"Being a nomad is okay, but you wouldn't believe the problems we have getting mail forwarded!"
4 Monads

What’s a monad?

Monad stances

State monads
What is a Monad?

A monad is a:

Parameterized Data Type

\[ \text{data } T \text{ a } = \ldots \]

with two operations:

Injection

\[ \text{return} :: a \rightarrow T \text{ a} \]

Continuation

\[ (\gg=) :: T \text{ a } \rightarrow (a \rightarrow T \text{ b}) \rightarrow T \text{ b} \]

In terms of Haskell:

\[
\text{class Monad m where}
\text{return} :: a \rightarrow m \text{ a}
(\gg=) :: m \text{ a } \rightarrow (a \rightarrow m \text{ b}) \rightarrow m \text{ b}
\]

(that satisfy three laws.)
A monad is NOT a thing.

A monad is a way of looking at things.

“X is a monad”:
• X can be treated as a monad
• X behaves like a monad

has different interpretations
4 Monads

What's a monad?

Monad metaphors

State monads
Monad Metaphors

- Context
- Box
- Label
- (Effectful) Computation

\( \equiv \)
The Box Metaphor

Context $\cong$ Box
The Box Metaphor

The things stored in the box

Put one thing in a box and close the box

"Repackaging":

\[ b \gg= f \]

(1) Open box \( b \) to access content (say \( x \))
(2) Generate new box(es) from box content using \( f \), i.e. \( f \ x \)
(3) Combine boxes into one result box

\[
\text{class Monad } m \text{ where }
\begin{align*}
\text{return} & : a \to m a \\
\text{(\(\gg=\))} & : m a \to (a \to m b) \to m b
\end{align*}
\]
The “Safety Seal” Box

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

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

```
class Monad Maybe where
  return = Just
  Just x >>= f = f x
  Nothing >>= _ = Nothing
```

Put one thing in a box and close the box

“Repackaging”:

\[ b >>= f \]

1. Open box \( b \) to access content (say \( x \))
2. Generate new box(es) from box content using \( f \), i.e. \( f x \)
3. Combine boxes into one result box

The box

The things stored in the box

A broken seal stays broken

Box contains only a single item

Creates only one box, i.e. step (3) is not needed
Safety Seal Example

type Height = Int

rise :: Int → Height → Height
rise x h = h+x

drop :: Int → Height → Height
drop x h | h>=x = h-x
          | otherwise = error "Plane Crash"

takeOff :: Height → Height
takeOff 0 = 50

loop :: Height → Height
loop = rise 100 . drop 150 . rise 50

> rise 40 . loop . rise 100 . takeOff $ 0
   190

> rise 40 . loop . takeOff $ 0
   *** Exception: Plane Crash

no control over runtime errors

Flight control
Using Maybe for Errors

\[
\text{riseM} :: \text{Int} \rightarrow \text{Maybe Height} \rightarrow \text{Maybe Height}
\]
\[
\text{riseM} \ x \ (\text{Just } h) = \text{Just } (h+x)
\]
\[
\text{riseM} \ _ \ \text{Nothing} = \text{Nothing}
\]

\[
\text{dropM} :: \text{Int} \rightarrow \text{Maybe Height} \rightarrow \text{Maybe Height}
\]
\[
\text{dropM} \ x \ (\text{Just } h) \ | \ h\geq x = \text{Just } (h-x)
\]
\[
\text{dropM} \ _ \ \text{Nothing} = \text{Nothing}
\]

\[
\text{takeOffM} :: \text{Maybe Height} \rightarrow \text{Maybe Height}
\]
\[
\text{takeOffM} \ (\text{Just } _) = \text{Just } 50
\]
\[
\text{takeOffM} \ \text{Nothing} = \text{Nothing}
\]

\[
\text{loopM} :: \text{Maybe Height} \rightarrow \text{Maybe Height}
\]
\[
\text{loopM} = \text{riseM} \ 100 \ . \ \text{dropM} \ 150 \ . \ \text{riseM} \ 50
\]

