Programming Paradigms

Lecture 6. Higher-order functions. Algebraic data types. Parametric polymorphism

Test N°5

See Moodle

Outline

- Recap
- Higher-order functions
- Algebraic data types
- Parametric polymorphism
- Parametric data types

Clarification: diamond operator (for Picture)

```
import CodeWorld

disk, square, myPicture :: Picture
disk        = colored red (solidCircle 5)
square      = solidRectangle 9 9

myPicture = disk <> square

main :: IO ()
main = drawingOf myPicture
```

Clarification: diamond operator (for Picture)

```haskell
import CodeWorld

disk, square, myPicture :: Picture
disk        = colored red (solidCircle 5)
square      = solidRectangle 9 9

myPicture = square <> disk

main :: IO ()
main = drawingOf myPicture
```

Higher-order functions: small example

```
addMod7 x y = (x + y) `mod` 7


twice f x = f (f x)

example :: Int
example = twice g 2
  where
    g x = addMod7 3 x
```

Higher-order functions: small example

```
addMod7 :: Int -> Int -> Int
addMod7 x y = (x + y) `mod` 7


twice f x = f (f x)

example :: Int
example = twice g 2
  where
    g x = addMod7 3 x
```

Higher-order functions: small example

```
addMod7 :: Int -> Int -> Int
addMod7 x y = (x + y) `mod` 7

twice ::  ? -> ? -> ?
twice f x = f (f x)

example :: Int
example = twice g 2
  where
    g x = addMod7 3 x
```

Higher-order functions: small example

```
addMod7 :: Int -> Int -> Int
addMod7 x y = (x + y) `mod` 7

twice :: ( ? -> ? ) -> ? -> ?
twice f x = f (f x)

example :: Int
example = twice g 2
  where
    g x = addMod7 3 x
```

Higher-order functions: small example

```
addMod7 :: Int -> Int -> Int
addMod7 x y = (x + y) `mod` 7

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

example :: Int
example = twice g 2
  where
    g x = addMod7 3 x
```

Higher-order functions: small example

```haskell
addMod7 :: Int -> Int -> Int
addMod7 x y = (x + y) `mod` 7

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

example :: Int
example = twice g 2
  where
    g x = addMod7 3 x

main :: IO ()
main = print example
```

Higher-order functions: lambda expression

```haskell
addMod7 :: Int -> Int -> Int
addMod7 x y = (x + y) `mod` 7

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

example :: Int
example = twice g 2
  where
    g = \x -> addMod7 3 x

main :: IO ()
main = print example
```

Higher-order functions: lambda expression

```haskell
addMod7 :: Int -> Int -> Int
addMod7 x y = (x + y) `mod` 7

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

example :: Int
example = twice g 2
  where
    g = \x -> addMod7 3 x

main :: IO ()
main = print example
```

λx . <expr>

Lambda abstraction in λ-calculus

Higher-order functions: lambda expression

```haskell
addMod7 :: Int -> Int -> Int
addMod7 x y = (x + y) `mod` 7

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

example :: Int
example = twice g 2
  where
    g = \x -> addMod7 3 x

main :: IO ()
main = print example
```

```
\x -> <expr>
```

Lambda expression
(anonymous function)

Higher-order functions: lambda expression

```haskell
addMod7 :: Int -> Int -> Int
addMod7 x y = (x + y) `mod` 7

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

example :: Int
example = twice (\x -> addMod7 3 x) 2



main :: IO ()
main = print example
```

Higher-order functions: lambda expression

```haskell
addMod7 :: Int -> Int -> Int
addMod7 = \x y -> (x + y) `mod` 7

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

example :: Int
example = twice (\x -> addMod7 3 x) 2



main :: IO ()
main = print example
```

Higher-order functions: lambda expression

```
addMod7 :: Int -> Int -> Int
addMod7 = \x y -> (x + y) `mod` 7

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

example :: Int
example = twice



main :: IO ()
main = print ex
```

```
\x1 … x3 -> <expr>
```

Lambda expression
(anonymous function of multiple arguments)

Higher-order functions: lambda expression

```
addMod7 :: Int -> Int -> Int
addMod7 = \x y -> (x + y) `mod` 7

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

example :: Int
example = twice

main :: IO ()
main = print ex
```

```
\x1 x2 x3 -> <expr>
```

Lambda expression
(anonymous function of multiple arguments)

