# Лекция 8. Функции высшего порядка. Алгебраические типы. Параметрический полиморфизм.

Функциональное и логическое программирование (осень 2025)

---

Николай Кудасов

13 октября 2025

институт ispring

# Содержание

# Функции высшего порядка

# Функции высшего порядка (на примере)

```haskell
around :: String -> String -> String -> String
around l r s = l ++ s ++ r



after f g x = f (g x)

sayE :: String -> String
sayE s = after say exclaim s
  where
    say s' = around "«" "»" s'
    exclaim s' = s' ++ "!"

main :: IO ()
main = putStrLn (sayE "Привет")
```

# Функции высшего порядка (типы)

```haskell
around :: String -> String -> String -> String
around l r s = l ++ s ++ r

after :: (String -> String) -> (String -> String) -> String -> String
after f g x = f (g x)

sayE :: String -> String
sayE s = after say exclaim s
  where
    say s' = around "«" "»" s'
    exclaim s' = s' ++ "!"

main :: IO ()
main = putStrLn (sayE "Привет")
```

# Функции высшего порядка (анонимные функции)

```haskell
around :: String -> String -> String -> String
around l r s = l ++ s ++ r

after :: (String -> String) -> (String -> String) -> String -> String
after f g x = f (g x)

sayE :: String -> String
sayE s = after say exclaim s
  where
    say = \s' -> around "«" "»" s'
    exclaim = \s' -> s' ++ "!"

main :: IO ()
main = putStrLn (sayE "Привет")
```

# Функции высшего порядка (анонимные функции)

```haskell
around :: String -> String -> String -> String
around l r s = l ++ s ++ r

after :: (String -> String) -> (String -> String) -> String -> String
after f g x = f (g x)

sayE :: String -> String
sayE s = after
  (\s' -> around "«" "»" s')
  (\s' -> s' ++ "!")
  s

main :: IO ()
main = putStrLn (sayE "Привет")
```

# Функции высшего порядка (анонимные функции)

```haskell
around :: String -> String -> String -> String
around = \l r s -> l ++ s ++ r

after :: (String -> String) -> (String -> String) -> String -> String
after = \f g x -> f (g x)

sayE :: String -> String
sayE s = after
  (\s' -> around "«" "»" s')
  (\s' -> s' ++ "!")
  s

main :: IO ()
main = putStrLn (sayE "Привет")
```

# Функции высшего порядка (каррирование)

```haskell
around :: String -> String -> String -> String
around = \l -> \r -> \s -> l ++ s ++ r

after :: (String -> String) -> (String -> String) -> String -> String
after = \f -> \g -> \x -> f (g x)

sayE :: String -> String
sayE s = after
  (\s' -> around "«" "»" s')
  (\s' -> s' ++ "!")
  s

main :: IO ()
main = putStrLn (sayE "Привет")
```

# Функции высшего порядка (каррирование)

```haskell
around :: String -> (String -> (String -> String))
around = \l -> \r -> \s -> l ++ s ++ r

after :: (String -> String) -> ((String -> String) -> (String -> String))
after = \f -> \g -> \x -> f (g x)

sayE :: String -> String
sayE s = after
  (\s' -> around "«" "»" s')
  (\s' -> s' ++ "!")
  s

main :: IO ()
main = putStrLn (sayE "Привет")
```

# Функции высшего порядка (каррирование)

```haskell
around :: String -> String -> (String -> String)
around l r = \s -> l ++ s ++ r

after :: (String -> String) -> (String -> String) -> (String -> String)
after f g = \x -> f (g x)

sayE :: String -> String
sayE s = after
  (\s' -> around "«" "»" s')
  (\s' -> s' ++ "!")
  s

main :: IO ()
main = putStrLn (sayE "Привет")
```

# Функции высшего порядка (частичное применение)

```haskell
around :: String -> String -> (String -> String)
around l r = \s -> l ++ s ++ r

after :: (String -> String) -> (String -> String) -> (String -> String)
after f g = \x -> f (g x)

sayE :: String -> String
sayE s = (after
  (\s' -> (around "«" "»") s')
  (\s' -> s' ++ "!"))
  s

main :: IO ()
main = putStrLn (sayE "Привет")
```

