

# What is a monad?

A monad is a triple  $(T, \eta, \mu)$  where  $T$  is an endofunctor  $T: X \rightarrow X$  and  $\eta: I \rightarrow T$  and  $\mu: T \times T \rightarrow T$  are 2 natural transformations satisfying these laws:

Identity law:  $\mu(\eta(T)) = T = \mu(T(\eta))$

Associative law:  $\mu(\mu(T \times T) \times T) = \mu(T \times \mu(T \times T))$

In other words: *"a monad in  $X$  is just a monoid in the category of endofunctors of  $X$ , with product  $\times$  replaced by composition of endofunctors and unit set by the identity endofunctor"*

## What's the problem?

CS 252:

*Advanced Programming Language Principles*



# Monads

Prof. Tom Austin

San José State University

# Review applicative functor lab

(in class)

# Fear, uncertainty, & doubt



*Review: what is a functor?*

A functor is something  
that can be mapped over.

```
fmap :: Functor f =>  
      (a -> b) -> f a -> f b
```

## Problem with Functors

- This works great:

```
fmap (+1) (Just 3)
```

- But this is an error:

```
fmap (+) (Just 3) (Just 4)
```

```
Couldn't match expected type  
  `Maybe a1 -> t0`  
with actual type  
  `Maybe (a0 -> a0)`  
...
```

# What is an *applicative functor*?

A functor that you can  
apply to other functors.

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

## Problem with Applicative Functors

- Now this works:

```
import Control.Applicative
fmap (+) (Just 3) <*> Just 4
```

- Which we could rewrite as:

```
Just (+3) <*> Just 4
```

- But this won't work:

```
Just (+3) <*> Just (+4) <*> Just 5
```

- No instance for (Num (a0 -> b0))  
arising from a use of `+`

...

## Monads to the rescue

- Monads can chain through a series of functions:

```
Just 3 >>= (\x -> Just (x+4))  
      >>= (\x -> Just (x+5))
```

- Or equivalently

```
return 3 >>= (\x -> return (x+4))  
         >>= (\x -> return (x+5))
```

So what *is* a Monad?



# The bind function

```
fmap :: Functor f =>  
      (a -> b) -> f a -> f b
```

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

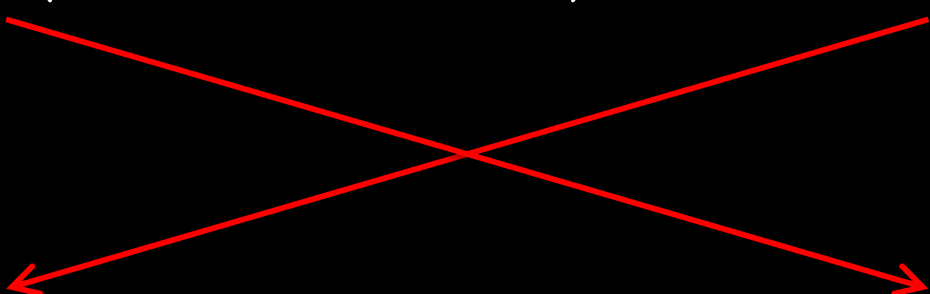
called "bind"

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

> (\x -> x+1) <\$> Just 1  
Just 2

> Just (\x -> x+1) <\*> Just 1  
Just 2

> Just 1 >>= (\x -> Just \$ x+1)  
Just 2



## Reimplementing `>>=`

`applyMaybe`

- Applies a function to a `Maybe` value
- Returns another `Maybe` value.

# applyMaybe

```
applyMaybe :: Maybe a  
            -> (a -> Maybe b)  
            -> Maybe b  
applyMaybe Nothing f = Nothing  
applyMaybe (Just x) f = f x
```

```
> Just 3 `applyMaybe`  
  (\x -> Just $ x * 2) `applyMaybe`  
  (\x -> Just $ x - 1)  
Just 5
```

```
> Just 3 `applyMaybe`  
  (\_ -> Nothing) `applyMaybe`  
  (\x -> Just $ x - 1)  
Nothing
```

## The Monad Typeclass

```
class Monad m where
```

```
  return :: a -> m a
```

```
  (>>=) :: m a -> (a -> m b) -> m b
```

```
  (>>)  :: m a -> m b -> m b
```

```
  x >> y = x >>= \_ -> y
```

```
  fail :: String -> m a
```

```
  fail msg = error msg
```

## Robot example (sans monads)

Model a robot moving on a grid:

