Monads and all that…
I. Monads

John Hughes
Chalmers University/Quviq AB

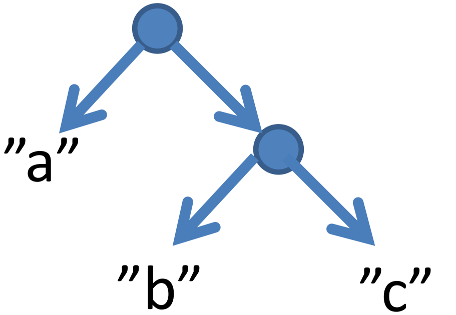
Binary Trees in Haskell

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

- Cf Coq:

```
Inductive tree (A:Set) : Set :=
  | leaf   : A -> tree A
  | branch : tree A -> tree A -> tree A
```

```
t = Branch (Leaf “a”)
        (Branch (Leaf “b”)
                (Leaf “c”))
```



Mapping over Trees

```
treeMap f (Leaf a)       = Leaf (f a)
treeMap f (Branch l r) =
  Branch (treeMap f l) (treeMap f r)
```

```
treeMap toUppers t
```

→

```
Branch (Leaf "A")
       (Branch (Leaf "B")
               (Leaf "C"))
```

```
treeMap :: (t -> a) -> Tree t -> Tree a
```

- Tree is a functor!

Functors in Haskell

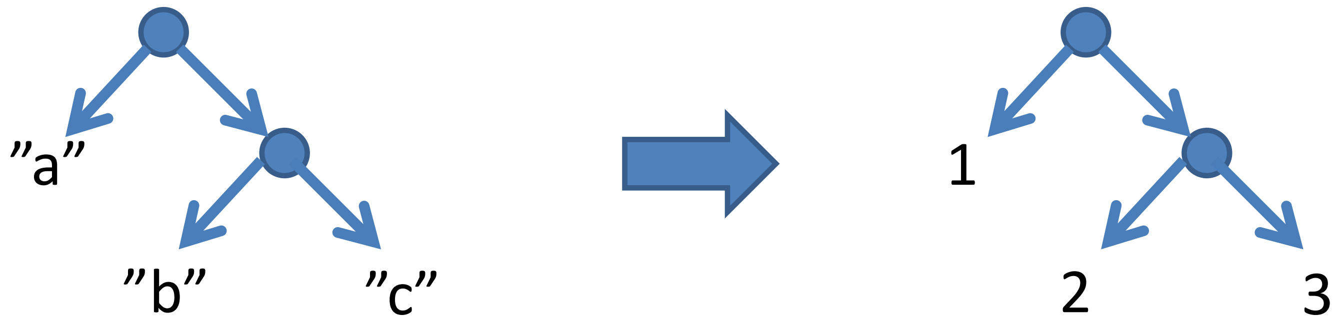
```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

```
instance Functor Tree where
  fmap f (Leaf a) = Leaf (f a)
  fmap f (Branch l r) = Branch (fmap f l) (fmap f r)
```

```
fmap toUppers t

Branch (Leaf "A")
       (Branch (Leaf "B")
               (Leaf "C"))
```

Label Nodes with DFO Index



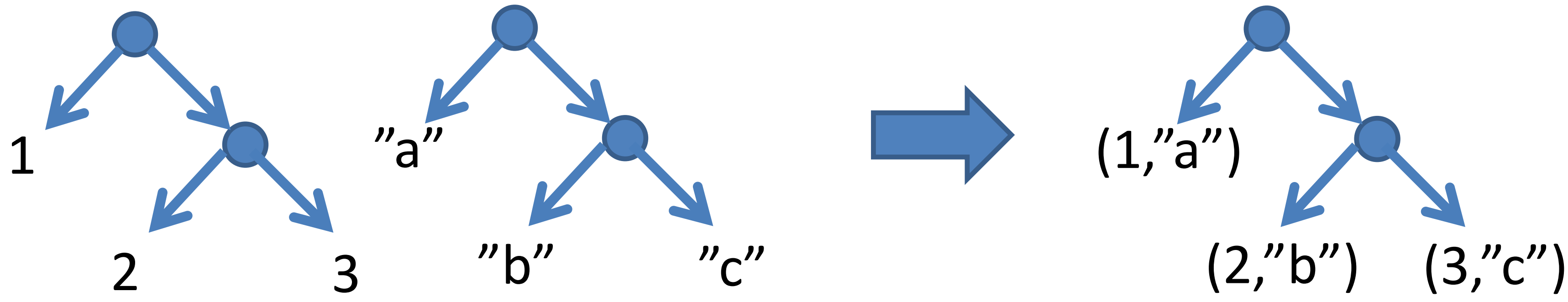

