Class 4: More…

February 7

next week:

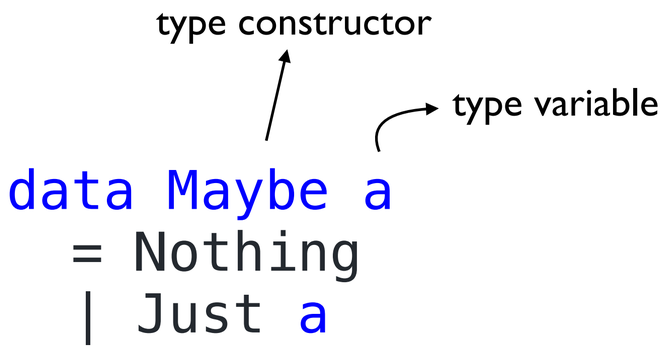
- lecture by guest speaker
- Jessica unavailable Monday and Tuesday

polymorphic data types

```
data FailableDouble
  = Failure
  | OK Double


data WeatherResult
  = Invalid
  | Valid Weather
```

suspiciously similar examples


type constructor
type variable
data Maybe a
= Nothing
| Just a

```haskell
toDouble :: Maybe Double -> Double
toDouble (Just n) = n
toDouble Nothing = 0


safeDiv :: Double -> Double -> Maybe Double
safeDiv _ 0 = Nothing
safeDiv m n = Just (m / n)
```

```
data List a
  = Nil
  | Cons a (List a)
```

lists

more polymorphic functions

```haskell
safeHead :: [a] -> Maybe a
safeHead (x : _) = Just x
safeHead [] = Nothing


head :: [a] -> a
head (x : _) = x
head [] = returns an error!
```

total vs. partial functions

(mapMaybe exercise)

review: syntax

```
greaterThan100 :: [Int] -> [Int]
greaterThan100 xs = filter (\x -> x > 100) xs
```

anonymous function

```haskell
greaterThan100 :: [Int] -> [Int]
greaterThan100 xs = filter (> 100) xs
```

operator section

function composition

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

dot operator

```
compose :: (b -> c) -> (a -> b) -> (a -> c)
compose f g =
```

dot operator

```
compose :: (b -> c) -> (a -> b) -> (a -> c)
compose f g = \x ->
```

dot operator

```haskell
compose :: (b -> c) -> (a -> b) -> (a -> c)
compose f g = \x -> f (g x)
```

dot operator

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(.) f g = \x -> f (g x)
```

dot operator

```haskell
myTest :: [Int] -> Bool
myTest = even . length . greaterThan100
```

old version

```haskell
myTest :: [Int] -> Bool
myTest xs = even (length (greaterThan100 xs))
```

partial application

```
f :: Int -> Int -> Int
```

```
f :: Int -> (Int -> Int)
```

subtlety:

function type arrows associate to the right
`W -> X -> Y -> Z` same as `W -> (X -> (Y -> Z))`

function applications associate to the left
`f 3 4 5` same as `((f 3) 4) 5`

```
f :: Int -> Int -> Int
f x y = 2 * x + y
```

currying

```haskell
f :: Int -> (Int -> Int)
f x = \y -> 2 * x + y
```

currying

```haskell
f :: (Int -> (Int -> Int))
f = \x -> (\y -> 2 * x + y)
```

currying

```haskell
plus :: Int -> Int -> Int
plus x y = x + y


plus3 :: ???
plus3 = plus 3
```

partial application

```
plus :: Int -> Int -> Int
plus x y = x + y


plus3 :: Int -> Int
plus3 = plus 3
```

partial application

```
greaterThan100 :: [Int] -> [Int]
greaterThan100 xs = filter (> 100) xs
```

partial application

```
greaterThan100 :: [Int] -> [Int]
greaterThan100 = filter (> 100)
```

partial application

wholemeal programming revisited

```haskell
foobar :: [Int] -> Int
foobar [] = 0
foobar (x : xs)
  | x > 3     = (7 * x + 2) + foobar xs
  | otherwise = foobar xs
```

```haskell
foobar :: [Int] -> Int
foobar [] = 0
foobar (x : xs)
  | x > 3     = (7 * x + 2) + foobar xs
  | otherwise = foobar xs
```

```haskell
foobar :: [Int] -> Int
foobar [] = 0
foobar (x : xs)
  | x > 3     = (7 * x + 2) + foobar xs
  | otherwise = foobar xs
```

```haskell
foobar :: [Int] -> Int
foobar [] = 0
foobar (x : xs)
  | x > 3     = (7 * x + 2) + foobar xs
  | otherwise = foobar xs
```

```haskell
foobar :: [Int] -> Int
foobar = sum . map (\x -> 7 * x + 2) . filter (> 3)
```

(and exercise)