\[
\text{Encapsulate in } \_ \rightarrow \_ \rightarrow \text{Maybe Height}
\]

\[
\text{data} \ \text{Maybe} \ a = \text{Just } a \ | \ \text{Nothing}
\]

\[
\text{Error handling is included in every function}
\]

\[
\text{Encapsulate in } \_ \rightarrow \_ \rightarrow \text{Maybe Height}
\]

\[
\text{Just } x \ \_ \rightarrow \_ = \_ \rightarrow \_ = \text{Nothing}
\]

\[
> \ \text{riseM} \ 40 \ . \ \text{loopM} \ . \ \text{riseM} \ 100 \ . \ \text{takeOffM} \ $ \ \text{Just } 0
\]
\[
\text{Just } 190
\]

\[
> \ \text{riseM} \ 40 \ . \ \text{loopM} \ . \ \text{takeOffM} \ $ \ \text{Just } 0
\]
\[
\text{Nothing}
\]
Abstracting Error Handling

```haskell
mrise :: Int → Height → Maybe Height
mrise x h = Just (h+x)

mdrop :: Int → Height → Maybe Height
mdrop x h | h>=x = Just (h-x)
            | otherwise = Nothing

mtakeOff :: Height → Maybe Height
mtakeOff _ = Just 50

mloop :: Height → Maybe Height
mloop h = mrise 50 h >>= mdrop 150 >>= mrise 100
```

```
data Maybe a = Just a | Nothing

Just x >>= f = f x
Nothing >>= _ = Nothing
```

```
> Just 0 >>= mtakeOff >>= mrise 100 >>= mloop >>= mrise 40
  Just 190

> Just 0 >>= mtakeOff >>= mloop >>= mrise 40
  Nothing
```

action composition through >>=
Exercises

Explain why the difference in the ordering of the functions is needed

\[
\text{riseM } 40 \ . \ \text{loopM} \ . \ \text{riseM } 100 \ . \ \text{takeOffM} \ \& \ \text{Just } 0
\]

\[
\text{Just } 0 \ \gg=\ \text{mtakeOff} \ \gg=\ \text{mrise } 100 \ \gg=\ \text{mloop} \ \gg=\ \text{mrise } 40
\]

Define the function $\triangleleft\triangleright$ that allows us to write the example as follows

\[
\text{mrise } 40 \ \triangleleft\triangleright\ (\text{mloop} \ \triangleleft\triangleright\ (\text{mrise } 100 \ \triangleleft\triangleright\ (\text{mtakeOff} \ \triangleleft\triangleright\ \text{Just } 0)))
\]
The “Collection” Box

The things stored in the box

Put one thing in a box and close the box

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

Create a new box for each list element (step 2)

Flatten boxes (step 3)

recht = a : [a] |

The box

data [a] = a : [a] | []

The things stored in the box

(1) Open box `b` to access content (say `x`)
(2) Generate new box(es) from box content using `f`, i.e. `f x`
(3) Combine boxes into one result box

class Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
Collection Monads

Histograms

\[
\text{plot} :: \text{Int} \to \text{String} \\
\text{plot } n = \text{take } n (\text{repeat } 'X') ++ "\n"
\]

\[
\text{histogram} :: \text{[Int]} \to \text{String} \\
\text{histogram } xs = xs >>= \text{plot}
\]

\[
\text{hist} :: \text{[Int]} \to \text{IO} () \\
\text{hist} = \text{putStrLn} \cdot \text{histogram}
\]