```
number (Leaf a) = Leaf (tick ())
number (Branch l r) = Branch (number l) (number r)
```

```
number (Leaf a) s = (Leaf s,s+1)
number (Branch l r) s =
  let (l',s')  = number l s
      (r',s'') = number r s'
  in (Branch l' r',s'')
```

Error prone

Zipping Trees

```
zipTree :: Tree a -> Tree b -> Tree (a,b)
```




```
zipTree (Leaf a) (Leaf b) =
  Leaf (a,b)
zipTree (Branch l r) (Branch l' r') =
  Branch (zipTree l l') (zipTree r r')
```

BUT what if…

```
*Lecture1> zipTree (Leaf "a") (Branch (Leaf "b") (Leaf "c"))
*** Exception: Lecture1.hs:(31,1)-(32,74): Non-exhaustive
patterns in function zipTree
```

- Easy to solve:

```
zipTree (Leaf a) (Leaf b) =
  Leaf (a,b)
zipTree (Branch l r) (Branch l' r') =
  Branch (zipTree l l') (zipTree r r')
zipTree _ _ = throw TreesOfDifferentShape
```

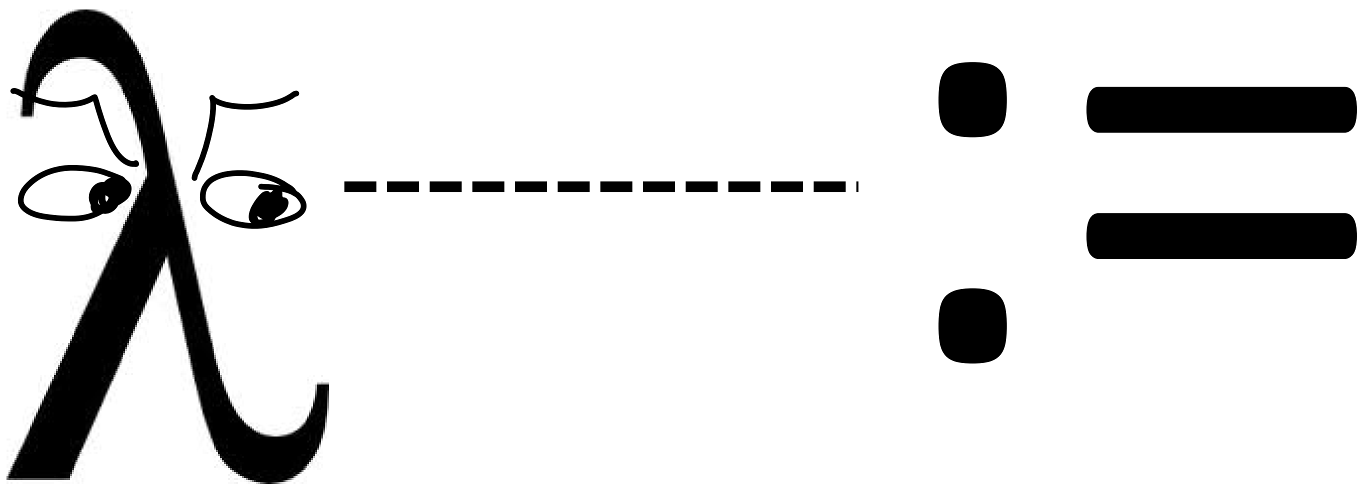
```
… catch (zipTree t1 t2) …
```

Modelling Exceptions

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