# Функции высшего порядка (частичное применение)

```haskell
around :: String -> String -> (String -> String)
around l r = \s -> l ++ s ++ r

after :: (String -> String) -> (String -> String) -> (String -> String)
after f g = \x -> f (g x)

sayE :: String -> String
sayE = after
  (around "«" "»")
  (\s' -> s' ++ "!")


main :: IO ()
main = putStrLn (sayE "Привет")
```

# Функции высшего порядка (пример)

```haskell
around :: String -> String -> String -> String
around l r s = l ++ s ++ r

after :: (String -> String) -> (String -> String) -> String -> String
after f g x = f (g x)

sayE :: String -> String
sayE = (around "«" "»") `after` (\s' -> s' ++ "!")




main :: IO ()
main = putStrLn (sayE "Привет")
```

# Функции высшего порядка (пример)

```haskell
around :: String -> String -> String -> String
around l r s = l ++ s ++ r



sayE :: String -> String
sayE = around "«" "»" . (++ "!")



main :: IO ()
main = putStrLn (sayE "Привет")
```

# Функции высшего порядка (суммирование)

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

Реализуйте функции через `sumOf`:

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

sumOfSquares :: [Double] -> Double
sumOfSquares =

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

# Функции высшего порядка (суммирование)

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

Следующие определения эквивалентны:

1. `sum xs = sumOf (\x -> x) xs`
2. `sum xs = sumOf id xs`
3. `sum xs = (sumOf id) xs`
4. `sum = \xs -> (sumOf id) xs`
5. `sum = sumOf id`

# Функции высшего порядка (суммирование)

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

Следующие определения эквивалентны:

1. `sum xs = sumOf (\x -> x) xs`
2. `sum xs = sumOf id xs`
3. `sum xs = (sumOf id) xs`
4. `sum = \xs -> (sumOf id) xs`
5. `sum = sumOf id`

# Функции высшего порядка (суммирование)

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

Следующие определения эквивалентны:

1. `sum xs = sumOf (\x -> x) xs`
2. `sum xs = sumOf id xs`
3. `sum xs = (sumOf id) xs`
4. `sum = \xs -> (sumOf id) xs`
5. `sum = sumOf id`

# Функции высшего порядка (суммирование)

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

Следующие определения эквивалентны:

1. `sum xs = sumOf (\x -> x) xs`
2. `sum xs = sumOf id xs`
3. `sum xs = (sumOf id) xs`
4. `sum = \xs -> (sumOf id) xs`
5. `sum = sumOf id`

# Функции высшего порядка (суммирование)

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

Следующие определения эквивалентны:

1. `sum xs = sumOf (\x -> x) xs`
2. `sum xs = sumOf id xs`
3. `sum xs = (sumOf id) xs`
4. `sum = \xs -> (sumOf id) xs`
5. `sum = sumOf id`

# Функции высшего порядка (суммирование)

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

Следующие определения эквивалентны:

1. `sum xs = sumOf (\x -> x) xs`
2. `sum xs = sumOf id xs`
3. `sum xs = (sumOf id) xs`
4. `sum = \xs -> (sumOf id) xs`
5. `sum = sumOf id`

# Функции высшего порядка (суммирование)

```
sumOfSquares :: [Double] -> Double
```

Следующие определения эквивалентны:

1. `sumOfSquares xs = sumOf (\x -> x ^ 2) xs`
2. `sumOfSquares xs = sumOf (^2) xs`
3. `sumOfSquares = sumOf (^2)`

# Функции высшего порядка (суммирование)

```
sumOfSquares :: [Double] -> Double
```

Следующие определения эквивалентны:

1. `sumOfSquares xs = sumOf (\x -> x ^ 2) xs`
2. `sumOfSquares xs = sumOf (^2) xs`
3. `sumOfSquares = sumOf (^2)`

# Функции высшего порядка (суммирование)

```
sumOfSquares :: [Double] -> Double
```

Следующие определения эквивалентны:

1. `sumOfSquares xs = sumOf (\x -> x ^ 2) xs`
2. `sumOfSquares xs = sumOf (^2) xs`
3. `sumOfSquares = sumOf (^2)`

