Before we begin

```
cd 00_setup && make
# Windows users: see setup.md
```

(setup)

Haskell 102

pravnar@, javran@, mihaimaruseac@, ibobyr@, nicuveo@

November 6, 2024

g2g

Goals

- 101: basic skills
  - Reading function types
  - Pattern matching
  - Data structures
- 102: first project
  - Genericity
  - IO and do notation
  - Build a game! (demo)

g2g

Roadmap

- 101 Recap
- Remaining obstacles
- Typeclasses overview
- Common examples
- Advanced syntax

Roadmap

- 101 Recap
- Remaining obstacles
- Typeclasses overview
- Typeclasses
- Typeclasses
- Typeclasses...

Curried functions, partial application

```haskell
f       :: Int ->    Int -> [Int]
```

(Haskell)

Curried functions, partial application

```haskell
f       :: Int -> ( Int -> [Int] )
f       :: Int ->   Int -> [Int]
```

(Haskell)

g2g

Curried functions, partial application

```haskell
f       :: Int -> ( Int -> [Int] )
f       :: Int ->   Int -> [Int]
f 1     ::            Int -> [Int]
```

(Haskell)

Curried functions, partial application

```haskell
 f       :: Int -> ( Int -> [Int] )
 f       :: Int ->   Int -> [Int]
 f 1     ::             Int -> [Int]

(f 1) 2  ::                   [Int]
```

(Haskell)

g2g

Curried functions, partial application

```
 f       :: Int -> ( Int -> [Int] )
 f       :: Int ->   Int -> [Int]
 f 1     ::            Int -> [Int]
 f 1   2 ::                   [Int]
(f 1) 2 ::                   [Int]
```

(Haskell)

g2g

Type constructors

```haskell
-- enum
data Bool  = False | True
data Color = Red | Green | Blue
```

(Haskell)

Type constructors

```haskell
-- enum
data Bool  = False | True
data Color = Red | Green | Blue
```

(Haskell)

```haskell
-- struct
data Point = Point { x :: Double, y :: Double }
```

(Haskell)

Type constructors

```haskell
-- enum
data Bool  = False | True
data Color = Red | Green | Blue
```
(Haskell)

```haskell
-- struct
data Point = Point { x :: Double, y :: Double }
```
(Haskell)

```haskell
-- a bit more interesting
data Minutes = Minutes Int
data Maybe a = Nothing | Just a
data List a  = Nil | Cell a (List a)
```
(Haskell)

Type constructors

```haskell
-- enum
data Bool  = False | True
data Color = Red | Green | Blue
```
(Haskell)

```haskell
-- struct
data Point = Point { x :: Double, y :: Double }
```
(Haskell)

```haskell
-- a bit more interesting
data Minutes = Minutes Int
data Maybe a = Nothing | Just a
data [a]     = [] | (a:[a])
```
(Haskell)

“Deconstructors” and pattern matching

```haskell
not :: Bool -> Bool
not True  = False
not False = True
```

(Haskell)

“Deconstructors” and pattern matching

```haskell
not :: Bool -> Bool
not True  = False
not False = True
```

(Haskell)

```haskell
magnitude :: Point -> Double
magnitude (Point x y) = sqrt $ x^2 + y^2
```

(Haskell)

“Deconstructors” and pattern matching

```haskell
not :: Bool -> Bool
not True  = False
not False = True
```

(Haskell)

```haskell
magnitude :: Point -> Double
magnitude (Point x y) = sqrt $ x^2 + y^2
```

(Haskell)

```haskell
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

(Haskell)

We know

- how to read function types
- how to declare new types
- how to do pattern matching

Our types are limited

```
$ ghci
>
```

Our types are limited

```
$ ghci
> data Color = Red | Green | Blue
```

Our types are limited

```
$ ghci
> data Color = Red | Green | Blue
>
```

Our types are limited

```
$ ghci
> data Color = Red | Green | Blue
> let mycolor = Red
```

Our types are limited

```
$ ghci
> data Color = Red | Green | Blue
> let mycolor = Red
>
```

Our types are limited

```
$ ghci
> data Color = Red | Green | Blue
> let mycolor = Red
> mycolor
```

Our types are limited

```
$ ghci
> data Color = Red | Green | Blue
> let mycolor = Red
> mycolor
    No instance for (Show Color)
    arising from a use of `print'
```

Our types are limited

```
$ ghci
> data Color = Red | Green | Blue
> let mycolor = Red
>
```

Our types are limited

```
$ ghci
> data Color = Red | Green | Blue
> let mycolor = Red
> mycolor == Red
```

Our types are limited

```
$ ghci
> data Color = Red | Green | Blue
> let mycolor = Red
> mycolor == Red
    No instance for (Eq Color)
    arising from a use of `(==)'
```

Type parameters constraints?

```haskell
sumI :: [Int] -> Int
sumI = foldl (+) 0

sumD :: [Double] -> Double
sumD = foldl (+) 0
```

(Haskell)

Type parameters constraints?

```haskell
sumI :: [Int] -> Int
sumI = foldl (+) 0

sumD :: [Double] -> Double
sumD = foldl (+) 0
```

(Haskell)

```haskell
sum :: [a] -> a
sum = foldl (+) 0
```

(Haskell)

Type parameters constraints?

```haskell
sumI :: [Int] -> Int
sumI = foldl (+) 0

sumD :: [Double] -> Double
sumD = foldl (+) 0
```

(Haskell)

```haskell
sum :: [a] -> a
sum = foldl (+) 0
```

(Haskell)

```
No instance for (Num a)
arising from a use of `(+)'
```

Cascading context

```haskell
getUser        :: Id   -> Maybe User
getNextOfKin   :: User -> Maybe Id
getPhoneNumber :: User -> Maybe PhoneNumber
```

(Haskell)

g2g

Cascading context

```haskell
getUser        :: Id   -> Maybe User
getNextOfKin   :: User -> Maybe Id
getPhoneNumber :: User -> Maybe PhoneNumber
```

(Haskell)

```haskell
getNextOfKinPhoneNumber :: Id -> Maybe PhoneNumber
```

(Haskell)

g2g

Cascading context

```haskell
getUser        :: Id   -> Maybe User
getNextOfKin   :: User -> Maybe Id
getPhoneNumber :: User -> Maybe PhoneNumber
```

(Haskell)

```haskell
getNextOfKinPhoneNumber :: Id -> Maybe PhoneNumber
getNextOfKinPhoneNumber userId =
```

(Haskell)

g2g

Cascading context

```haskell
getUser        :: Id   -> Maybe User
getNextOfKin   :: User -> Maybe Id
getPhoneNumber :: User -> Maybe PhoneNumber
```

(Haskell)

