval (Var v) s =      case 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, (v, n) : s')

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

Implementing eval: threading state and uncurrying

```
eval :: Expr -> (Store -> (Int, Store)) -- Smells like a Monad

eval (Lit n) = \s -> (n, s)

eval (Neg e) = \s -> let (e', s') = eval e s
                      in (negate e', s')

eval (BinOp e1 op e2) = \s -> let (e1', s1) = eval e1 s
                                (e2', s2) = eval e2 s1
                                in (doop op e1' e2', s2)

eval (Var v) = \s -> case 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, (v, n) : s')

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

The State Monad: Modeling Computations with Side-Effects

Can we make a monad where the result is a `Store -> (Int, Store)` function?
In `Control.Monad.State`:

```
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  -- Run last step
                                State g = f a    -- Prepare next
                                in  g s'         -- Take next 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's like a state machine's **next state function**

a is the return value s is actually a state

Eval

```
eval :: Expr -> (Store -> (Int, Store)) -- Smells like a Monad

eval (Lit n) = \s -> (n, s)

eval (Neg e) = \s -> let (e', s') = eval e s
                      in      (negate e', s')

eval (BinOp e1 op e2) = \s -> let (e1', s1) = eval e1 s
                                (e2', s2) = eval e2 s1
                                in      (doop op e1' e2', s2)

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

eval (Asn v e) = \s -> let (e', s') = eval e s in
                      (e',
                        (v, e') : s')

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

Eval using the State Monad

```
eval :: Expr -> State Store Int

eval (Lit n) = return n

eval (Neg e) =          do e'    <- eval e
                        return (negate e'    )

eval (BinOp e1 op e2) =          do e1'    <- eval e1
                                e2'    <- eval e2
                                return (doop op e1' e2'    )

eval (Var v) = do s <- get
              case lookup v s of
                Just n -> return n
                Nothing -> error $ v ++ " undefined"

eval (Asn v e) =          do e' <-          eval e
                        modify (\s' -> (v, e') : s')
                        return e'

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

The Eval Function in Action: evalState, execState, and runState

```
ghci> a = Var "a" ; b = Var "b"
ghci> ex =    a <== 3    #    b <== a + 1    #    a * (b <== b + 1) + b

ghci> evalState (eval ex) []    -- Result only
20

ghci> execState (eval ex) []    -- Final state only
[("b",5),("b",4),("a",3)]

ghci> runState (eval ex) []    -- Both
(20,[("b",5),("b",4),("a",3)])
```

Harnessing Monads

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

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

```
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'
```

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

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

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

Harnessing Monads

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

FIXME: liftM, liftM2, ap, etc. Put earlier