```
zipTree :: Tree a -> Tree b -> Maybe (Tree (a,b))
zipTree (Leaf a) (Leaf b) =
  Just (Leaf (a,b))
zipTree (Branch l r) (Branch l' r') =
  case zipTree l l' of
    Nothing -> Nothing
    Just l'' ->
      case zipTree r r' of
        Nothing -> Nothing
        Just r'' ->
          Just (Branch l'' r'')
zipTree _ _ = Nothing
```

Effect Envy



Do we need to use *effects* to write modular code??

Let’s examine the code…

```
zipTree :: Tree a -> Tree b -> Ma
zipTree (Leaf a) (Leaf b) =
  Just (Leaf (a,b))
zipTree (Branch l r) (Branch l' r') =
  case zipTree l l' of
    Nothing -> Nothing
    Just l'' ->
      case zipTree r r' of
        Nothing -> Nothing
        Just r'' ->
          Just (Branch l'' r'')
zipTree _ _ = Nothing
```

This is how we return a value:
Just <expr>

This is how we use a value

Let’s abstract the common parts

```
Just (Leaf (a,b))

Just x

return x = Just x

return :: a -> Maybe a
```

```
case zipTree l l' of
  Nothing  -> Nothing
  Just l'' -> …

case x of
  Nothing  -> Nothing
  Just l'' -> f l''

x >>= f =
  case x of
    Nothing  -> Nothing
    Just l'' -> f l''

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

”bind”

”use x in f”

Revisiting the code

```
zipTree :: Tree a -> Tree b -> Maybe (Tree (a,b))
zipTree (Leaf a) (Leaf b) =
  return (Leaf (a,b))
zipTree (Branch l r) (Branch l' r') =
  zipTree l l' >>= \l'' ->
  zipTree r r' >>= \r'' ->
  return (Branch l'' r'')
zipTree _ _ = Nothing
```

Back to node numbering…

```
number (Leaf a) s = (Leaf s,s+1)
number (Branch l r) s =
  let (l',s')  = number l s
      (r',s'') = number r s'
  in (Branch l' r',s'')
```

This is how we use a value

This is how we return a value

```
return x  = \s -> (x,s)
(x >>= f) = \s -> let (a,s’) = x s in f a s’
```

Node numbering revisited

```
number (Leaf a) s = (Leaf s,s+1)
number (Branch l r) s =
  let (l',s')  = number l s
      (r',s'') = number r s'
  in (Branch l' r',s'')
```

…all the nasty state manipulation is gone

```
number (Branch l r) = number l >>= \l′ ->
                      number r >>= \r′ ->
                      return (Branch l′ r′)

number (Leaf a) = tick >>= \s ->
                  return (Leaf s)

tick s = (s,s+1)
```

Apart from in tick…

What are the types?

```
return x  = \s -> (x,s)
(x >>= f) = \s -> let (a,s') = x s in f a s'
```

```
return :: a -> s -> (a,s)
(>>=)  :: (s -> (a,s)) ->
          (a -> s -> (b,s)) ->
          s -> (b,s)

type State s a = s -> (a,s)

return :: a -> State s a
(>>=)  :: State s a -> (a -> State s b) -> State s b
```

Compare to:

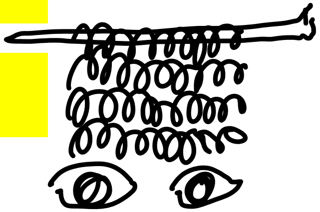
```
return :: a -> Maybe   a
(>>=)  :: Maybe   a -> (a -> Maybe   b) -> Maybe   b
```

The Common Pattern

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

```
instance Monad Maybe where …
```

```
instance Monad (State s) where …
```



- m a is *a computation delivering type a*
- return converts a *value* into a *computation*
- (>>=) *sequences* two computations