Cartesian Products

\[
\text{cart} :: \text{[a]} \to \text{[b]} \to \text{[(a,b)]} \\
\text{cart } xs ys = xs >>= (\langle x, y \rangle \mapsto \langle x, y \rangle)
\]

\[
> \text{hist } [5,7,3,4] \\
\text{xxxxxx} \\
\text{xxxxxxxx} \\
\text{xxx} \\
\text{xxxx}
\]

\[
> \text{cart } [2..5] ['a'..'c'] \\
[\text{(2,'a'),(2,'b'),(2,'c'),(3,'a'),(3,'b'),(3,'c')}, \ldots]
\]
The “do” Notation

\[ f :: a \rightarrow T b \]
\[ f \ x = e(x) \]

\[ m ::= T \ a \]
\[ m >>\ f \]

\[ m +++ \ \_\rightarrow e \quad \equiv \quad m >>\ e \]

\[ \text{bind value(s) extracted from monad to variable} \]

\[ \text{mloop} :: \text{Height} \rightarrow \text{Maybe Height} \]
\[ \text{mloop} \ h = \text{mrise} 50 \ h >>\ \text{mdrop} 150 >>\ \text{mrise} 100 \]

\[ \text{cart} :: \text{[a]} \rightarrow \text{[b]} \rightarrow \text{[(a,b)]} \]
\[ \text{cart} \ xs \ ys = \text{xs} >>\ (\_\rightarrow \text{map} \ (x,) \ y) \]

\[ \text{mloop} \ h = \text{do} \ u \leftarrow \text{mrise} 50 \ h \]
\[ d \leftarrow \text{mdrop} 150 \ u \]
\[ \text{mrise} 100 \ d \]

\[ \text{cart} \ xs \ ys = \text{do} \ x \leftarrow \text{xs} \]
\[ y \leftarrow \text{ys} \]
\[ \text{return} \ (x,y) \]
Exercises

Predict the results of the following expressions

return 3 >>= do [4]
return 3 >>= Just 4
[3] >>= Just 4
Just 4 >>= Just
Nothing >>= Just
[] >>= return

[3,4] >>= return
Just 4 >>= Just . succ
[3,4] >>= return . succ
Just 3 >>= return . Just
Just 3 >>= return . return
[1..3] >>= (\x->[1..x])
Monad Metaphors

Box ⟶ (Effectful) Computation

Label
The Label Metaphor

Label
The Label Metaphor

Assign a (default) label to an item

“Relabeling”:

1 >>= f
(1) Isolate label from 1 to reveal item (say x)
(2) Generate new labeled item(s) using f, i.e. f x
(3) Combine labeled items into one labeled item

class Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b
Collecting Traces

Assign a (default) label to an item

```
data Trace a = T String a
```

Labeled items

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

The things to be labeled

```
instance Monad Trace where
  return x = T "" x
  (T t x) >>= f = let T u y = f x
                  in T (t++u) y
```

Isolated label (step 1)

"Relabeling":

1. Isolate label from l to reveal item (say x)
2. Generate new labeled item(s) using f, i.e. f x
3. Combine labeled items into one labeled item

Empty default label

New labeled item (step 2)

Concatenating traces (step 3)
Tracing Computations

```haskell
data Trace a = T String a

instance Monad Trace where
    return x = T "" x
    (T t x) >>= f = let T u y = f x
                    in T (t++u) y

trace :: Show a => a -> Trace a
trace x = T (show x++" ;") x

data Exp = N Int
          | Plus Exp Exp
          | Neg Exp

eval :: Exp -> Int
eval (N i) = i
eval (Plus e e') = eval e + eval e'
eval (Neg e) = -(eval e)

teval :: Exp -> Trace Int
teval (N i) = trace i
teval (Plus e e') = teval e >>= (\i->
                        teval e' >>= (\j->
                                        trace (i+j)))
teval (Neg e) = teval e >>= (\i->
                               trace (-i))
```

Interpreter

Tracing Interpreter
Tracing Computations

\[
\text{trace} :: \text{Show } a \Rightarrow a \rightarrow \text{Trace } a \\
\text{trace } x \equiv \text{T} (\text{show } x++";") x
\]

\[
\text{teval} :: \text{Exp } \rightarrow \text{Trace } \text{Int} \\
\text{teval } (\text{N } i) \equiv \text{trace } i \\
\text{teval } (\text{Plus } e e') \equiv \text{teval } e \gg= (\langle i \rightarrow \text{teval } e' \gg= (\langle j \rightarrow \text{trace } (i+j) ))) \\
\text{teval } (\text{Neg } e) \equiv \text{teval } e \gg= (\langle i \rightarrow \text{trace } (-i) )
\]

\[
> \text{eval } $ \text{Plus } (\text{N } 9) (\text{Neg } (\text{Plus } (\text{N } 3) (\text{N } 4)))$ \\
2
\]

\[
> \text{teval } $ \text{Plus } (\text{N } 9) (\text{Neg } (\text{Plus } (\text{N } 3) (\text{N } 4)))$ \\
\text{T } "9;3;4;7;-7;2;" 2
\]
Exercise