Higher-order functions: lambda expression

```
addMod7 :: Int -> Int -> Int
addMod7 = \x y -> (x + y) `mod` 7

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

example :: Int
example = twice

main :: IO ()
main = print ex
```

```
\x1 x2 x3 -> <expr>
\x1 -> (\x2 -> (\x3 -> <expr>))
```

Lambda expression
(anonymous function of multiple arguments)

Higher-order functions: currying

```
addMod7 :: Int -> Int -> Int
addMod7 = \x -> \y -> (x + y) `mod` 7

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

example :: Int
example = twice (\x -> addMod7 3 x) 2



main :: IO ()
main = print example
```

Higher-order functions: currying

```haskell
addMod7 :: Int -> (Int -> Int)
addMod7 = \x -> \y -> (x + y) `mod` 7

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

example :: Int
example = twice (\x -> addMod7 3 x) 2



main :: IO ()
main = print example
```

Higher-order functions: currying

```haskell
addMod7 :: Int -> (Int -> Int)
addMod7 x = \y -> (x + y) `mod` 7

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

example :: Int
example = twice (\x -> addMod7 3 x) 2



main :: IO ()
main = print example
```

Higher-order functions: currying

```
addMod7 :: Int -> (Int -> Int)
addMod7 x = \y -> (x + y) `mod` 7

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

example :: Int
example = twice (\x -> (addMod7 3) x) 2



main :: IO ()
main = print example
```

Higher-order functions: currying

```haskell
addMod7 :: Int -> (Int -> Int)
addMod7 x = \y -> (x + y) `mod` 7

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

example :: Int
example = twice (addMod7 3) 2



main :: IO ()
main = print example
```

Higher-order functions: currying

```haskell
addMod7 :: Int -> (Int -> Int)
addMod7 x y = (x + y) `mod` 7

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

example :: Int
example = twice (addMod7 3) 2



main :: IO ()
main = print example
```

Higher-order functions: more examples

```
sumOf :: (Double -> Double) -> [Double] -> Double
sumOf f []       = 0.0
sumOf f (x:xs) = f x + sumOf f xs
```

Higher-order functions: more examples

```
sumOf :: (Double -> Double) -> [Double] -> Double
sumOf f []       = 0.0
sumOf f (x:xs) = f x + sumOf f xs

sum :: [Double] -> Double
sum xs =
```

Higher-order functions: more examples

```
sumOf :: (Double -> Double) -> [Double] -> Double
sumOf f []     = 0.0
sumOf f (x:xs) = f x + sumOf f xs

sum :: [Double] -> Double
sum xs = sumOf (       ) xs
```

Higher-order functions: more examples

```
sumOf :: (Double -> Double) -> [Double] -> Double
sumOf f []       = 0.0
sumOf f (x:xs) = f x + sumOf f xs

sum :: [Double] -> Double
sum xs = sumOf (\x ->   ) xs
```

Higher-order functions: more examples

```
sumOf :: (Double -> Double) -> [Double] -> Double
sumOf f []     = 0.0
sumOf f (x:xs) = f x + sumOf f xs

sum :: [Double] -> Double
sum xs = sumOf (\x -> x) xs
```

Higher-order functions: more examples

```
sumOf :: (Double -> Double) -> [Double] -> Double
sumOf f []       = 0.0
sumOf f (x:xs) = f x + sumOf f xs

sum :: [Double] -> Double
sum xs = sumOf id xs
```

Higher-order functions: more examples

```haskell
sumOf :: (Double -> Double) -> ([Double] -> Double)
sumOf f []     = 0.0
sumOf f (x:xs) = f x + sumOf f xs

sum :: [Double] -> Double
sum xs = sumOf id xs
```

Higher-order functions: more examples

```
sumOf :: (Double -> Double) -> ([Double] -> Double)
sumOf f []      = 0.0
sumOf f (x:xs) = f x + sumOf f xs

sum :: [Double] -> Double
sum xs = (sumOf id) xs
```

Higher-order functions: more examples

```
sumOf :: (Double -> Double) -> ([Double] -> Double)
sumOf f []       = 0.0
sumOf f (x:xs) = f x + sumOf f xs

sum :: [Double] -> Double
sum = sumOf id
```

Higher-order functions: more examples

```
sumOf :: (Double -> Double) -> ([Double] -> Double)
sumOf f []       = 0.0
sumOf f (x:xs) = f x + sumOf f xs

sum :: [Double] -> Double
sum = sumOf id

sumOfSquares :: [Double] -> Double
sumOfSquares = sumOf (\x -> x^2)
```

