Haskell 102

@nicuveo

July 12, 2019

Goals

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

Roadmap

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

Roadmap

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

Checklist

- Haskell 101
- download the codelab
- apt-get install haskell-platform

Curried functions, partial application

```
f :: Int → Int → [Int]
```

Curried functions, partial application

```
f :: Int → ( Int → [Int] )
f ::       Int → Int → [Int]
```

Curried functions, partial application

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

Curried functions, partial application

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

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

Curried functions, partial application

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

Type constructors

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

Type constructors

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

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

Type constructors

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

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

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

Type constructors

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

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

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

“Deconstructors” and pattern matching

```
not :: Bool → Bool
not True  = False
not False = True
```

“Deconstructors” and pattern matching

```
not :: Bool → Bool
not True  = False
not False = True

magnitude :: Point → Double
magnitude (Point x y) = sqrt $ x^2 + y^2
```

“Deconstructors” and pattern matching

```haskell
not :: Bool → Bool
not True  = False
not False = True

magnitude :: Point → Double
magnitude (Point x y) = sqrt $ x^2 + y^2

length :: [a] → Int
length []     = 0
length (_:xs) = 1 + length xs
```

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?

```
sumI :: [Int] → Int
sumI = foldl' (+) 0
```

```
sumD :: [Double] → Double
sumD = foldl' (+) 0
```

Type parameters constraints?

```
sumI :: [Int] → Int
sumI = foldl' (+) 0
```

```
sumD :: [Double] → Double
sumD = foldl' (+) 0
```

```
sum :: [a] → a
sum = foldl' (+) 0
```

Type parameters constraints?

```
sumI :: [Int] → Int
sumI = foldl' (+) 0
```

```
sumD :: [Double] → Double
sumD = foldl' (+) 0
```

```
sum :: [a] → a
sum = foldl' (+) 0
```

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

Cascading context

```
getUser        :: Id   → Maybe User
getNextOfKin   :: User → Maybe Id
getPhoneNumber :: User → Maybe PhoneNumber
```

Cascading context

```
getUser          :: Id   → Maybe User
getNextOfKin     :: User → Maybe Id
getPhoneNumber   :: User → Maybe PhoneNumber

getNextOfKinPhoneNumber :: Id → Maybe PhoneNumber
```

Cascading context

```
getUser        :: Id   → Maybe User
getNextOfKin   :: User → Maybe Id
getPhoneNumber :: User → Maybe PhoneNumber

getNextOfKinPhoneNumber :: Id → Maybe PhoneNumber
getNextOfKinPhoneNumber userId =
```

Cascading context

```
getUser        :: Id   → Maybe User
getNextOfKin   :: User → Maybe Id
getPhoneNumber :: User → Maybe PhoneNumber

getNextOfKinPhoneNumber :: Id → Maybe PhoneNumber
getNextOfKinPhoneNumber userId = case getUser userId of
```

Cascading context

```
getUser        :: Id   → Maybe User
getNextOfKin   :: User → Maybe Id
getPhoneNumber :: User → Maybe PhoneNumber

getNextOfKinPhoneNumber :: Id → Maybe PhoneNumber
getNextOfKinPhoneNumber userId = case getUser userId of
    Nothing   → Nothing
```

Cascading context

```
getUser        :: Id   → Maybe User
getNextOfKin   :: User → Maybe Id
getPhoneNumber :: User → Maybe PhoneNumber

getNextOfKinPhoneNumber :: Id → Maybe PhoneNumber
getNextOfKinPhoneNumber userId = case getUser userId of
    Nothing   → Nothing
    Just user →
```

Cascading context

```
getUser        :: Id   → Maybe User
getNextOfKin   :: User → Maybe Id
getPhoneNumber :: User → Maybe PhoneNumber

getNextOfKinPhoneNumber :: Id → Maybe PhoneNumber
getNextOfKinPhoneNumber userId = case getUser userId of
    Nothing   → Nothing
    Just user → case getNextOfKin user of
```

Cascading context

```
getUser        :: Id   → Maybe User
getNextOfKin   :: User → Maybe Id
getPhoneNumber :: User → Maybe PhoneNumber

getNextOfKinPhoneNumber :: Id → Maybe PhoneNumber
getNextOfKinPhoneNumber userId = case getUser userId of
    Nothing   → Nothing
    Just user → case getNextOfKin user of
        Nothing          → Nothing
```