```haskell
getNextOfKinPhoneNumber :: Id -> Maybe PhoneNumber
getNextOfKinPhoneNumber userId = case getUser userId of
  Nothing   -> Nothing
  Just user ->
```

(Haskell)

g2g

Cascading context

```haskell
getUser        :: Id   -> Maybe User
getNextOfKin   :: User -> Maybe Id
getPhoneNumber :: User -> Maybe PhoneNumber
```

(Haskell)

```haskell
getNextOfKinPhoneNumber :: Id -> Maybe PhoneNumber
getNextOfKinPhoneNumber userId = case getUser userId of
  Nothing   -> Nothing
  Just user -> case getNextOfKin user of
    Nothing           -> Nothing
    Just nextOfKinId ->
```

(Haskell)

g2g

Cascading context

```haskell
getUser        :: Id   -> Maybe User
getNextOfKin   :: User -> Maybe Id
getPhoneNumber :: User -> Maybe PhoneNumber
```

(Haskell)

```haskell
getNextOfKinPhoneNumber :: Id -> Maybe PhoneNumber
getNextOfKinPhoneNumber userId = case getUser userId of
  Nothing   -> Nothing
  Just user -> case getNextOfKin user of
    Nothing           -> Nothing
    Just nextOfKinId -> case getUser nextOfKinId of
      Nothing         -> Nothing
      Just nextOfKin ->
```

(Haskell)

g2g

Cascading context

```haskell
getUser        :: Id   -> Maybe User
getNextOfKin   :: User -> Maybe Id
getPhoneNumber :: User -> Maybe PhoneNumber
```

(Haskell)

```haskell
getNextOfKinPhoneNumber :: Id -> Maybe PhoneNumber
getNextOfKinPhoneNumber userId = case getUser userId of
  Nothing   -> Nothing
  Just user -> case getNextOfKin user of
    Nothing          -> Nothing
    Just nextOfKinId -> case getUser nextOfKinId of
      Nothing         -> Nothing
      Just nextOfKin -> getPhoneNumber nextOfKin
```

(Haskell)

- Can't apply regular functions to IO values

- Can't apply regular functions to IO values

```
$ ghci
>
```

- Can't apply regular functions to IO values

```
$ ghci
> let name = getLine -- IO String
>
```

- Can't apply regular functions to IO values

```
$ ghci
> let name = getLine -- IO String
> putStr $ "Hello " ++ name ++ "!"
```

Can't apply regular functions to IO values

```
$ ghci
> let name = getLine -- IO String
> putStr $ "Hello " ++ name ++ "!"
    Expected type: `String'
      Actual type: `IO String'
    In second argument of (++)
```

IO

- Can't apply regular functions to IO values
- Can't get values out of IO

IO

- Can't apply regular functions to IO values
- Can't get values out of IO
- Can't pattern match on IO

IO

- Can't apply regular functions to IO values
- Can't get values out of IO
- Can't pattern match on IO

IO's implementation
details are hidden

We don't know

- how to extend our data types
- how to express type constraints
- how to chain contextual functions
- how to use IO

Typeclasses

```haskell
class Show a where
  show :: a -> String
```

(typeclass)

Typeclasses

```haskell
class Show a where
  show :: a -> String
```
(typeclass)

```haskell
data Color = Red | Green | Blue
```
(type)

Typeclasses

```haskell
class Show a where
  show :: a -> String
```
(typeclass)

```haskell
data Color = Red | Green | Blue
```
(type)

```haskell
instance Show Color where
  show Red   = "Red"
  show Green = "Green"
  show Blue  = "Blue"
```
(instance)

Constraints

```
show :: Show a => a -> String
```
(Show)

Constraints

```
show :: Show a => a -> String
```
(Show)

```
sum  :: Num a  => [a] -> a
```
(Num)

Constraints

```
show :: Show a => a -> String
```
(Show)

```
sum  :: Num a  => [a] -> a
```
(Num)

```
(==) :: Eq a   => a -> a -> Bool
```
(Eq)

Constraints in instance declarations

```haskell
instance Show (Maybe a) where
```

(Haskell)

Constraints in instance declarations

```haskell
instance Show (Maybe a) where
  show Nothing  = "Nothing"
```

(Haskell)

Constraints in instance declarations

```haskell
instance Show (Maybe a) where
  show Nothing  = "Nothing"
  show (Just x) = "Just " ++ show x
```

(Haskell)

Constraints in instance declarations

```haskell
instance Show a => Show (Maybe a) where
  show Nothing  = "Nothing"
  show (Just x) = "Just " ++ show x
```

(Haskell)

Quick tour

Show Read Eq Ord Bounded Enum

Show Read Eq Ord Bounded Enum

```
data Color = Red | Green | Blue                    (def)
```

```
> show Blue
```

Show Read Eq Ord Bounded Enum

```haskell
data Color = Red | Green | Blue
```
(def)

```
> show Blue
"Blue"
```

Show Read Eq Ord Bounded Enum

```haskell
data Color = Red | Green | Blue
```
(def)

```
> show Blue
"Blue"
> read "Green" :: Color
```

Show Read Eq Ord Bounded Enum

```haskell
data Color = Red | Green | Blue
```
(def)

```
> show Blue
"Blue"
> read "Green" :: Color
Green
```

Show Read **Eq** Ord Bounded Enum

```haskell
data Color = Red | Green | Blue
```

(def)

```haskell
class Eq a where
  (==) :: a -> a -> Bool

  (/=) :: a -> a -> Bool
```

(Haskell)

Show Read **Eq** Ord Bounded Enum

```haskell
data Color = Red | Green | Blue
```
(def)

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  (==) a b = not $ a /= b
  (/=) :: a -> a -> Bool
  (/=) a b = not $ a == b
```
(Haskell)

Show Read **Eq** Ord Bounded Enum

```haskell
data Color = Red | Green | Blue
```
(def)

```haskell
class Eq a where
  (==) :: a -> a -> Bool
  a == b = not $ a /= b
  (/=) :: a -> a -> Bool
  a /= b = not $ a == b
```
(Haskell)

Quick boring tour...

Show Read **Eq** Ord Bounded Enum

```haskell
data Color = Red | Green | Blue
```
(def)

```haskell
instance Eq Color where
  Red   == Red   = True
  Green == Green = True
  Blue  == Blue  = True
  _     == _     = False
```
(Haskell)

Quick boring tour...

Show Read Eq **Ord** Bounded Enum

```haskell
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<=)    :: a -> a -> Bool
  (>=)    :: a -> a -> Bool
  (<)     :: a -> a -> Bool
  (>)     :: a -> a -> Bool
  max     :: a -> a -> a
  min     :: a -> a -> a
```

(Haskell)

g2g

Quick boring tour...

Show Read Eq **Ord** Bounded Enum

```haskell
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<=)    :: a -> a -> Bool
```
(Haskell)

Quick boring tour...