Higher-order functions: more examples

```
sumOf :: (Double -> Double) -> ([Double] -> Double)
sumOf f []       = 0.0
sumOf f (x:xs) = f x + sumOf f xs

sum :: [Double] -> Double
sum = sumOf id

sumOfSquares :: [Double] -> Double
sumOfSquares = sumOf (^2)
```

Higher-order functions: more examples

```
sumOf :: (Double -> Double) -> ([Double] -> Double)
sumOf f []     = 0.0
sumOf f (x:xs) = f x + sumOf f xs

sum :: [Double] -> Double
sum = sumOf id

sumOfSquares :: [Double] -> Double
sumOfSquares = sumOf (^2)

length :: [Double] -> Double
length = sumOf ?
```

Top-down development: standard deviation

$$\bar{x} := \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \textsf{stddev} := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})}$$

```
stddev :: [Double] -> Double
stddev xs =
```

Top-down development: standard deviation

$$\bar{x} := \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \text{stddev} := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})}$$

```
stddev :: [Double] -> Double
stddev xs = sqrt (bigSum / n)
  where
    ...
```

Top-down development: standard deviation

$$\bar{x} := \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \text{stddev} := \sqrt{\frac{1}{n}\boxed{\sum_{i=1}^{n}(x_i - \bar{x})}}$$

```
stddev :: [Double] -> Double
stddev xs = sqrt (bigSum / n)
  where
    ...
```

Top-down development: standard deviation

$$\bar{x} := \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \text{stddev} := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})}$$

```
stddev :: [Double] -> Double
stddev xs = sqrt (bigSum / n)
  where
    n = fromIntegral (length xs)
```

Top-down development: standard deviation

$$\bar{x} := \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \text{stddev} := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})}$$

```haskell
stddev :: [Double] -> Double
stddev xs = sqrt (bigSum / n)
  where
    n = fromIntegral (length xs)
    bigSum = sumOf (\x -> ?) xs
```

Top-down development: standard deviation

$$\bar{x} := \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \text{stddev} := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})}$$

```haskell
stddev :: [Double] -> Double
stddev xs = sqrt (bigSum / n)
  where
    n = fromIntegral (length xs)
    bigSum = sumOf (\x -> (x - mean)^2) xs
```

Top-down development: standard deviation

$$\bar{x} := \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \text{stddev} := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})}$$

```
stddev :: [Double] -> Double
stddev xs = sqrt (bigSum / n)
  where
    n = fromIntegral (length xs)
    bigSum = sumOf (\x -> (x - mean)^2) xs
    mean = sum xs / n
```

Algebraic data types: product types

```haskell
-- | A 2D vector.
data Vector = Vector Double Double
```

Name of the new type

Types of parameters of the value constructor

Name of the **value constructor**

Algebraic data types: enumeration types

```haskell
-- | One of the basic colors.
data Color = Red | Green | Blue
```

Name of the new type

Name of the **value constructor**

Algebraic data types: sum types (sum of products)

```haskell
-- | A result of some computation.
data BoolResult
  = Success Bool | Failure String
```

Name of the **value constructor**

Types of parameters of value constructors

Algebraic data types: sum types (sum of products)

```haskell
-- | A result of some computation.
data BoolResult
  = Success Bool
  | Failure String


parseAnswer :: String -> BoolResult
```

Algebraic data types: sum types (sum of products)

```
-- | A result of some computation.
data BoolResult
  = Success Bool
  | Failure String


parseAnswer :: String -> BoolResult
parseAnswer "yes" = Success True
```

Algebraic data types: sum types (sum of products)

```haskell
-- | A result of some computation.
data BoolResult
  = Success Bool
  | Failure String


parseAnswer :: String -> BoolResult
parseAnswer "yes" = Success True
parseAnswer "no"  = Success False
```

Algebraic data types: sum types (sum of products)

```haskell
-- | A result of some computation.
data BoolResult
  = Success Bool
  | Failure String


parseAnswer :: String -> BoolResult
parseAnswer "yes" = Success True
parseAnswer "no"  = Success False
parseAnswer input
  = Failure ("unrecognized input: " ++ input)
```

Algebraic data types: sum types (sum of products)

```haskell
-- | A basic shape.
data Shape
  = Circle Radius
  | Rectangle Width Height
```