Example: Random Generation

- Programs using randomness must pass around a *seed*:

```
next  :: StdGen -> (Int,StdGe…
split :: StdGen -> (StdGen,St…
```

```
randomInt bound seed =
  let (n,seed') = next seed in …  mod  …
```

```
randomPair randomFst randomSnd seed =
  let (seed1,seed2) = split seed in
    (randomFst seed1, randomSnd seed2)
```

e.g. `randomPair (randomInt 3) (randomInt 3)` s_1
→ `(2,1)`

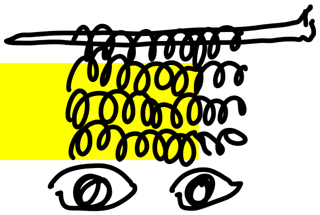
…if we give each generator its own seed

A Random List Generator

```
randomList randomEl seed =
  let (seed1,seed2) = split seed in
  case randomInt 5 seed1 of
    0 -> []
    _ ->
      let (seed3,seed4) = split seed2 in
      randomEl seed3 : randomList randomEl seed4
```

A Random Monad

```
type Random a = StdGen -> a
```



```
instance Monad Random where
  return a = \seed -> a
  x >>= f  = \seed ->
               let (seed1,seed2) = split seed
                   a             = x seed1
               in f a seed2)
```

```
generate :: Random Int
generate = \seed -> fst (next seed)
```

A Random Monad

isomorphic

```
newtype Random a =
  MkRandom {unRandom :: StdGen -> a}
```

Constructor

Destructor

```
instance Monad Random where
  return a = MkRandom (\seed -> a)
  x >>= f  = MkRandom (\seed ->
                 let (seed1,seed2) = split seed
                     a             = unRandom x seed1
                 in unRandom (f a) seed2)
```

```
generate :: Random Int
generate = MkRandom (\seed -> fst (next seed))
```

Random Lists Revisited

```
randomList randomEl seed =
  let (seed1,seed2) = split seed in
  case randomInt 5 seed1 of
    0 -> []
    _ ->
      let (seed3,seed4) = split seed2 in
      randomEl seed3 : randomList randomEl seed4
```

```
randomList randomEl =
  randomInt 5 >>= \n ->
  case n of
    0 -> return []
    _ -> randomEl            >>= \x ->
         randomList randomEl >>= \xs ->
         return (x:xs)
```

This is (almost) the Gen monad in QuickCheck

Example: Changing the World

- Wouldn't it be great if we could change the world with functional programs?

```
putStr :: String -> World -> World
```

  – A really nice way to

Can't duplicate the real world

- There's a problem:

```
let w1 = first_method world
    w2 = second_method world
in if nicer w1 w2 then w1 else w2
```

Can't discard the real world

We need to *enforce linearity!*

The IO Monad: Enforcing Linearity

```
newtype IO a = MkIO (World -> (a,World))
```

All IO a computations use the World linearly

Haskell main programs are IO computations

The programmer *cannot* call a World-> fun… but the RTS can, then updates the World