Show Read Eq Ord **Bounded** Enum

```haskell
class Bounded a where
  minBound :: a
  maxBound :: a
```
(Haskell)

Quick boring tour...

Show Read Eq Ord Bounded **Enum**

```haskell
class Enum a where
  succ           :: a -> a
  pred           :: a -> a
  toEnum         :: Int -> a
  fromEnum       :: a -> Int
  enumFrom       :: a -> [a]
  enumFromThen   :: a -> a -> [a]
  enumFromTo     :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
```

(Haskell)

g2g

Deriving

```haskell
data Color = Red | Green | Blue
```

(Haskell)

Deriving

```haskell
data Color = Red | Green | Blue
  deriving ( Show
           , Read
           , Eq
           , Ord
           , Bounded
           , Enum
           )
```

(Haskell)

Codelab :: Section 1

directory : `01_mastermind/`

change : `src/Color.hs`

(replace `codelab` with implementation)

check with : `make ARGS="check 1"`

Codelab :: Section 2

directory : `01_mastermind/`

change : `src/ColorMap.hs`

(replace `codelab` with implementation)

check with : `make ARGS="check 2"`

g2g

Contexts / wrappers

A → value

Contexts / wrappers

A	→	value
Maybe A	→	optional value

Contexts / wrappers

A	→	value
Maybe A	→	optional value
List A	→	repeated value

Contexts / wrappers

A	→	value
Maybe A	→	optional value
List A	→	repeated value
IO A	→	impure value

g2g

Contexts / wrappers

A	→	value
Maybe A	→	optional value
List A	→	repeated value
IO A	→	impure value
C A	→	”contextual” value

Similarities

- Wrapping is trivial

Similarities

- Wrapping is trivial

```
wrap x = [x]
```

Similarities

- Wrapping is trivial

```
wrap x = [x]
wrap x = Just x
```

Similarities

- Wrapping is trivial

```
wrap x = [x]
wrap x = Just x
wrap x = pure x
```

Similarities

- Wrapping is trivial
- Unwrapping is non-trivial / destructive / impossible

Similarities

- Wrapping is trivial
- Unwrapping is non-trivial / destructive / impossible

```
unwrap Nothing = ???
```

Similarities

- Wrapping is trivial
- Unwrapping is non-trivial / destructive / impossible

```
unwrap Nothing = ???
unwrap [1,2,3] = ???
```

Similarities

- Wrapping is trivial
- Unwrapping is non-trivial / destructive / impossible

```
unwrap Nothing = ???
unwrap [1,2,3] = ???
unwrap getLine = NOPE
```

Similarities

- Wrapping is trivial
- Unwrapping is non-trivial / destructive / impossible
- What we want: to apply functions in the context

Context functions

Three standard functions to deal with contexts

Three standard functions to deal with contexts

```
fmap ::    (a -> b) -> c a -> c b
```

Three standard functions to deal with contexts

```
fmap ::    (a -> b) -> c a -> c b

ap   :: c (a -> b) -> c a -> c b
```

Three standard functions to deal with contexts

```
fmap ::    (a -> b) -> c a -> c b

ap   :: c (a -> b) -> c a -> c b

bind :: (a -> c b) -> c a -> c b
```

- fmap is like map...

- fmap is like map...

```haskell
map :: (a -> b) ->   [a] ->   [b]
```

(Haskell)

fmap

- fmap is like map...
- but generalised to all contexts

```haskell
 map :: (a -> b) ->    [a] ->    [b]
fmap :: (a -> b) -> c  a  -> c  b
```

(Haskell)

fmap

- fmap is like map...
- but generalised to all contexts

```haskell
fmap  show  [1, 2, 3] =
```

(Haskell)

fmap

- fmap is like map...
- but generalised to all contexts

```
fmap  show  [1, 2, 3] = ["1", "2", "3"]
```

(Haskell)

fmap

- fmap is like map...
- but generalised to all contexts

```haskell
fmap  show  [1, 2, 3] = ["1", "2", "3"]
fmap  show  (Just 42) =
```

(Haskell)

fmap

- fmap is like map...
- but generalised to all contexts

```haskell
fmap  show  [1, 2, 3] = ["1", "2", "3"]
fmap  show  (Just 42) = Just "42"
```

(Haskell)

fmap

- fmap is like map...
- but generalised to all contexts

```haskell
fmap  show  [1, 2, 3] = ["1", "2", "3"]
fmap  show  (Just 42) = Just "42"
fmap  show   Nothing  =
```

(Haskell)

fmap

- fmap is like map...
- but generalised to all contexts

```haskell
fmap  show  [1, 2, 3] = ["1", "2", "3"]
fmap  show  (Just 42) = Just "42"
fmap  show   Nothing  = Nothing
```

(Haskell)

fmap

- fmap is like map...
- but generalised to all contexts

```haskell
fmap  show   [1, 2, 3] = ["1", "2", "3"]
fmap  show   (Just 42) = Just "42"
fmap  show    Nothing  = Nothing
fmap length   getLine  =
```

(Haskell)

fmap

- fmap is like map...
- but generalised to all contexts

```haskell
fmap  show   [1, 2, 3] = ["1", "2", "3"]
fmap  show   (Just 42) = Just "42"
fmap  show    Nothing  = Nothing
fmap length   getLine  = ¯\_(ツ)_/¯
```

(Haskell)

fmap

- fmap is like map...
- but generalised to all contexts

```haskell
fmap  show   [1, 2, 3] = ["1", "2", "3"]
fmap  show   (Just 42) = Just "42"
fmap  show    Nothing  = Nothing
fmap length   getLine  = ¯\_(ツ)_/¯   --  IO Int
```

(Haskell)

g2g

fmap

- fmap is like map...
- but generalised to all contexts
- and it also has an operator version

```haskell
show     <$> [1, 2, 3] = ["1", "2", "3"]
show     <$> (Just 42) = Just "42"
show     <$>  Nothing   = Nothing
length   <$>  getLine   = ¯\_(ツ)_/¯    --  IO Int
```

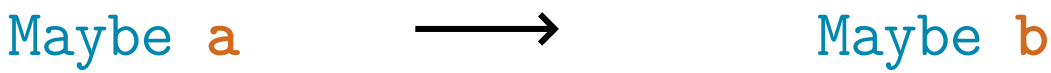
(Haskell)

g2g

When to use fmap

- We have one or more “wrapped” values
- Want to apply a function uniformly for all values

```
f :: a -> b
```



When to use fmap

- We have one or more “wrapped” values
- Want to apply a function uniformly for all values

```
f :: a -> b
```

```
Maybe a      ⟶   Maybe b
Minutes a    ⟶   Minutes b
[a]          ⟶   [b]
BinTree a    ⟶   BinTree b
```