Algebraic data types: sum types (sum of products)

```haskell
-- | A basic shape.
data Shape
  = Circle Radius
  | Rectangle Width Height

renderShape :: Shape -> Picture
```

Algebraic data types: sum types (sum of products)

```haskell
-- | A basic shape.
data Shape
  = Circle Radius
  | Rectangle Width Height

renderShape :: Shape -> Picture
renderShape (Circle r) = circle r
renderShape (Rectangle w h)
  = rectangle w h
```

Algebraic data types: sum types (sum of products)

```haskell
-- | An integer expression.
data IntExpr
  = Literal Int
  | Plus IntExpr IntExpr   -- e1 + e2
  | Mult IntExpr IntExpr   -- e1 * e2
```

Algebraic data types: sum types (sum of products)

```haskell
-- | An integer expression.
data IntExpr
  = Literal Int
  | Plus IntExpr IntExpr   -- e1 + e2
  | Mult IntExpr IntExpr   -- e1 * e2

-- (1 + 2) * 3
example :: IntExpr
example = Mult
  (Plus (Literal 1) (Literal 2))
  (Literal 3)
```

Algebraic data types: sum types (sum of products)

```haskell
-- | An integer expression.
data IntExpr
  = Literal Int
  | Plus IntExpr IntExpr   -- e1 + e2
  | Mult IntExpr IntExpr   -- e1 * e2

eval :: IntExpr -> Int
```

Algebraic data types: sum types (sum of products)

```haskell
-- | An integer expression.
data IntExpr
  = Literal Int
  | Plus IntExpr IntExpr   -- e1 + e2
  | Mult IntExpr IntExpr   -- e1 * e2

eval :: IntExpr -> Int
eval (Literal n)  = n
```

Algebraic data types: sum types (sum of products)

```
-- | An integer expression.
data IntExpr
  = Literal Int
  | Plus IntExpr IntExpr   -- e1 + e2
  | Mult IntExpr IntExpr   -- e1 * e2

eval :: IntExpr -> Int
eval (Literal n)   = n
eval (Plus e1 e2) = eval e1 + eval e2
```

Algebraic data types: sum types (sum of products)

```haskell
-- | An integer expression.
data IntExpr
  = Literal Int
  | Plus IntExpr IntExpr   -- e1 + e2
  | Mult IntExpr IntExpr   -- e1 * e2

eval :: IntExpr -> Int
eval (Literal n)   = n
eval (Plus e1 e2) = eval e1 + eval e2
eval (Mult e1 e2) = eval e1 * eval e2
```

Parametric polymorphism

```
addMod7 :: Int -> (Int -> Int)
addMod7 x y = (x + y) `mod` 7

twice :: (Int -> Int) -> Int -> Int
twice f x = f (f x)

example :: Int
example = twice (addMod7 3) 2



main :: IO ()
main = print example
```

Parametric polymorphism

```haskell
addMod7 :: Int -> (Int -> Int)
addMod7 x y = (x + y) `mod` 7


twice f x = f (f x)

example :: Int
example = twice (addMod7 3) 2



main :: IO ()
main = print example
```

Parametric polymorphism

```haskell
{-# OPTIONS_GHC -Wall #-}

addMod7 :: Int -> (Int -> Int)
addMod7 x y = (x + y) `mod` 7


twice f x = f (f x)

example :: Int
example = twice (addMod7 3) 2



main :: IO ()
main = print example
```

Parametric polymorphism

```
{-# OPTIONS_GHC -Wall #-}

addMod7 :: Int -> (Int -> Int)
addMod7 x y = (x + y) `mod` 7


twice f x = f (f x)

ex
ex
```

```
Line 3, Column 1-5: warning: [-Wmissing-signatures]
    Top-level binding with no type signature:
      twice :: (t -> t) -> t -> t
```

```
main :: IO ()
main = print example
```

Parametric polymorphism

```
addMod7 :: Int -> (Int -> Int)
addMod7 x y = (x + y) `mod` 7

twice :: (t -> t) -> t -> t
twice f x = f (f x)
```

Parametric polymorphism

```
addMod7 :: Int -> (Int -> Int)
addMod7 x y = (x + y) `mod` 7

twice :: (t -> t) -> t -> t
twice f x = f (f x)
```