Cascading context

```
getUser          :: Id   → Maybe User
getNextOfKin     :: User → Maybe Id
getPhoneNumber   :: User → Maybe PhoneNumber

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

Cascading context

```
getUser        :: Id   → Maybe User
getNextOfKin   :: User → Maybe Id
getPhoneNumber :: User → Maybe PhoneNumber

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
```

Cascading context

```
getUser          :: Id   → Maybe User
getNextOfKin     :: User → Maybe Id
getPhoneNumber   :: User → Maybe PhoneNumber

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
```

Cascading context

```
getUser        :: Id   → Maybe User
getNextOfKin   :: User → Maybe Id
getPhoneNumber :: User → Maybe PhoneNumber

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 →
```

Cascading context

```
getUser        :: Id   → Maybe User
getNextOfKin   :: User → Maybe Id
getPhoneNumber :: User → Maybe PhoneNumber

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
```

- Can't apply regular functions on IO values

- Can't apply regular functions on IO values

```
$ ghci
>
```

- Can't apply regular functions on IO values

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

- Can't apply regular functions on IO values

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

- Can't apply regular functions on IO values

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

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

IO

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

IO is impure

IO

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

IO

- Can't apply regular functions on 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

```
class Show a where
    show :: a → String
```

Typeclasses

```
class Show a where
    show :: a → String

data Color = Red | Green | Blue
```

Typeclasses

```haskell
class Show a where
    show :: a → String

data Color = Red | Green | Blue

instance Show Color where
    show Red   = "Red"
    show Green = "Green"
    show Blue  = "Blue"
```

Constraints

```
show :: Show a ⇒ a → String
```

Constraints

```
show :: Show a ⇒ a → String
sum  :: Num a  ⇒ [a] → a
```

Constraints

```
show :: Show a ⇒ a → String
sum  :: Num a  ⇒ [a] → a
(==) :: Eq a   ⇒ a → a → Bool
```

Constraints

```
show :: Show a ⇒ a → String
sum  :: Num a  ⇒ [a] → a
(==) :: Eq a   ⇒ a → a → Bool

instance Show (Maybe a) where
```

Constraints

```
show :: Show a ⇒ a → String
sum  :: Num a  ⇒ [a] → a
(==) :: Eq a   ⇒ a → a → Bool

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

Constraints

```
show :: Show a ⇒ a → String
sum  :: Num a  ⇒ [a] → a
(==) :: Eq a   ⇒ a → a → Bool

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

Constraints

```
show :: Show a ⇒ a → String
sum  :: Num a  ⇒ [a] → a
(==) :: Eq a   ⇒ a → a → Bool

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

Show Read Eq Ord Bounded Enum

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

Show Read Eq Ord Bounded Enum

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

```
> show Blue
```

Show Read Eq Ord Bounded Enum

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

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

Show Read Eq Ord Bounded Enum

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

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

Show Read Eq Ord Bounded Enum

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

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

Show Read **Eq** Ord Bounded Enum

```
data Color = Red | Green | Blue

class Eq a where
    (==) :: a → a → Bool
    (/=) :: a → a → Bool
```

Show Read **Eq** Ord Bounded Enum

```
data Color = Red | Green | Blue

class Eq a where
    (==) :: a → a → Bool
    (/=) :: a → a → Bool
    a == b = not $ a /= b
    a /= b = not $ a == b
```

Show Read **Eq** Ord Bounded Enum

```
data Color = Red | Green | Blue

instance Eq Color where
    Red   == Red   = True
    Green == Green = True
    Blue  == Blue  = True
    _     == _     = False
```

Show Read Eq **Ord** Bounded Enum

```
data Color = Red | Green | Blue

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
```

Show Read Eq **Ord** Bounded Enum

```
data Color = Red | Green | Blue

class Eq a ⇒ Ord a where
    compare :: a → a → Ordering
    (<=)    :: a → a → Bool
```

Show Read Eq Ord **Bounded** Enum

```
data Color = Red | Green | Blue

class Bounded a where
    minBound :: a
    maxBound :: a
```

Show Read Eq Ord Bounded **Enum**

```
data Color = Red | Green | Blue

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

Deriving

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

Deriving

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

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

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

Contexts / wrappers

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

Similarities

- Wrapping is trivial

- Wrapping is trivial

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

- Wrapping is trivial

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

- Wrapping is trivial

```
wrap x = [x]
wrap x = Just x
wrap x = return 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...

```
map :: (a → b) → [a] → [b]
```

fmap

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

```
fmap :: (a → b) → c a → c b
```

fmap

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

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

fmap

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

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

fmap

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

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

fmap

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

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

fmap

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

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

fmap

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

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

fmap

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

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

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

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

fmap

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

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

fmap

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

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

fmap

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

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

- what about functions with several arguments?