When to use fmap

- We have one or more “wrapped” values
- Want to apply a function uniformly for all values

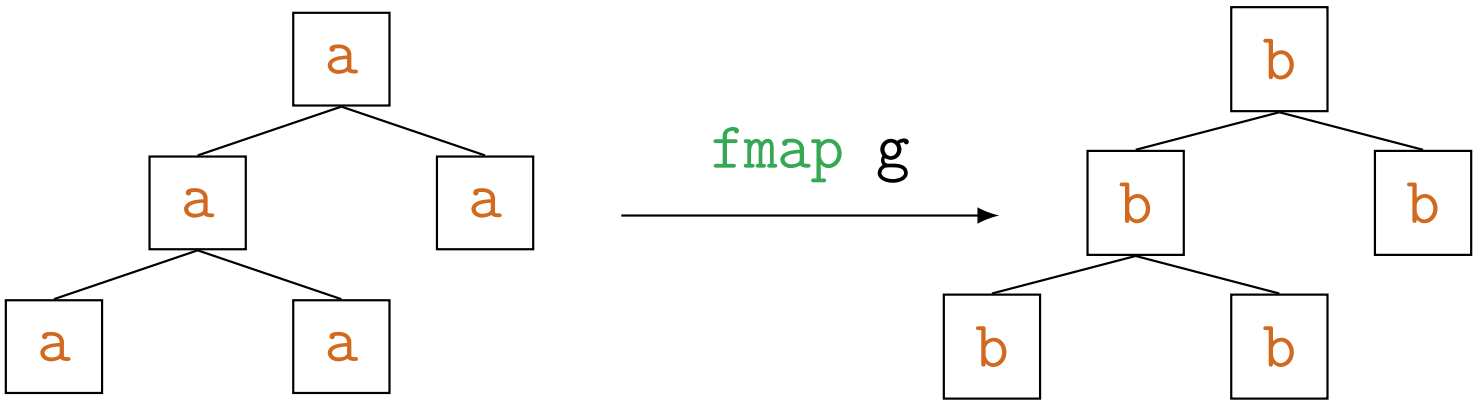
```
f :: a -> b
```

```
Maybe a       ⟶   Maybe b
Minutes a     ⟶   Minutes b
[a]           ⟶   [b]
BinTree a     ⟶   BinTree b
Either c a    ⟶   Either c b
```

“Lifting” a function

```
fmap :: (a -> b) ->  f a -> f b
fmap :: (a -> b) -> (f a -> f b)
```
(Haskell)

```
g :: a -> b
```

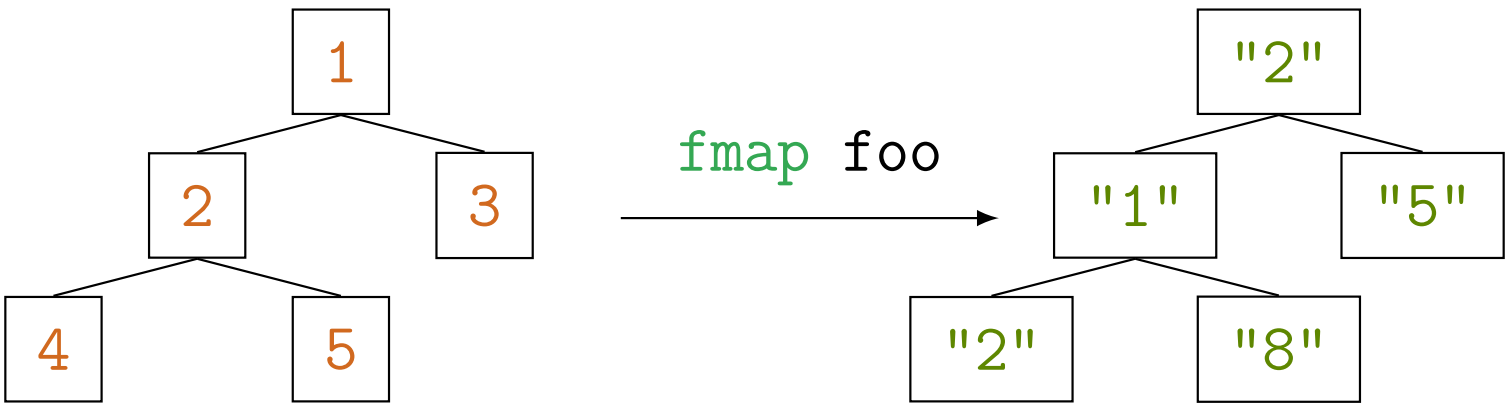


g2g

“Lifting” a function

```haskell
foo :: Int -> String
foo x
  | odd x = show $ (3 * x + 1) `div` 2
  | otherwise = show $ x `div` 2
```

(Haskell)



g2g

- what about functions with several arguments?

- what about functions with several arguments?

```haskell
fmap (+)          (Just 3) =
```

ap(ply)

- what about functions with several arguments?

```haskell
fmap (+)          (Just 3) = Just (3+)
```

(Haskell)

- what about functions with several arguments?

```haskell
fmap (+)          (Just 3) = Just (3+)
-- Just (3+) :: Maybe (Int -> Int)
```

(Haskell)

ap(ply)

- what about functions with several arguments?
- apply to the rescue!

```haskell
ap :: c (a -> b) -> c a -> c b

fmap (+)          (Just 3) = Just (3+)
ap (Just (3+)) (Just 39) =
```

(Haskell)

ap(ply)

- what about functions with several arguments?
- apply to the rescue!

```haskell
ap :: c (a -> b) -> c a -> c b

fmap (+)          (Just 3) = Just (3+)
ap (Just (3+)) (Just 39) = Just 42
```

(Haskell)

ap(ply)

- what about functions with several arguments?
- apply to the rescue!
- also defined as an operator

```haskell
(<*>) :: c (a -> b) -> c a -> c b

fmap (+)          (Just 3) = Just (3+)
Just (3+)  <*>     Just 39 = Just 42
```

(Haskell)

ap(ply)

- what about functions with several arguments?
- apply to the rescue!
- also defined as an operator