For any type `t`,
`twice` has type `(t -> t) -> t -> t`

Parametric polymorphism

```
addMod7 :: Int -> (Int -> Int)
addMod7 x y = (x + y) `mod` 7

twice :: (t -> t) -> t -> t
twice f x = f (f x)
```

For any type `t`,
`twice` has type `(t -> t) -> t -> t`

```
example1 = twice (+1) 2              -- t = ?
```

Parametric polymorphism

```
addMod7 :: Int -> (Int -> Int)
addMod7 x y = (x + y) `mod` 7

twice :: (t -> t) -> t -> t
twice f x = f (f x)
```

For any type `t`,
`twice` has type `(t -> t) -> t -> t`

```
example1 = twice (+1) 2          -- t = Int
```

Parametric polymorphism

```haskell
addMod7 :: Int -> (Int -> Int)
addMod7 x y = (x + y) `mod` 7

twice :: (t -> t) -> t -> t
twice f x = f (f x)
```

For any type `t`,
`twice` has type `(t -> t) -> t -> t`

```haskell
example1 = twice (+1) 2          -- t = Int
example2 = twice (1:) [2, 3]     -- t = ?
```

Parametric polymorphism

```
addMod7 :: Int -> (Int -> Int)
addMod7 x y = (x + y) `mod` 7

twice :: (t -> t) -> t -> t
twice f x = f (f x)
```

For any type `t`,
`twice` has type `(t -> t) -> t -> t`

```
example1 = twice (+1) 2          -- t = Int
example2 = twice (1:) [2, 3]     -- t = [Int]
```

Parametric polymorphism

```
addMod7 :: Int -> (Int -> Int)
addMod7 x y = (x + y) `mod` 7

twice :: (t -> t) -> t -> t
twice f x = f (f x)
```

For any type `t`,
`twice` has type `(t -> t) -> t -> t`

```
example1 = twice (+1) 2          -- t = Int
example2 = twice (1:) [2, 3]     -- t = [Int]
example3 = twice twice (+1) 0    -- t = ?
```

Parametric polymorphism

```haskell
addMod7 :: Int -> (Int -> Int)
addMod7 x y = (x + y) `mod` 7

twice :: (t -> t) -> t -> t
twice f x = f (f x)
```

For any type `t`,
`twice` has type `(t -> t) -> t -> t`

```haskell
example1 = twice (+1) 2          -- t = Int
example2 = twice (1:) [2, 3]     -- t = [Int]
example3 = twice twice (+1) 0    -- t = Int -> Int
```

The `map` function

```
map :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x:xs) = f x : map f xs
```

```
example4 = map (+1) [1, 2, 3]
-- a = ?
-- b = ?
```

The `map` function

```haskell
map :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x:xs) = f x : map f xs


example4 = map (+1) [1, 2, 3]
-- a = Int
-- b = Int
```

The `map` function

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs


example5 = map length ["hello", "world"]
-- a = ?
-- b = ?
```

The `map` function

```haskell
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs


example5 = map length ["hello", "world"]
-- a = String
-- b = Int
```

The `map` function

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs


example6 = map (> 0) [4, -2, 1, 0]
-- a = ?
-- b = ?
```

The map function

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs


example6 = map (> 0) [4, -2, 1, 0]
-- a = Int
-- b = Bool
```

The map function

```
map :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x:xs) = f x : map f xs


example7 = map (\f -> f 3) [(+1), (^2)]
-- a = ?
-- b = ?
```

The `map` function

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs


example7 = map (\f -> f 3) [(+1), (^2)]
-- a = Int -> Int
-- b = Int
```

The `map` function

```
map :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x:xs) = f x : map f xs


example8 = map (\x -> [x]) []
-- a = ?
-- b = ?
```

The `map` function

```
map :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x:xs) = f x : map f xs
```

```
example8 = map (\x -> [x]) []
-- a = t
-- b = [t]
```

Parametric data types

```
data BoolResult = Success Bool | Failure String


parseAnswer :: String -> BoolResult
parseAnswer "yes" = Success True
parseAnswer "no"  = Success False
parseAnswer input
  = Failure ("unrecognized input: " ++ input)
```

Parametric data types

```
data Result a    = Success a    | Failure String


parseAnswer :: String -> BoolResult
parseAnswer "yes" = Success True
parseAnswer "no"  = Success False
parseAnswer input
  = Failure ("unrecognized input: " ++ input)
```

Parametric data types

```
data Result a    = Success a     | Failure String
```

Type constructor

Value constructors

Type variable
(parameter for type constructor)