# Функции высшего порядка (суммирование)

```
sumOfSquares :: [Double] -> Double
```

Следующие определения эквивалентны:

1. `sumOfSquares xs = sumOf (\x -> x ^ 2) xs`
2. `sumOfSquares xs = sumOf (^2) xs`
3. `sumOfSquares = sumOf (^2)`

# Реализация сверху вниз

Реализуйте следующие формулы:

$$\mathsf{stddev}(x) := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mathsf{mean}(x))^2}$$

$$\mathsf{mean}(x) := \frac{1}{n}\sum_{i=1}^{n} x_i$$

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

# Реализация сверху вниз

Реализуйте следующие формулы:

$$\mathsf{stddev}(x) := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mathsf{mean}(x))^2}$$

$$\mathsf{mean}(x) := \frac{1}{n}\sum_{i=1}^{n} x_i$$

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

# Реализация сверху вниз

Реализуйте следующие формулы:

$$\mathsf{stddev}(x) := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mathsf{mean}(x))^2}$$

$$\mathsf{mean}(x) := \frac{1}{n}\sum_{i=1}^{n} x_i$$

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

# Реализация сверху вниз

Реализуйте следующие формулы:

$$\mathsf{stddev}(x) := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mathsf{mean}(x))^2}$$

$$\mathsf{mean}(x) := \frac{1}{n}\sum_{i=1}^{n} x_i$$

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

# Реализация сверху вниз

Реализуйте следующие формулы:

$$\mathsf{stddev}(x) := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mathsf{mean}(x))^2}$$

$$\mathsf{mean}(x) := \frac{1}{n}\sum_{i=1}^{n} x_i$$

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

# Реализация сверху вниз

Реализуйте следующие формулы:

$$\mathsf{stddev}(x) := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mathsf{mean}(x))^2}$$

$$\mathsf{mean}(x) := \frac{1}{n}\sum_{i=1}^{n} x_i$$

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

# Реализация сверху вниз

Реализуйте следующие формулы:

$$\mathsf{stddev}(x) := \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mathsf{mean}(x))^2}$$

$$\mathsf{mean}(x) := \frac{1}{n}\sum_{i=1}^{n} x_i$$

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

# Алгебраические типы

# Типы-произведения

*Типы-произведения* — это по записи с безымянными полями (или именованные кортежи):

```haskell
-- | Двумерный вектор.
data Vector = MkVector Double Double
```

1. `Vector` — это новый тип.
2. `MkVector` — это **конструктор значений**, принимающий два аргумента типа `Double`.

```haskell
v :: Vector
v = MkVector 3 4

norm :: Vector -> Double
norm (MkVector x y) = sqrt (x^2 + y^2)
```

# Типы-перечисления

*Типы-перечисления* состоят из набора конструкторов без аргументов.

```haskell
-- | Один из основных цветов.
data Color = Red | Green | Blue
```

1. `Color` — это **тип**.
2. `Red`, `Green`, `Blue` — это **конструкторы значений**.

```haskell
someColor :: Int -> Color
someColor n = if n > 0 then Red else Blue

colorToHex :: Color -> String
colorToHex Red   = "#FF0000"
colorToHex Green = "#00FF00"
colorToHex Blue  = "#0000FF"
```

# Типы-суммы

*Типы-суммы* состоят из набора *произвольных* конструкторов.

```haskell
-- | Результат какого-то вычисления.
data BoolResult
  = Success Bool
  | Failure String
```

1. `BoolResult` — это **тип**.
2. `Success`, `Failure` — это **конструкторы значений**.

```haskell
parseAnswer :: String -> BoolResult
parseAnswer "да"  = Success True
parseAnswer "нет" = Success False
parseAnswer input = Failure ("не понимаю ответ: " ++ input)
```

# Типы-суммы

*Типы-суммы* состоят из набора *произвольных* конструкторов.

```haskell
-- | Фигуры.
data Shape
  = Circle Radius
  | Rectangle Width Height
```

1. `Shape` — это **тип**.
2. `Circle`, `Rectangle` — это **конструкторы значений**.