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

  (+)       <$>   Just 3  = Just (3+)
Just (3+)   <*>   Just 39 = Just 42
```

(Haskell)

ap(ply)

- what about functions with several arguments?
- apply to the rescue!
- also defined as an operator

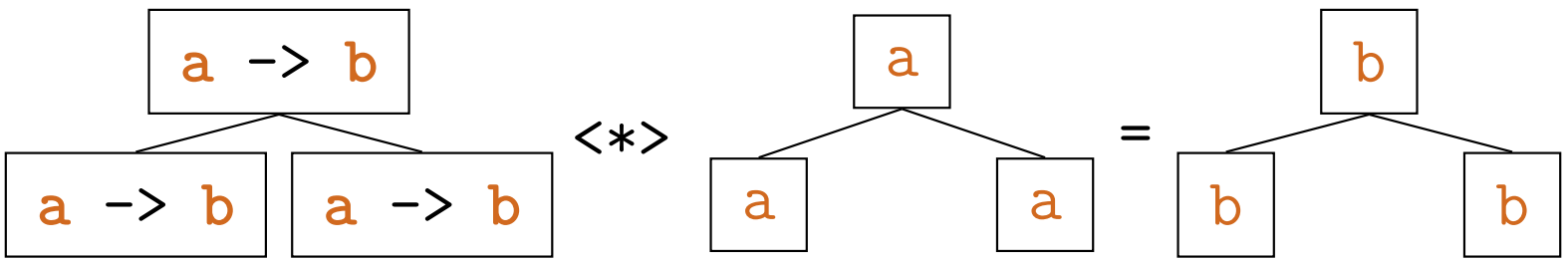
```haskell
    (+)         <$> Just 3  = Just (3+)
  Just (3+)     <*> Just 39 = Just 42

(+) <$> Just 3 <*> Just 39 = Just 42
```

(Haskell)

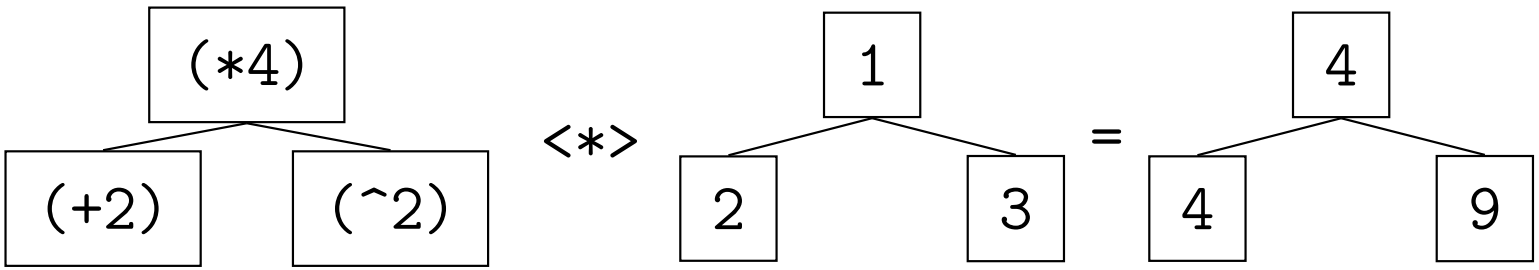
When to use ap

- “Wrapped” values and functions
- Want to apply functions to values



When to use ap

- “Wrapped” values and functions
- Want to apply functions to values



`Tree (Int -> Int)` `Tree Int` `Tree Int`

Using ap on lists

```haskell
data ZipList a = ZipList [a]
funs = ZipList [(+1), (^2)]
vals = ZipList [2, 3, 4]
funs <*> vals = ZipList [3, 9]
```

(Haskell)

Using ap on lists

```haskell
data ZipList a = ZipList [a]
funs = ZipList [(+1), (^2)]
vals = ZipList [2, 3, 4]
funs <*> vals = ZipList [3, 9]
```

(Haskell)

```haskell
data AllList a = AllList [a]
funs = AllList [(+1), (^2)]
vals = AllList [2, 3, 4]
funs <*> vals = AllList [3, 4, 5, 4, 9, 16]
```

(Haskell)

Using ap on lists

```haskell
newtype ZipList a = ZipList [a] -- newtype vs data
funs = ZipList [(+1), (^2)]
vals = ZipList [2, 3, 4]
funs <*> vals = ZipList [3, 9]
```

(Haskell)

```haskell
-- [a] for cartesian product
funs = [(+1), (^2)]
vals = [2, 3, 4]
funs <*> vals = [3, 4, 5, 4, 9, 16]
```

(Haskell)

Applicative style

- If we have a `f :: a -> b -> c`, how do we get a `c`?

Applicative style

- If we have a `f :: a -> b -> c`, how do we get a `c`?
- We need `x :: a` and `y :: b`

```haskell
f       x       y -- result :: c
```

Applicative style

- If we have a `f :: a -> b -> c`, how do we get a `c`?
- We need `x :: a` and `y :: b`
- What if we have `wx :: W a` and `wy :: W b`?
  (for some "wrapper" `W`)

```
f      x      y -- result :: c
```
(Haskell)

Applicative style

- If we have a `f :: a -> b -> c`, how do we get a `c`?
- We need `x :: a` and `y :: b`
- What if we have `wx :: W a` and `wy :: W b`?
  (for some "wrapper" `W`)
- `fmap` and `ap` give us `W c`!

```
f       x       y -- result ::    c
f <$> wx <*> wy -- result :: W c
```
(Haskell)

- what about chaining functions?

- what about chaining functions?

```haskell
div2 :: Int -> Maybe Int
```

(Haskell)

- what about chaining functions?

```haskell
div2 :: Int -> Maybe Int
div2 42 = Just 21
div2 21 = Nothing
```

(Haskell)

- what about chaining functions?

```
div2 :: Int -> Maybe Int
```

(given)

```
div4 :: Int -> Maybe Int
div4 x =


```

(Haskell)

- what about chaining functions?

```haskell
div2 :: Int -> Maybe Int
```

(given)

```haskell
div4 :: Int -> Maybe Int
div4 x = let y = div2 x
         in
```

(Haskell)

- what about chaining functions?

```haskell
div2 :: Int -> Maybe Int
```

(given)

```haskell
div4 :: Int -> Maybe Int
div4 x = let y = div2 x
         in fmap div2 y
```

(Haskell)

- what about chaining functions?

```
div2 :: Int -> Maybe Int
```

(given)

```
div4 :: Int -> Maybe Int
div4 x = let y = div2 x -- Maybe Int
         in fmap div2 y -- Maybe (Maybe Int)
```

(Haskell)

bind

- what about chaining functions?
- bind to the rescue!

```
bind :: (a -> c b) -> c a -> c b
```
(bind)

bind

- what about chaining functions?
- bind to the rescue!

```
bind :: (a -> c b) -> c a -> c b
```
(bind)

```
div2, div4 :: Int -> Maybe Int
div4 x = let y = div2 x -- Maybe Int
         in bind div2 y -- Maybe Int
// a = b = Int, c = Maybe
```
(Haskell)

bind

- what about chaining functions?
- bind to the rescue!