- what about functions with several arguments?

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

- what about functions with several arguments?

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

- what about functions with several arguments?

```
fmap (+) (Just 3) = Just (3+)
Just (3+) :: Maybe (Int → Int)
```

ap(ply)

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

```
ap :: c (a → b) → c a → c b

fmap (+) (Just 3) = Just (3+)
Just (3+) :: Maybe (Int → Int)
```

ap(ply)

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

```
ap :: c (a → b) → c a → c b

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

ap(ply)

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

```
ap :: c (a → b) → c a → c b

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

ap(ply)

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

```
ap :: c (a → b) → c a → c b

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

ap(ply)

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

```
ap :: c (a → b) → c a → c b

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

ap(ply)

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

```
ap :: c (a → b) → c a → c b

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

ap(ply)

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

```
ap :: c (a → b) → c a → c b

(+)
```

ap(ply)

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

```
ap :: c (a → b) → c a → c b

(+) <$> Just 3
```

ap(ply)

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

```
ap :: c (a → b) → c a → c b

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

ap(ply)

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

```
ap :: c (a → b) → c a → c b

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

- what about chaining functions?

- what about chaining functions?

```
div2 :: Int → Maybe Int
```

- what about chaining functions?

```
div2 :: Int → Maybe Int

div2 42 = Just 21
```

- what about chaining functions?

```
div2 :: Int → Maybe Int

div2 42 = Just 21
div2 21 = Nothing
```

- what about chaining functions?

```
div2 :: Int → Maybe Int

div4 :: Int → Maybe Int
```

- what about chaining functions?

```
div2 :: Int → Maybe Int


div4 :: Int → Maybe Int
div4 x =
```

- what about chaining functions?

```
div2 :: Int → Maybe Int
```

```
div4 :: Int → Maybe Int
div4 x = let y = div2 x
         in
```

- what about chaining functions?

```
div2 :: Int → Maybe Int
```

```
div4 :: Int → Maybe Int
div4 x = let y = div2 x
         in fmap div2 y
```

- what about chaining functions?

```
div2 :: Int → Maybe Int


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

bind

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

```
div2 :: Int → Maybe Int
bind :: (a → c b) → c a → c b

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

bind

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

```
div2 :: Int → Maybe Int
bind :: (a → c b) → c a → c b

div4 :: Int → Maybe Int
div4 x = let y = div2 x -- Maybe Int
         in bind div2 y -- Maybe Int
```

bind

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

```
div2 :: Int → Maybe Int
bind :: (a → c b) → c a → c b

div4 :: Int → Maybe Int
div4 x = bind div2 $ div2 x
```

bind

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

```
div2 :: Int → Maybe Int
(>>=) :: c a → (a → c b) → c b

div4 :: Int → Maybe Int
div4 x = bind div2 $ div2 x
```

bind

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

```
div2 :: Int → Maybe Int
(>>=) :: c a → (a → c b) → c b

div4 :: Int → Maybe Int
div4 x = div2 x >>= div2
```

bind

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

```
div2 :: Int → Maybe Int
(>>=) :: c a → (a → c b) → c b

div8 :: Int → Maybe Int
div8 x = div2 x >>= div2 >>= div2
```

bind

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

```
div2 :: Int → Maybe Int
(>>=) :: c a → (a → c b) → c b

div16 :: Int → Maybe Int
div16 x = div2 x >>= div2 >>= div2 >>= div2
```

bind - continued

```
getUser        :: Id   → Maybe User
getNextOfKin   :: User → Maybe Id
getPhoneNumber :: User → Maybe PhoneNumber
```

bind - continued

```
getUser          :: Id   → Maybe User
getNextOfKin     :: User → Maybe Id
getPhoneNumber   :: User → Maybe PhoneNumber
```