Explain why the following alternative implementations are all equivalent

\[
\begin{align*}
teval \ (N \ i) &= \ trace \ i \ >> \ return \ i \\
teval \ (N \ i) &= \ trace \ i \ >>> \ return \\
teval \ (N \ i) &= \ trace \ i
\end{align*}
\]

\[
m \ >>= \ \_ \rightarrow e \equiv m \ >> \ e
\]
Improved Tracing

```haskell
trace :: (Show a, Show b) => a -> b -> Trace b
trace e x = T (show e ++ " => " ++ show x ++ "\n") x

pt :: Trace a -> IO ()
pt (T t _) = putStrLn t

> pt $ teval $ Plus (N 9) (Neg (Plus (N 3) (N 4)))
N 9 => 9
N 3 => 3
N 4 => 4
Plus (N 3) (N 4) => 7
Neg (Plus (N 3) (N 4)) => -7
Plus (N 9) (Neg (Plus (N 3) (N 4))) => 2
```

Show arguments of evaluations too
Monad Metaphors

Box ⟶ (Effectful) Computation

Label
The Computation Metaphor

(Effectful) Computation

Monads
Counting Computations

data Exp = N Int
    | Plus Exp Exp
    | Neg Exp

eval :: Exp → Int
eval (N i) = i
eval (Plus e e') = eval e + eval e'
eval (Neg e) = -(eval e)

type Counter = Int
evalC :: Counter → Exp → (Int,Counter)
evalC n (N i) = (i,n)
evalC n (Plus e e') = (i+j,0+1)
    where (i,m) = evalC n e
         (j,o) = evalC m e'
evalC n (Neg e) = (-i,m+1)
    where (i,m) = evalC n e

> eval $ Plus (N 9) (Neg (Plus (N 3) (N 4)))
2

> evalC 0 $ Plus (N 9) (Neg (Plus (N 3) (N 4)))
(2,3)
Exercise

Try to implement a Count monad and use it for re-implementing evalC.

Why does it not work?

data Count a = C Int a

instance Monad Count where
  return x = C 0 x
  C c x >>= f = ...
The Computation Metaphor

(Effectful) Computation

Trivial computation (noop)
just returning a result

“Threading”:
_ c >>= f

(0) Build a computation such that if run does:
(1) Run computation c to produce a result (say x)
(2) Generate a new computation d from result using f, i.e. f x
(3) Run computation d

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

What’s a monad?

Monad metaphors

State monads
**The State (Transformer) Monad**

**instance Monad (State s) where**
- `return x = State (\s → (x, s))`
- `State c >>= f = State (\s → let (x, s') = c s in d s')`

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

**Result of computation**
- `data State s a = State (s → (a, s))`

**Threading**:
1. Run computation `c` to produce a result (say `x`) and new state
2. Generate a new computation `d` from result using `f`, i.e. `f x`
3. Run computation `d`