```
bind :: (a -> c b) -> c a -> c b
```

(bind)

```
div2, div4 :: Int -> Maybe Int
div4 x = bind div2 $ div2 x
```

(Haskell)

bind

- what about chaining functions?
- bind to the rescue!
- of course, also has an operator

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

(bind operator)

```
div2, div4 :: Int -> Maybe Int
div4 x = div2 x >>= div2
```

(Haskell)

bind

- what about chaining functions?
- bind to the rescue!
- of course, also has an operator

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

(bind operator)

```
div2, div4, div8 :: Int -> Maybe Int
div4 x = div2 x >>= div2
div8 x = div2 x >>= div2 >>= div2
```

(Haskell)

bind

- what about chaining functions?
- bind to the rescue!
- of course, also has an operator

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

(bind operator)

```
div2, div4, div8, div16 :: Int -> Maybe Int
div4  x = div2 x >>= div2
div8  x = div2 x >>= div2 >>= div2
div16 x = div2 x >>= div2 >>= div2 >>= div2
```

(Haskell)

bind - continued

```haskell
getUser        :: Id   -> Maybe User
getNextOfKin   :: User -> Maybe Id
getPhoneNumber :: User -> Maybe PhoneNumber
(>>=) :: c a -> (a -> c b) -> c b -- c == Maybe
```

(Haskell)

```haskell
getNextOfKinPhoneNumber :: Id -> Maybe PhoneNumber
getNextOfKinPhoneNumber uid =
```

(Haskell)

bind - continued

```haskell
getUser        :: Id   -> Maybe User
getNextOfKin   :: User -> Maybe Id
getPhoneNumber :: User -> Maybe PhoneNumber
(>>=) :: c a -> (a -> c b) -> c b -- c == Maybe
```

(Haskell)

```haskell
getNextOfKinPhoneNumber :: Id -> Maybe PhoneNumber
getNextOfKinPhoneNumber uid =
      getUser uid     -- Maybe User
```

(Haskell)

bind - continued

```haskell
getUser        :: Id   -> Maybe User
getNextOfKin   :: User -> Maybe Id
getPhoneNumber :: User -> Maybe PhoneNumber
(>>=) :: c a -> (a -> c b) -> c b -- c == Maybe
```

(Haskell)

```haskell
getNextOfKinPhoneNumber :: Id -> Maybe PhoneNumber
getNextOfKinPhoneNumber uid =
      getUser uid      -- Maybe User
  >>= getNextOfKin     -- Maybe Id
```

(Haskell)

bind - continued

```haskell
getUser        :: Id   -> Maybe User
getNextOfKin   :: User -> Maybe Id
getPhoneNumber :: User -> Maybe PhoneNumber
(>>=) :: c a -> (a -> c b) -> c b -- c == Maybe
```

(Haskell)

```haskell
getNextOfKinPhoneNumber :: Id -> Maybe PhoneNumber
getNextOfKinPhoneNumber uid =
      getUser uid      -- Maybe User
  >>= getNextOfKin     -- Maybe Id
  >>= getUser          -- Maybe User
```

(Haskell)

bind - continued

```haskell
getUser        :: Id   -> Maybe User
getNextOfKin   :: User -> Maybe Id
getPhoneNumber :: User -> Maybe PhoneNumber
(>>=) :: c a -> (a -> c b) -> c b -- c == Maybe
```

(Haskell)

```haskell
getNextOfKinPhoneNumber :: Id -> Maybe PhoneNumber
getNextOfKinPhoneNumber uid =
      getUser uid      -- Maybe User
  >>= getNextOfKin     -- Maybe Id
  >>= getUser          -- Maybe User
  >>= getPhoneNumber   -- Maybe User
```

(Haskell)

When to use bind

- “Collapse” the “wrapping”

When to use bind

- “Collapse” the “wrapping”
- “Decide” on “wrapping”

When to use bind

- “Collapse” the “wrapping”
- “Decide” on “wrapping”
- Parsers for context free languages vs context sensitive ones

The typeclasses

```
fmap ::     (a -> b) -> c a -> c b
ap   :: c (a -> b) -> c a -> c b
bind :: (a -> c b) -> c a -> c b
```

The typeclasses

```
fmap ::     (a -> b) -> c a -> c b
ap   :: c (a -> b) -> c a -> c b
bind :: (a -> c b) -> c a -> c b
```

But what about the typeclasses?

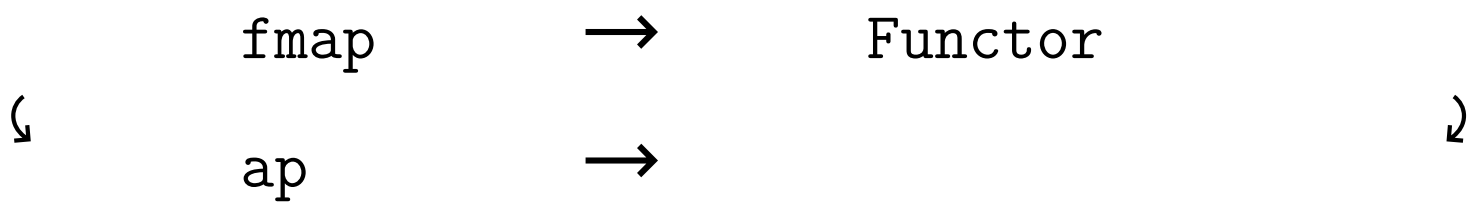
The typeclasses

`fmap` $\longrightarrow$

The typeclasses

`fmap` → `Functor`

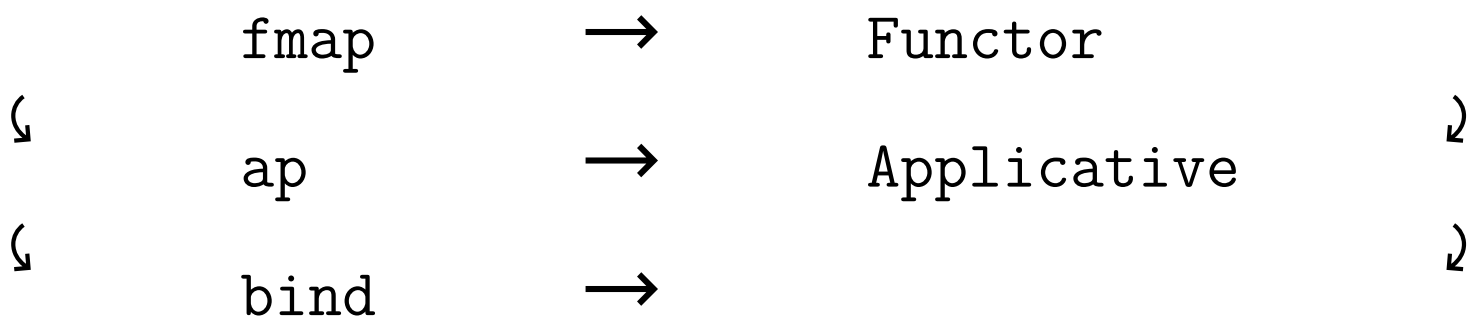
The typeclasses



The typeclasses

$$\left(\begin{array}{lcl} \texttt{fmap} & \longrightarrow & \texttt{Functor} \\ \texttt{ap} & \longrightarrow & \texttt{Applicative} \end{array}\right)$$

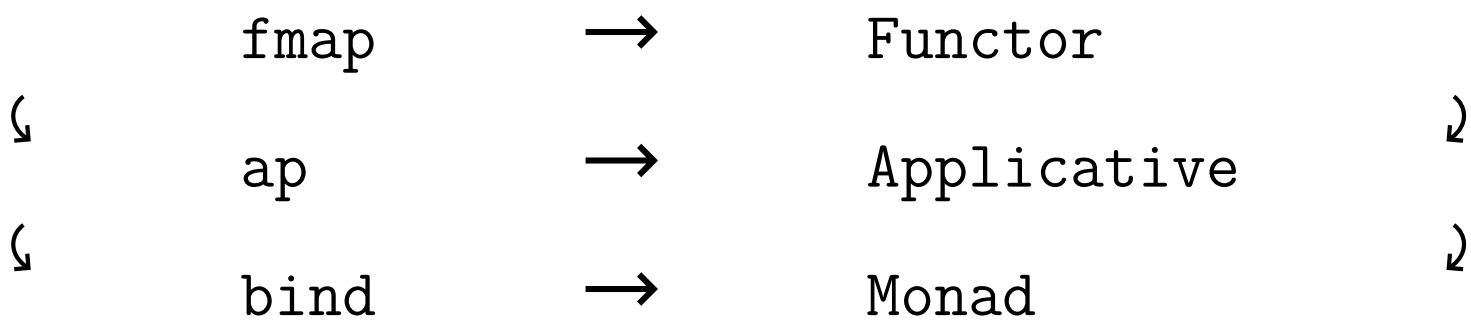
The typeclasses



MONAD

g2g

The typeclasses



g2g

Monad