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

# Типы-суммы

*Типы-суммы* состоят из набора *произвольных* конструкторов.

```haskell
-- | Целочисленное выражение.
data IntExpr
  = Literal Int
  | Plus IntExpr IntExpr  -- e1 + e2
  | Mult IntExpr IntExpr  -- e1 × e2
```

1. `IntExpr` — это **тип**.
2. `Literal`, `Plus`, `Mult` — это **конструкторы значений**.

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

# Типы-суммы

*Типы-суммы* состоят из набора *произвольных* конструкторов.

```haskell
-- | Целочисленное выражение.
data IntExpr
  = Literal Int
  | Plus IntExpr IntExpr  -- e1 + e2
  | Mult IntExpr IntExpr  -- e1 × e2
```

1. `IntExpr` — это **тип**.
2. `Literal`, `Plus`, `Mult` — это **конструкторы значений**.

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

# Перерыв



# Параметрический полиморфизм (на примере)

```
around :: String -> String -> String -> String
around l r s = l ++ s ++ r


after f g x = f (g x)

sayE :: String -> String
sayE s = after say exclaim s
  where
    say s' = around "«" "»" s'
    exclaim s' = s' ++ "!"

main :: IO ()
main = putStrLn (sayE "Привет")
```

Каков тип `after`? Давайте спросим компилятор!

# Параметрический полиморфизм (на примере)

```
around :: String -> String -> String -> String
around l r s = l ++ s ++ r


after f g x = f (g x)

sayE :: String -> String
sayE s = after say exclaim s
  where
    say s' = around "«" "»" s'
    exclaim s' = s' ++ "!"

main :: IO ()
main = putStrLn (sayE "Привет")
```

Каков тип `after`? Давайте спросим компилятор!

# Подсказка компилятора

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



after f g x = f (g x)
```

```
Line 3, Column 1-5: warning: [-Wmissing-signatures]
    Top-level binding with no type signature:
      after :: (t1 -> t2) -> (t3 -> t1) -> t3 -> t2
```

# Параметрический полиморфизм (примеры)

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

after :: (t1 -> t2) -> (t3 -> t1) -> t3 -> t2
after f x = f (f x)
```

Каковы значения типов `t1`, `t2` и `t3` в следующих примерах?

1. `after (+1) (+2) 0`
2. `after (1:) (:[])`
3. `after (+1) length`
4. `after length show`
5. `after after (+) 0 length`

# Параметрический полиморфизм (примеры)

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

after :: (t1 -> t2) -> (t3 -> t1) -> t3 -> t2
after f x = f (f x)
```

Каковы значения типов `t1`, `t2` и `t3` в следующих примерах?

1. `after (+1) (+2) 0`
2. `after (1:) (:[])`
3. `after (+1) length`
4. `after length show`
5. `after after (+) 0 length`

# Параметрический полиморфизм (примеры)

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

after :: (t1 -> t2) -> (t3 -> t1) -> t3 -> t2
after f x = f (f x)
```

Каковы значения типов `t1`, `t2` и `t3` в следующих примерах?

1. `after (+1) (+2) 0`
2. `after (1:) (:[])`
3. `after (+1) length`
4. `after length show`
5. `after after (+) 0 length`

# Параметрический полиморфизм (примеры)

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

after :: (t1 -> t2) -> (t3 -> t1) -> t3 -> t2
after f x = f (f x)
```

Каковы значения типов `t1`, `t2` и `t3` в следующих примерах?

1. `after (+1) (+2) 0`
2. `after (1:) (:[])`
3. `after (+1) length`
4. `after length show`
5. `after after (+) 0 length`

# Параметрический полиморфизм (примеры)

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

after :: (t1 -> t2) -> (t3 -> t1) -> t3 -> t2
after f x = f (f x)
```

Каковы значения типов `t1`, `t2` и `t3` в следующих примерах?