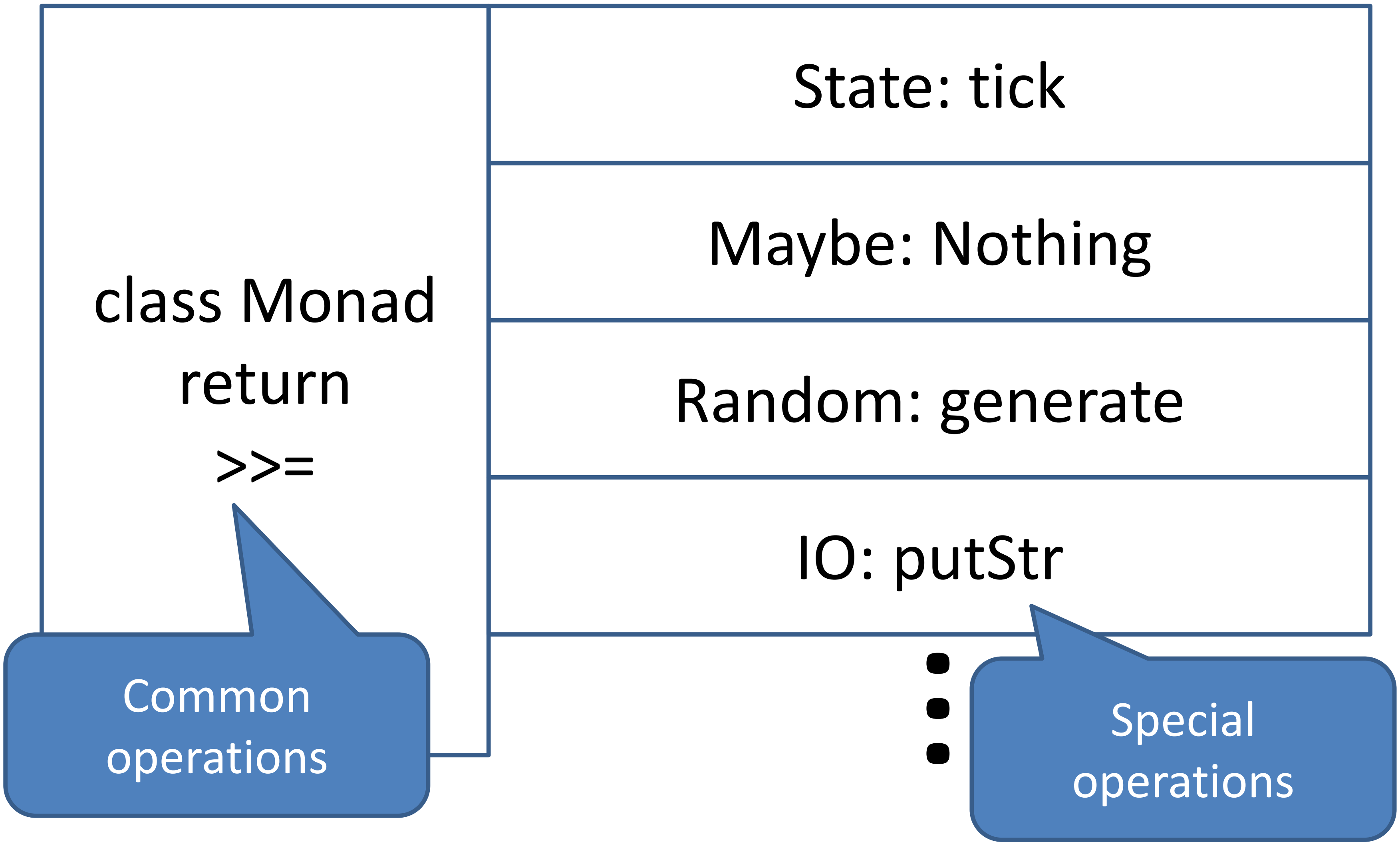
- T d, s e
  - = u ea

- The IO type i *abstract*
  - IO a can only be built from IO primitives…
  - …which use the world linearly
  - You can't "get rid of that pesky IO type"

I'm referentially transparent! You ain't got nuthin on me—it was the run-time system wot dun it!

:=

The Big Picture



What's the advantage of common plumbing?

- Libraries that work with *all* monads

- Syntactic support

Libraries: Control.Monad

- For example:

```
liftM2 :: Monad m => (a->b->c) -> m a -> m b -> m c
liftM2 f ma mb =
  ma >>= \a ->
  mb >>= \b ->
  return (f a b)
```

A monad is a functor ☺

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

```
sequence :: Monad m => [m a] -> m [a]
sequence = foldr (liftM2 (:)) (return [])
```

Syntactic Support

- Instead of

```
ma >>= \a ->
mb >>= \b ->
return (f a b)
```

- We write

```
do a <- ma
   b <- mb
   return (f a b)
```

Rewriting **do**

```
do pat <- expr
   block
```

```
expr >>= \pat ->
do block
```

```
do expr
   block
```

```
expr >>= \_ ->
do block
```

```
do expr
```

```
expr
```