```haskell
class Functor f where
  fmap   ::   (b -> a) -> f b -> f a
class Functor f => Applicative f where
  pure   ::                  a -> f a
  (<*>)  :: f (b -> a) -> f b -> f a
class Applicative m => Monad m where
  return ::                  a -> m a
  (>>=)  :: m b -> (b -> m a) -> m a
```

(Haskell)

The basic building blocks

g2g

Codelab :: Section 3

directory : `01_mastermind/`

change : `src/ErrorOr.hs`

(replace `codelab` with implementation)

check with : `make ARGS="check 3"`

g2g

We still don't know

- ~~how to extend our data types~~
- ~~how to express type constraints~~
- ~~how to chain contextual functions~~
- how to use IO

Using IO

```haskell
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String -> String -> IO ()
```

(Haskell)

Using IO

```haskell
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String -> String -> IO ()
```

(Haskell)

```haskell
main         :: IO ()
```

(Haskell)

Using IO

```haskell
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String -> String -> IO ()
```

(Haskell)

```haskell
main           :: IO ()
main =
```

(Haskell)

Using IO

```haskell
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String -> String -> IO ()
```

(Haskell)

```haskell
main           :: IO ()
main =
  getFirstName
```

(Haskell)

Using IO

```haskell
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String -> String -> IO ()
(>>=)        :: c a -> (a -> c b) -> c b
```

(Haskell)

```haskell
main           :: IO ()
main =
  getFirstName >>=
```

(Haskell)

Using IO

```haskell
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String -> String -> IO ()
(>>=)        :: c a -> (a -> c b) -> c b
```

(Haskell)

```haskell
main           :: IO ()
main =
  getFirstName >>= (\firstname -> ...)
```

(Haskell)

Using IO

```haskell
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String -> String -> IO ()
(>>=)        :: c a -> (a -> c b) -> c b
```

(Haskell)

```haskell
main           :: IO ()
main =
  getFirstName >>= (\firstname ->
    ...)
```

(Haskell)

Using IO

```haskell
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String -> String -> IO ()
(>>=)        :: c a -> (a -> c b) -> c b
```

(Haskell)

```haskell
main           :: IO ()
main =
  getFirstName >>= (\firstname ->
    getLastName ...)
```

(Haskell)

Using IO

```haskell
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String -> String -> IO ()
(>>=)        :: c a -> (a -> c b) -> c b
```

(Haskell)

```haskell
main           :: IO ()
main =
  getFirstName >>= (\firstname ->
    getLastName >>= (\lastname ->
      ...))
```

(Haskell)

Using IO

```haskell
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String -> String -> IO ()
(>>=)        :: c a -> (a -> c b) -> c b
```

(Haskell)

```haskell
main           :: IO ()
main =
  getFirstName >>= (\firstname ->
    getLastName >>= (\lastname ->
      greetUser firstname lastname))
```

(Haskell)

Using IO

```haskell
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String -> String -> IO ()
(>>=)        :: c a -> (a -> c b) -> c b
```

(Haskell)

```haskell
main           :: IO ()
main =
  getFirstName >>= \firstname ->
    getLastName >>= \lastname ->
      greetUser firstname lastname
```

(Haskell)

Do notation

```haskell
main :: IO ()
main =
  getFirstName >>= \firstname ->
    getLastName >>= \lastname ->
      greetUser firstname lastname
```

(Haskell)

Do notation

```
main :: IO ()
main =
  getFirstName      firstname
    getLastName      lastname
      greetUser firstname lastname
```

(important parts)

Do notation

```
main :: IO ()
main =
  getFirstName -> firstname
    getLastName -> lastname
      greetUser firstname lastname
```

(what we want)

Do notation

```
main :: IO ()
main =
  getFirstName -> firstname
  getLastName -> lastname
  greetUser firstname lastname
```

(align spaces)

g2g

Do notation

```haskell
main :: IO ()
main = do
  firstname <- getFirstName
  lastname <- getLastName
  greetUser firstname lastname
```

(Haskell)

Do notation

```haskell
main :: IO ()
main = do
  firstname <- getFirstName
  lastname <- getLastName
  greetUser firstname lastname
```

(Haskell)

"Haskell is the best imperative language"

Do notation

```haskell
getNextOfKinPhoneNumber :: Id -> Maybe PhoneNumber
getNextOfKinPhoneNumber uid =
       getUser uid
  >>= getNextOfKin
  >>= getUser
  >>= getPhoneNumber
```

(Haskell)

Do notation

```haskell
getNextOfKinPhoneNumber :: Id -> Maybe PhoneNumber
getNextOfKinPhoneNumber uid =
  getUser uid >>= \user ->
    getNextOfKin user >>= \nextOfKinId ->
      getUser nextOfKinId >>= \nextOfKinUser ->
        getPhoneNumber nextOfKinUser
```

(Haskell)

g2g

Do notation