1. `after (+1) (+2) 0`
2. `after (1:) (:[])`
3. `after (+1) length`
4. `after length show`
5. `after after (+) 0 length`

# Параметрический полиморфизм (примеры)

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

after :: (t1 -> t2) -> (t3 -> t1) -> t3 -> t2
after f x = f (f x)
```

Каковы значения типов `t1`, `t2` и `t3` в следующих примерах?

1. `after (+1) (+2) 0`
2. `after (1:) (:[])`
3. `after (+1) length`
4. `after length show`
5. `after after (+) 0 length`

# Функция map

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

Каковы значения типов a и b в следующих примерах?

1. `map (+1) [1, 2, 3]`
2. `map length ["hello", "world"]`
3. `map (> 0) [4, -2, 1, 0]`
4. `map (\f -> f 3) [(+1), (^2)]`
5. `map (\x -> [x]) []`
6. `map map [(+1)]`

# Функция map

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

Каковы значения типов a и b в следующих примерах?

1. `map (+1) [1, 2, 3]`
2. `map length ["hello", "world"]`
3. `map (> 0) [4, -2, 1, 0]`
4. `map (\f -> f 3) [(+1), (^2)]`
5. `map (\x -> [x]) []`
6. `map map [(+1)]`

# Функция map

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

Каковы значения типов a и b в следующих примерах?

1. `map (+1) [1, 2, 3]`
2. `map length ["hello", "world"]`
3. `map (> 0) [4, -2, 1, 0]`
4. `map (\f -> f 3) [(+1), (^2)]`
5. `map (\x -> [x]) []`
6. `map map [(+1)]`

# Функция map

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

Каковы значения типов a и b в следующих примерах?

1. `map (+1) [1, 2, 3]`
2. `map length ["hello", "world"]`
3. `map (> 0) [4, -2, 1, 0]`
4. `map (\f -> f 3) [(+1), (^2)]`
5. `map (\x -> [x]) []`
6. `map map [(+1)]`

# Функция map

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

Каковы значения типов a и b в следующих примерах?

1. `map (+1) [1, 2, 3]`
2. `map length ["hello", "world"]`
3. `map (> 0) [4, -2, 1, 0]`
4. `map (\f -> f 3) [(+1), (^2)]`
5. `map (\x -> [x]) []`
6. `map map [(+1)]`

# Функция map

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

Каковы значения типов a и b в следующих примерах?

1. `map (+1) [1, 2, 3]`
2. `map length ["hello", "world"]`
3. `map (> 0) [4, -2, 1, 0]`
4. `map (\f -> f 3) [(+1), (^2)]`
5. `map (\x -> [x]) []`
6. `map map [(+1)]`

# Функция map

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

Каковы значения типов a и b в следующих примерах?

1. `map (+1) [1, 2, 3]`
2. `map length ["hello", "world"]`
3. `map (> 0) [4, -2, 1, 0]`
4. `map (\f -> f 3) [(+1), (^2)]`
5. `map (\x -> [x]) []`
6. `map map [(+1)]`

# Параметрические типы

```haskell
data BoolResult = Success Bool | Failure String
```

1. `BoolResult` — это **тип**.
2. `Success`, `Failure` — это **конструкторы значений**.

```haskell
parseAnswer :: String -> BoolResult
parseAnswer "да"  = Success True
parseAnswer "нет" = Success False
parseAnswer input = Failure ("не понимаю ответ: " ++ input)
```

# Параметрические типы

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

1. `Result` — это **конструктор типов** (но НЕ тип!).
2. `Result a` — это **тип** (для любого типа `a`).
3. `Result Bool` — это **тип**.
4. `Success`, `Failure` — это **конструкторы значений**.

```haskell
parseAnswer :: String -> Result Bool
parseAnswer "да"  = Success True
parseAnswer "нет" = Success False
parseAnswer input = Failure ("не понимаю ответ: " ++ input)
```

# Параметрические типы (Maybe)

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

1. `Maybe` — это **конструктор типов** (но НЕ тип!).
2. `Maybe a` — это **тип** (для любого типа `a`).
3. `Maybe Bool` — это **тип**.
4. `Nothing`, `Just` — это **конструкторы типов**.

```haskell
parseAnswer :: String -> Maybe Bool
parseAnswer "да"  = Just True
parseAnswer "нет" = Just False
parseAnswer input = Nothing
```

# Параметрические типы (Maybe, упражнение)

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

type UserId = String
data Status  = Sending | Sent | Read
data Message = Message UserId String Status
```

Реализуйте функцию, статус последнего сообщения от заданного пользователя:

```haskell
lastMessageStatus :: UserId -> [Message] -> Maybe Status
lastMessageStatus userId messages =


```

**Спасибо за внимание!**