Revisiting zipTree… again

```
zipTree :: Tree a -> Tree b -> Maybe (Tree (a,b))
zipTree (Leaf a) (Leaf b) =
  return (Leaf (a,b))
zipTree (Branch l r) (Branch l' r') =
  zipTree l l' >>= \l'' ->
  zipTree r r' >>= \r'' ->
  return (Branch l'' r'')
zipTree _ _ = Nothing
```

```
zipTree (Leaf a) (Leaf b) =
  return (Leaf (a,b))
zipTree (Branch l r) (Branch l′ r′) =
  liftM2 Branch (zipTree l l′) (zipTree r r′)
zipTree _ _ = Nothing
```

Revisiting node numbering… again

```
number (Branch l r) = number l >>= \l' ->
                      number r >>= \r' ->
                      return (Branch l' r')
number (Leaf a) = tick >>= \s ->
                  return (Leaf s)
```

```
number (Branch l r) =
  liftM2 Branch (number l) (number r)
number (Leaf a) =
  liftM Leaf tick
```

"Associative" **do**-notation begs the question…

```
do x <- e1
   y <- e2
   e3
```

=

```
do x <- e1
   do y <- e2
      e3
```

=?

```
do y <- do x <- e1
           e2
   e3
```

What about return?

```
do x <- e
   return x
```

=?

```
e
```

```
do y <- return x
   f y
```

=?

```
f x
```

The Monad Laws

- After desugaring:

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

- Do they hold for our monads??

NO!!!

- E.g. for State s

```
return x >>= ⊥
==
\s -> let (x',s') = return x s in ⊥ x' s'
==
\s -> ⊥ x s
==
\s -> ⊥
/=
⊥
==
⊥ x
```

NO!!!

- For Random

```
randomInt 5

randomInt 5 >>= return
```

Use different seeds to generate the Int!

Yes… near enough

- For total values
  - (Fast and loose reasoning is morally correct)
- Up to a reasonable equivalence
  - Same distribution in the case of Random

Testing the Monad Laws

- Let's use QuickCheck to test our monads!
- QuickCheck tests *properties* written as monomorphic functions

```
prop_Rev, prop_RevRev :: [Integer] -> Bool
prop_Rev xs    = reverse xs == xs
prop_RevRev xs = reverse (reverse xs) == xs
```

- QuickCheck tests using random arguments

```
*Lecture1> quickCheck prop_RevRev
+++ OK, passed 100 tests.
*Lecture1> quickCheck prop_Rev
*** Failed! Falsifiable (after 4 tests and 2 shrinks):
[0,1]
```

QuickCheck Constraints

- Property arguments must be
  - In class Arbitrary (can be generated)

```
class Arbitrary a where
  arbitrary :: Gen a
```

  - In class Show (can be printed)
- Functions are not printable, but QuickCheck Fun values are, and *contain* a function

```
Fun _ f :: Fun a b
```

The Monad Laws as Properties

- We state generic laws…

```
prop_LeftUnit x (Fun _ f) =
   (return x >>= f) == f x
prop_RightUnit m =
   (m >>= return) == m
prop_Assoc m (Fun _ f) (Fun _ g) =
   ((m >>= f) >>= g) == (m >>= \x -> f x >>= g)
```

- …and test particular instances

```
prop_MaybeAssoc :: Maybe Integer ->
                   Fun Integer (Maybe Integer) ->
                   Fun Integer (Maybe Integer) ->
                   Bool
prop_MaybeAssoc = prop_Assoc
```

Testing

- Of course, the tests pass

```
*MonadLaws> quickCheck prop_MaybeAssoc
+++ OK, passed 100 tests.
```

- But if we swap f and g on one side of prop_Assoc, to get a property that is false…

```
*MonadLaws> quickCheck prop_MaybeAssoc
*** Failed! Falsifiable (after 8 tests and 11 shrinks):
Just 0
{_->Just 0}
{_->Just 1}
```

f

g

Exercises