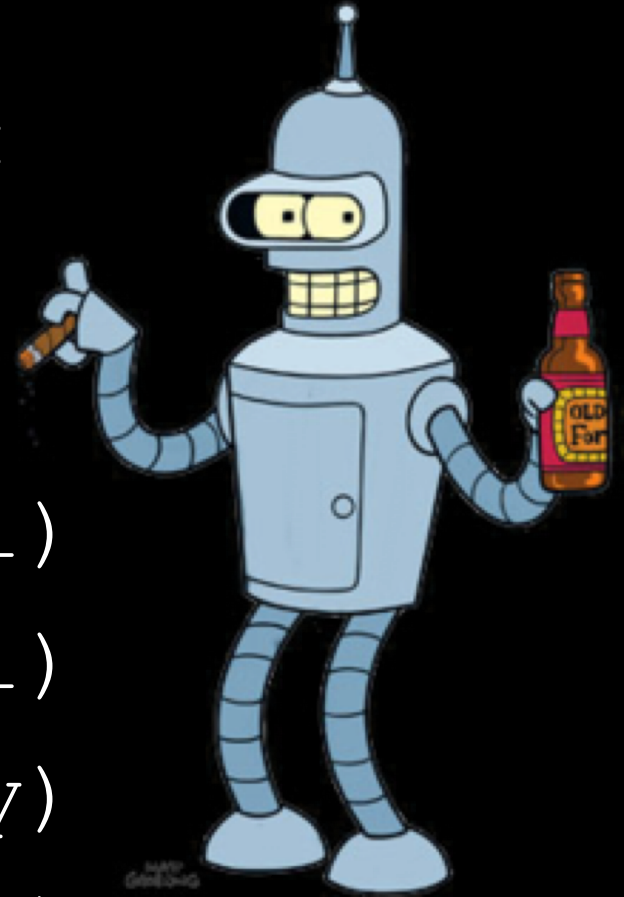
```
type Pos = (Int, Int)
```

```
up (x, y) = (x, y+1)
```

```
down (x, y) = (x, y-1)
```

```
left (x, y) = (x-1, y)
```

```
right (x, y) = (x+1, y)
```



```
x -: f = f x
```

```
start = (0,0)
```

```
> start -: up -: right
```

```
(1,1)
```

```
> start -: up -: left -: left -:
```

```
right -: down
```

```
(-1,0)
```

Now let's modify our program to account for failure.



If Bender finds beer, he ignores all future commands.

## The Maybe Monad

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

## Defining where Bender ignores commands

```
beerPos = Map.empty  
-: Map.insert (0,2) True  
-: Map.insert (-1,3) True  
-: Map.insert (-3,-8) True
```

```
moveTo :: Pos -> Maybe Pos
```

```
moveTo p =
```

```
  if Map.member p beerPos
```

```
    then Nothing
```

```
    else Just p
```

```
up (x, y) = moveTo (x, y+1)
```

```
down (x, y) = moveTo (x, y-1)
```

```
left (x, y) = moveTo (x-1, y)
```

```
right (x, y) = moveTo (x+1, y)
```

```
> return start >>= up >>= left  
  >>= left >>= right >>= down  
Just (-1,0)
```

```
> return start >>= left >>= left  
  >>= up >>= up >>= right >>= up  
  >>= right >>= right >>= down  
Nothing
```

What if we have many Maybe  
values that we need to compute?

*Theirs not to reason why,  
Theirs but to do and die.*

**--Alfred Tennyson**

## Division example, sans do

```
mydiv x y =  
  x >>= (\numer ->  
  y >>= (\denom ->  
  if denom > 0 then  
    Just $ numer `div` denom  
  else fail "div by 0"))
```

## Division example, with do

```
mydiv' x y = do
  numer <- x
  denom <- y
  if denom > 0 then
    Just $ numer `div` denom
  else fail "div by 0"
```

## Division example, with do & return

```
mydiv' x y = do
  numer <- x
  denom <- y
  if denom > 0 then
    return $ numer `div` denom
  else fail "div by 0"
```

# List Monad

```
instance Monad [] where
```

```
    return x = [x]
```

```
    xs >>= f = concat (map f xs)
```

```
    fail _ = []
```

```
listOfTuples :: [(Int,Char)]
listOfTuples = do
    n <- [1,2]
    ch <- ['a','b']
    return (n,ch)
```

list comprehensions:  
syntactic sugar for  
using lists as monads.

```
[(n,ch) | n <- [1,2],
          ch <- ['a','b']]
```

## 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, newState) = h s  
        (State g) = f a  
    in g newState
```

## Lab: Monads

This lab is available in Canvas.  
Starter code is available on the  
course website.