```
getNextOfKinPhoneNumber :: Id -> Maybe PhoneNumber
getNextOfKinPhoneNumber uid =
  getUser uid       user
    getNextOfKin user       nextOfKinId
      getUser nextOfKinId       nextOfKinUser
        getPhoneNumber nextOfKinUser
```

(important parts)

g2g

Do notation

```haskell
getNextOfKinPhoneNumber :: Id -> Maybe PhoneNumber
getNextOfKinPhoneNumber uid = do
  user <- getUser uid
  nextOfKinId <- getNextOfKin user
  nextOfKinUser <- getUser nextOfKinId
  getPhoneNumber nextOfKinUser
```

(Haskell)

Do notation

```haskell
getNextOfKinPhoneNumber :: Id -> Maybe PhoneNumber
getNextOfKinPhoneNumber uid = do
  user <- getUser uid
  nextOfKinId <- getNextOfKin user
  nextOfKinUser <- getUser nextOfKinId
  getPhoneNumber nextOfKinUser
```

(Haskell)

```haskell
main :: IO ()
main = do
  firstname <- getFirstName
  lastname <- getLastName
  greetUser firstname lastname
```

(Haskell)

- Programmable semicolon

The power of monads

- Programmable semicolon
- Welcome to Haskell. We have:

The power of monads

- Programmable semicolon
- Welcome to Haskell. We have:
  - **optional chaining (using >>=/monads)**

The power of monads

- Programmable semicolon
- Welcome to Haskell. We have:
  - **optional chaining (using >>=/monads)**
  - **input/output (using >>=/monads)**

The power of monads

- Programmable semicolon
- Welcome to Haskell. We have:
  - **optional chaining (using >>=/monads)**
  - **input/output (using >>=/monads)**
  - **multithreading (using >>=/monads)**

The power of monads

- Programmable semicolon
- Welcome to Haskell. We have:
  - **optional chaining (using >>=/monads)**
  - **input/output (using >>=/monads)**
  - **multithreading (using >>=/monads)**
  - **error handling (using >>=/monads)**

The power of monads

- Programmable semicolon
- Welcome to Haskell. We have:
  - optional chaining (using >>=/monads)
  - input/output (using >>=/monads)
  - multithreading (using >>=/monads)
  - error handling (using >>=/monads)
  - mutable state (using >>=/monads)

The power of monads

- Programmable semicolon
- Welcome to Haskell. We have:
  - optional chaining (using >>=/monads)
  - input/output (using >>=/monads)
  - multithreading (using >>=/monads)
  - error handling (using >>=/monads)
  - mutable state (using >>=/monads)
  - list comprehension (using >>=/monads)

The power of monads

- Programmable semicolon
- Welcome to Haskell. We have:
  - optional chaining (using >>=/monads)
  - input/output (using >>=/monads)
  - multithreading (using >>=/monads)
  - error handling (using >>=/monads)
  - mutable state (using >>=/monads)
  - list comprehension (using >>=/monads)
  - randomness (using >>=/monads)

The power of monads

- Programmable semicolon
- Welcome to Haskell. We have:
  - optional chaining (using >>=/monads)
  - input/output (using >>=/monads)
  - multithreading (using >>=/monads)
  - error handling (using >>=/monads)
  - mutable state (using >>=/monads)
  - list comprehension (using >>=/monads)
  - randomness (using >>=/monads)
  - async/await (using >>=/monads)

Bonus: the power of monads

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

Bonus: the power of monads

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

```haskell
div2 :: Int -> Maybe Int
div2 x
  | even x    = Just (x `div` 2)
  | otherwise = Nothing
```

(Haskell)

g2g

Bonus: the power of monads

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

```haskell
div2 :: Int -> Maybe Int
div2 x
  | even x    = Just (x `div` 2)
  | otherwise = Nothing
```

(Haskell)

```
Prelude> mapM div2 [2,4,6,8]
Just [1,2,3,4]
Prelude> mapM div2 [2,3,4,5]
Nothing
```

(eval)

g2g

Bonus: the power of monads

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

```haskell
div2 :: Int -> Either String Int
div2 x
  | even x     = Right (x `div` 2)
  | otherwise = Left ("Odd: " ++ show x)
```

(Haskell)

g2g

Bonus: the power of monads

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

```haskell
div2 :: Int -> Either String Int
div2 x
  | even x     = Right (x `div` 2)
  | otherwise = Left ("Odd: " ++ show x)
```

(Haskell)

```
Prelude> mapM div2 [2,4,6,8]
Right [1,2,3,4]
Prelude> mapM div2 [2,3,4,5]
Left "Odd: 3"
```

(eval)

Bonus: the power of monads

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

```haskell
div2 :: Int -> IO Int
div2 x
  | even x    = return (x `div` 2)
  | otherwise = putStrLn ("Odd: " ++ show x) >>
                return 0
```

(Haskell)

g2g

Bonus: the power of monads

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

```haskell
div2 :: Int -> IO Int
div2 x
  | even x    = return (x `div` 2)
  | otherwise = putStrLn ("Odd: " ++ show x) >>
                return 0
```

(Haskell)

```
Prelude> mapM div2 [2,4,6,8]
[1,2,3,4]
Prelude> mapM div2 [2,3,4,5]
Odd: 3
Odd: 5
[1,0,2,0]
```

(eval)

Bonus: the power of monads

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

```haskell
div2 :: Int -> Writer [String] Int
div2 x
  | even x    = return (x `div` 2)
  | otherwise = tell ["Odd: " ++ show x] >>
                return 0
```

(Haskell)

g2g

Bonus: the power of monads

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

```haskell
div2 :: Int -> Writer [String] Int
div2 x
  | even x    = return (x `div` 2)
  | otherwise = tell ["Odd: " ++ show x] >>
                return 0
```

(Haskell)

```
Prelude> mapM div2 [2,4,6,8]
WriterT (Identity ([1,2,3,4],[]))
Prelude> mapM div2 [2,3,4,5]
WriterT (Identity ([1,0,2,0],["Odd: 3","Odd: 5"]))
```

(eval)

The end of the theoretical part!

Questions?

Links

- adit.io (functors, applicatives, and monads in pictures)
- dev.stephendiehl.com/hask/ (what I wish I knew)
- Forbes - Why Purely Functional Programming Is A Great Idea With A Misleading Name

Codelab :: Section 4

directory : `01_mastermind/`

change : `src/Code.hs`

(replace `codelab` with implementation)

check with : `make ARGS="check 4"`

g2g

Codelab :: Section 5

directory : `01_mastermind/`

change : `src/Do.hs`

(replace `codelab` with implementation)

check with : `make ARGS="check 5"`

Codelab :: The game

directory : `01_mastermind/`

play : `make ARGS=play`

solve (AI) : `make ARGS=solve`

The end!

Questions?