```
getNextOfKinPhoneNumber :: Id → Maybe PhoneNumber
```

bind - continued

```
getUser          :: Id   → Maybe User
getNextOfKin     :: User → Maybe Id
getPhoneNumber   :: User → Maybe PhoneNumber
bind             :: (a → c b) → c a → c b

getNextOfKinPhoneNumber :: Id → Maybe PhoneNumber
```

bind - continued

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

getNextOfKinPhoneNumber :: Id → Maybe PhoneNumber
```

bind - continued

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

getNextOfKinPhoneNumber :: Id → Maybe PhoneNumber
getNextOfKinPhoneNumber uid =
```

bind - continued

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

getNextOfKinPhoneNumber :: Id → Maybe PhoneNumber
getNextOfKinPhoneNumber uid =
      getUser uid     -- Maybe User
```

bind - continued

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

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

bind - continued

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

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

bind - continued

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

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

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?

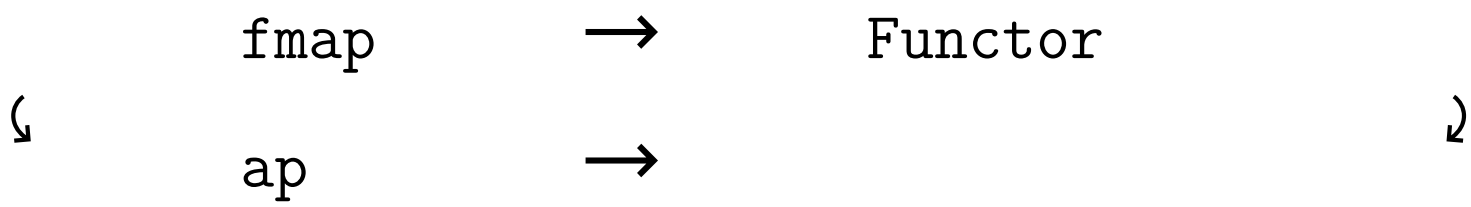
The typeclasses

`fmap` $\longrightarrow$

The typeclasses

`fmap` → `Functor`

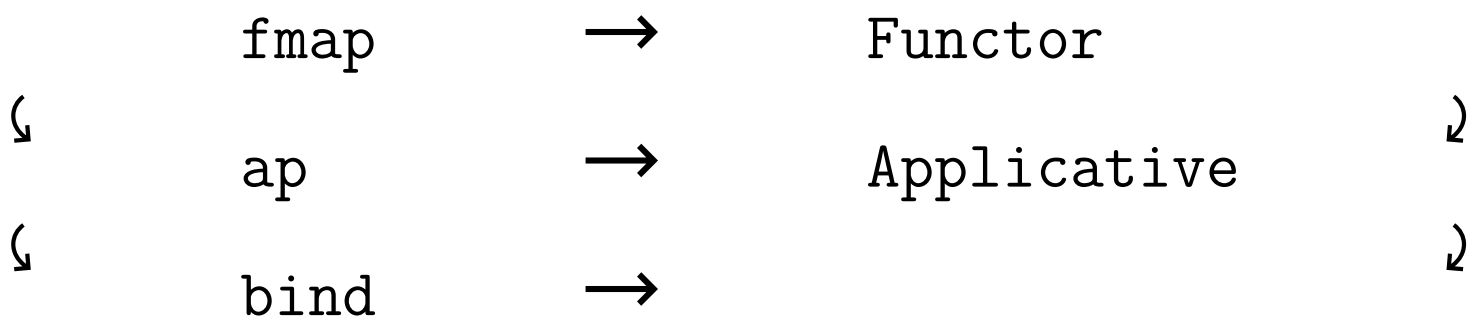
The typeclasses



The typeclasses

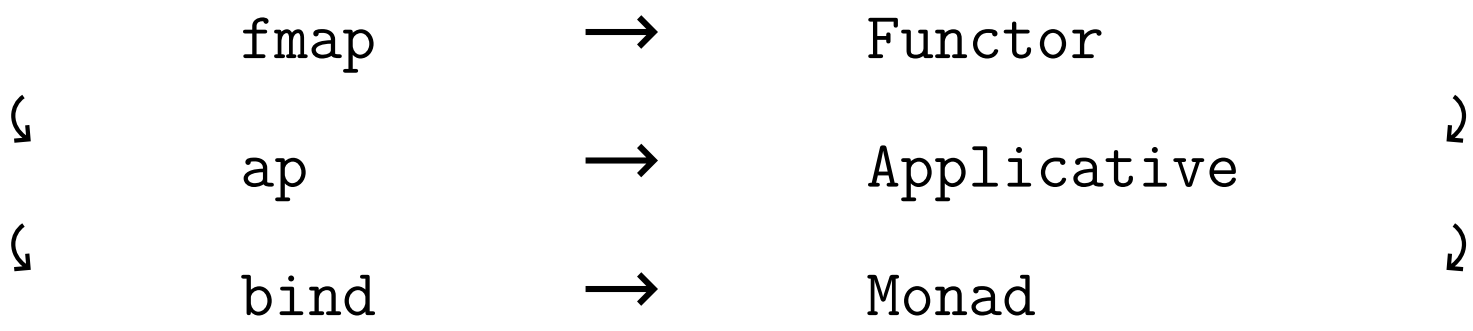
$$\left(\begin{array}{c}\texttt{fmap} \\ \texttt{ap}\end{array}\right) \longrightarrow \texttt{Applicative}$$

The typeclasses



MONAD

The typeclasses



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

Monad

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

The basic building block

Monad

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

(>>)       :: m a → m b → m b
(>=>)      :: (a → m b) → (b → m c) → (a → m c)
```

Powerful abstractions!

Monad

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

(>>)       :: m a → m b → m b
(>=>)      :: (a → m b) → (b → m c) → (a → m c)
sequence :: (Traversable t) ⇒ t (m a) → m (t a)
```

...that could use some sugar

Do notation

```
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String → String → IO ()
main         :: IO ()

main =
```

Do notation

```
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String → String → IO ()
main         :: IO ()

main =
    getFirstName
```

Do notation

```
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String → String → IO ()
main         :: IO ()

main =
    getFirstName >>=
```

Do notation

```
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String → String → IO ()
main         :: IO ()

main =
    getFirstName >>= λ firstname →
```

Do notation

```
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String → String → IO ()
main         :: IO ()

main =