**Noop computation: return state unchanged plus result**
- Build a computation such that if `run` does:
  - (0) Build a computation such that if `run` does:
  - (1) Run computation `c` to produce a result (say `x`) and new state
  - (2) Generate a new computation `d` from result using `f`, i.e. `f x`
  - (3) Run computation `d`
data State s a = State (s → (a, s))

\[
\text{State } c = \text{State } (s → \text{State } (x, s'))
\]

\[
\text{State } d = f x \text{ in } d s'
\]

\[
\text{State } c >>= f = \text{State } (s → \text{let } (x, s') = c s \text{ in } d s')
\]
Monadic Counting

\[
\text{seval :: } \text{Exp} \rightarrow \text{Count Int}
\]
\[
\text{seval} (N\ i) = \text{return } i
\]
\[
\text{seval} (\text{Plus } e\ e') = \text{do } i \leftarrow \text{seval } e
\]
\[
\text{seval} (\text{Neg } e') = \text{do } i \leftarrow \text{seval } e
\]
\[
\text{incCounter :: } \text{Count } ()
\]
\[
\text{incCounter} = \text{State } \text{\c } \rightarrow ((),\text{c+1})
\]

\[
\text{type Count } a = \text{State } \text{Counter } a
\]

\[
\text{evalC :: } \text{Counter} \rightarrow \text{Exp} \rightarrow (\text{Int},\text{Counter})
\]
\[
\text{evalC} n (N\ i) = (i,n)
\]
\[
\text{evalC} n (\text{Plus } e\ e') = (i+j,o+1)
\]
\[
\text{evalC} n (\text{Neg } e) = (-i,m+1)
\]

\[
\text{where } (i,m) = \text{evalC} n\ e
\]
\[
(j,o) = \text{evalC} m\ e'
\]

\[
\text{evalC} n (\text{Plus } e\ e') = (i+j,o+1)
\]

\[
\text{where } (i,m) = \text{evalC} n\ e
\]
\[
(j,o) = \text{evalC} m\ e'
\]

\[
\text{evalC} n (\text{Neg } e) = (-i,m+1)
\]

\[
\text{where } (i,m) = \text{evalC} n\ e
\]

\[
\text{evalC} n (\text{Neg } e) = (-i,m+1)
\]

\[
\text{where } (i,m) = \text{evalC} n\ e
\]

\[
\text{runWith } = \text{flip runState}
\]

\[
> \text{runWith } 0 \$ \text{seval }$
\]
\[
\text{Plus } (N\ 9) (\text{Neg } (\text{Plus } (N\ 3) (N\ 4)))
\]
\[
(2,3)
\]
State Monad Operations

```
data State s a = State (s → (a,s))

runState :: State s a → s → (a,s)
runState (State f) = f

runWith :: s → State s a → (a,s)
runWith = flip runState

onState :: (s → s) → State s ()
onState f = State $ \s → ((,),f s)

fromState :: (s → a) → State s a
fromState f = State $ \s → (f s,s)

readState :: State s s
readState = State $ \s → (s,s)

init :: s → State s ()
init s = State $ \_ → ((,),s)
```
Using the State monad, define a function mineval that keeps track of the lowest intermediate result during an evaluation.

Using the State monad, redefine the function teval for tracing computations.
Unifying Monad Stances

Safety Seal Box

\[
\text{data} \ \text{Maybe} \ a = \text{Just} \ a \ |
\text{Nothing}
\]

\[
\text{class} \ \text{Monad} \ \text{Maybe} \ \\
\text{return} = \text{Just} \\
\text{Just} \ x \ = \ = f \ = \ f \ x \\
\text{Nothing} \ = \ = _ \ = \ \text{Nothing}
\]

Collection Box

\[
\text{data} \ [a] = a : [a] \ |
[]
\]

\[
\text{class} \ \text{Monad} \ [a] \ \\
\text{return} x = [x] \\
xs \ = \ = f \ = \ \text{concat} \ (\text{map} \ f \ xs)
\]

Computational Interpretation

Computations that can fail

Nondeterministic Computations

(Effectful) Computation
Nondeterminism

```haskell
data [a] = a : [a] | []

class Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)

choose :: [a → a] → [a] → [a]
choose [] xs = []
choose (f:fs) xs = map f xs ++ choose fs xs

interest p x = x*(1+p/100)
spend = flip (-)
keep = interest 5
wallStreet _ = 0

> choose [wallStreet,keep] . choose [spend 400,keep] $ [1000]
[0.0,0.0,630.0,1102.5]
```