Parametric data types

```
data Result a    = Success a     | Failure String


parseAnswer :: String -> BoolResult
parseAnswer "yes" = Success True
parseAnswer "no"  = Success False
parseAnswer input
  = Failure ("unrecognized input: " ++ input)
```

Parametric data types

```haskell
data Result a    = Success a     | Failure String
type BoolResult = Result Bool

parseAnswer :: String -> BoolResult
parseAnswer "yes" = Success True
parseAnswer "no"  = Success False
parseAnswer input
  = Failure ("unrecognized input: " ++ input)
```

Parametric data types

```
data Result a    = Success a     | Failure String


parseAnswer :: String -> Result Bool
parseAnswer "yes" = Success True
parseAnswer "no"  = Success False
parseAnswer input
  = Failure ("unrecognized input: " ++ input)
```

Parametric data types: Maybe

```
data Maybe a    = Nothing | Just a


parseAnswer :: String -> Maybe Bool
parseAnswer "yes" = Just True
parseAnswer "no"  = Just False
parseAnswer input = Nothing
```

Parametric data types: Maybe

```
data Maybe a    = Nothing | Just a

type Name     = String
data Grade    = A | B | C | D
data Student = Student Name Grade
```

Parametric data types: Maybe

```
data Maybe a    = Nothing | Just a

type Name     = String
data Grade    = A | B | C | D
data Student = Student Name Grade

gradeOf :: Name -> [Student] -> Maybe Grade
gradeOf = ?
```

Parametric data types: Maybe

```
data Maybe a    = Nothing | Just a

type Name     = String
data Grade    = A | B | C | D
data Student = Student Name Grade

gradeOf :: Name -> [Student] -> Maybe Grade
gradeOf _target [] = ?
gradeOf target (student : students) = ?
```

Parametric data types: Maybe

```
data Maybe a    = Nothing | Just a

type Name     = String
data Grade    = A | B | C | D
data Student = Student Name Grade

gradeOf :: Name -> [Student] -> Maybe Grade
gradeOf _target [] = Nothing
gradeOf target (student : students) = ?
```

Parametric data types: Maybe

```
data Maybe a    = Nothing | Just a

type Name     = String
data Grade    = A | B | C | D
data Student = Student Name Grade

gradeOf :: Name -> [Student] -> Maybe Grade
gradeOf _target [] = Nothing
gradeOf target (Student name grade : students) = ?
```

Parametric data types: Maybe

```
data Maybe a    = Nothing | Just a

type Name     = String
data Grade    = A | B | C | D
data Student = Student Name Grade

gradeOf :: Name -> [Student] -> Maybe Grade
gradeOf _target [] = Nothing
gradeOf target (Student name grade : students)
   | name == target = ?
   | otherwise      = ?
```

Parametric data types: Maybe

```
data Maybe a    = Nothing | Just a

type Name     = String
data Grade    = A | B | C | D
data Student = Student Name Grade

gradeOf :: Name -> [Student] -> Maybe Grade
gradeOf _target [] = Nothing
gradeOf target (Student name grade : students)
   | name == target = Just grade
   | otherwise      = ?
```

Parametric data types: Maybe

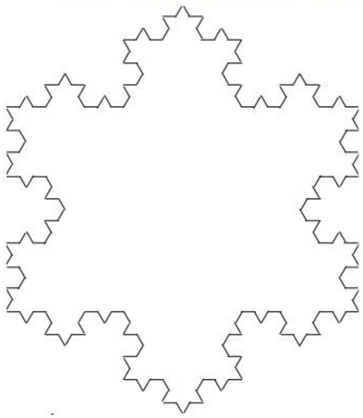
```haskell
data Maybe a    = Nothing | Just a

type Name     = String
data Grade    = A | B | C | D
data Student = Student Name Grade

gradeOf :: Name -> [Student] -> Maybe Grade
gradeOf _target [] = Nothing
gradeOf target (Student name grade : students)
   | name == target = Just grade
   | otherwise      = gradeOf target students
```

Homework (self-study)

1. Install **Haskell** https://www.haskell.org/downloads/
2. Read **Learn you a Haskell for Great Good** Chapters 2, 4, and 5 http://learnyouahaskell.com/chapters
3. Test yourself by implementing a program that renders a Koch snowflake of a given rank in Haskell on Code.World platform (https://code.world/haskell):



What was the most unclear part of the lecture for you?

See Moodle