    getFirstName >>= λ firstname →
        getLastName
```

Do notation

```
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String → String → IO ()
main         :: IO ()

main =
    getFirstName >>= λ firstname →
        getLastName >>=
```

Do notation

```
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String → String → IO ()
main         :: IO ()

main =
    getFirstName >>= λ firstname →
        getLastName >>= λ lastname →
```

Do notation

```
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String → String → IO ()
main         :: IO ()

main =
    getFirstName >>= λ firstname →
        getLastName >>= λ lastname →
            greetUser firstname lastname
```

Do notation

```
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String → String → IO ()
main         :: IO ()

main =
    getFirstName       firstname
        getLastName       lastname
            greetUser firstname lastname
```

Do notation

```
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String → String → IO ()
main         :: IO ()

main =
    getFirstName →    firstname
        getLastName →    lastname
            greetUser firstname lastname
```

Do notation

```
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String → String → IO ()
main         :: IO ()

main =
    getFirstName >>= λ firstname →
        getLastName >>= λ lastname →
            greetUser firstname lastname

main =
```

Do notation

```
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String → String → IO ()
main         :: IO ()

main =
    getFirstName >>= λ firstname →
        getLastName >>= λ lastname →
            greetUser firstname lastname

main = do
```

Do notation

```
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String → String → IO ()
main         :: IO ()

main =
    getFirstName >>= λ firstname →
        getLastName >>= λ lastname →
            greetUser firstname lastname

main = do
    firstname ← getFirstName
```

Do notation

```
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String → String → IO ()
main         :: IO ()

main =
    getFirstName >>= λ firstname →
        getLastName >>= λ lastname →
            greetUser firstname lastname

main = do
    firstname ← getFirstName
    lastname  ← getLastName
```

Do notation

```
getFirstName :: IO String
getLastName  :: IO String
greetUser    :: String → String → IO ()
main         :: IO ()

main =
    getFirstName >>= λ firstname →
        getLastName >>= λ lastname →
            greetUser firstname lastname

main = do
    firstname ← getFirstName
    lastname  ← getLastName
    greetUser firstname lastname
```

Recap

- typeclasses to extend our data types
- typeclasses to express type constraints
- monadic operators to chain contextual functions
- do notation to use IO

Links

- adit.io (functors, applicatives, and monads in pictures)
- dev.stephendiehl.com/hask/ (what I wish I knew)