

Функциональное программирование

Лекция 5. Операторы, форсирование, списки

Денис Николаевич Москвин

НИУ ВШЭ — СПб, ШИФТ,
бакалавриат ПМИ, 2 курс

30.09.2025

- 1 Операторы и их сечения
- 2 Ленивость и строгость, форсирование
- 3 Стандартные списки и работа с ними

- 1 Операторы и их сечения
- 2 Ленивость и строгость, форсирование
- 3 Стандартные списки и работа с ними

Операторы

- *Оператор* — это комбинация из одного или более символов

! # \$ % & * + . / < > ? @ ^ | - ~ = \ :

- Все операторы *бинарные* и *инфиксные*.
- Исключение: унарный префиксный минус, который всегда ссылается на `Prelude.negate`.
- Операторы, начинающиеся на двоеточие, должны быть конструкторами данных.
- Пример: оператор для суммы квадратов

```
a *** b = a ^ 2 + b ^ 2
```

```
GHCi> 3 *** 4  
25
```

Инфиксная и префиксная нотация

- Операторы могут определяться и использоваться в префиксном (функциональном) стиле.
- Функции, в свою очередь, могут определяться и использоваться в инфиксном (операторном) стиле.

```
(****) a b = a ^ 3 + b ^ 3  
x `plusminus` y = (x + y, x - y)
```

```
GHCi> (****) 2 3  
35  
GHCi> 2 **** 3  
35  
GHCi> plusminus 4 3  
(7,1)  
GHCi> 4 `plusminus` 3  
(7,1)
```

Проблема приоритета и ассоциативности

Чему равны значения выражений?

1 ******* 2 **+** 3

1 ******* 2 ******* 3

Проблема приоритета и ассоциативности

Чему равны значения выражений?

1 ******* 2 + 3

1 ******* 2 ******* 3

Инфиксные операторы требуют определения

- *приоритета*: какой оператор из цепочки выполнять первым;
- *ассоциативности*: какой оператор из цепочки выполнять первым при равном приоритете.

Приоритет и ассоциативность (fixity)

С помощью объявлений `infixl`, `infixr` или `infix` задаётся приоритет и ассоциативность операторов и функций.

```
infixl 6  ***, *****
```

Теперь введённые нами операторы левоассоциативны и имеют тот же приоритет, что и обычный оператор сложения.

Задача: расставьте скобки и вычислите

```
1  *** 2 + 3  
3  + 1 *** 2 * 3
```

Функциям тоже можно задавать приоритет

```
infix 5 `plusminus`
```

```
GHCI> 5 + 3 `plusminus` 6 * 2  
(20,-4)
```


Приоритет стандартных операторов

```
infixl 9  !!
infixr 9   .
infixr 8   ^, ^^, **
infixl 7   *, /, `quot`, `rem`, `div`, `mod`
infixl 6   +, -
infixr 5   ++, :
infix  4   ==, /=, <, <=, >=, >, `elem`, `notElem`
infixr 3   &&
infixr 2   ||
infixl 1   >>, >>=
infixr 1   =<<
infixr 0   $, $!, `seq`
```

- В GHCi можно подглядеть, набрав `:info (&&)`.
- Применение имеет наивысший (10) приоритет.

Стандартный оператор (\$)

- Оператор \$ задаёт применение, но с наименьшим возможным приоритетом

```
infixr 0 $  
($) :: (a -> b) -> a -> b  
f $ x = f x
```

- Используется для элиминации избыточных скобок:

$$f (g x) \equiv f \$ g x$$
$$f (g x (h y)) \equiv f \$ g x (h y) \equiv f \$ g x \$ h y$$

- Из примера ясна причина правоассоциативности.
- \$ используют также для передачи аппликации в ФВП.

- Операторы на самом деле просто функции и, поэтому, допускают частичное применение.
- *Сечения* (sections) — синтаксический сахар для частичного применения как к левому, так и к правому аргументу.
- Левое сечение:

```
(2 ***) ≡ (***) 2 ≡ \y -> 2 *** y
```

- Правое сечение:

```
(*** 3) ≡ \x -> x *** 3
```

```
GHCi> :t (!!)  
(!!) :: [a] -> Int -> a  
GHCi> :t (!! 0)  
(!! 0) :: [a] -> a
```

- Наличие скобок при задании сечений обязательно, это часть их синтаксиса.

Стандартный оператор (.)

Оператор (.) задаёт композицию функций

```
infixr 9  .  
(.) :: (b -> c) -> (a -> b) -> a -> c  
f . g = \ x -> f (g x)
```

Например, выражение $(^2) \cdot (+5)$ — это функция, прибавляющая 5 к своему аргументу, а затем возводящая результат в квадрат:

```
GHCI> (^2) . (+5) $ 1  
36  
GHCI> (^2) . (+5) $ 2  
49
```

```
infixl 1 &                                -- Data.Function
(&) :: a -> (a -> b) -> b
(&) = flip ($)
```

Этот оператор разворачивает конвейер вычислений

```
GHCI> (+12) $ (*10) $ 3
42
GHCI> 3 & (*10) & (+12)
42
```

- 1 Операторы и их сечения
- 2 Ленивость и строгость, форсирование
- 3 Стандартные списки и работа с ними

Значение \perp

Вспомним выражение `bot :: Bool`, определённое рекурсивно

```
bot :: Bool
bot = not bot
```

Его значение — не `True` и не `False`, а \perp (основание, дно). В Haskell'e \perp — значение, разделяемое всеми типами:

```
 $\perp$  :: forall {a}. a
```

Ошибкам тоже приписывается это значение.

```
GHCi> :set -fprint-explicit-foralls
GHCi> :t undefined
undefined :: forall {a}. a
GHCi> :t error
error :: forall {a}. [Char] -> a
```

Нестрогая (ленивая) семантика

Haskell по умолчанию гарантирует вызов-по-необходимости

```
ignore x = 42
```

```
GHCI> ignore undefined
```

```
42
```

```
GHCI> ignore bot
```

```
42
```

Такие функции как `ignore`, игнорирующие значение своего аргумента, называются *нестрогими* по этому аргументу.

Для *строгих* функций, наоборот, всегда выполняется

```
f ⊥ = ⊥
```


Как форсировать вычисления

- Для форсированного вычисления используют функцию

```
seq :: a -> b -> b
seq ⊥ b = ⊥
seq a b = b, если a ≠ ⊥
```

- Синтаксически `seq` похожа на `\a b -> b`. Но она нарушает ленивую семантику языка, позволяя форсировать вычисление без необходимости.
- `seq` потворствует распространению \perp , интересуясь значением своего первого аргумента

```
GHCi> seq undefined 42
*** Exception: Prelude.undefined
GHCi> seq (id undefined) 42
*** Exception: Prelude.undefined
```

Как сильно seq форсирует?

- seq производит вычисление своего первого аргумента, если в нем имеется редекс на верхнем уровне.
- Однако конструкторы данных, лямбда-абстракции и частично примененные функции, являясь «значениями», обеспечивают барьер для распространения \perp

```
GHCi> seq (undefined,undefined) 42
42
GHCi> seq (\x -> undefined) 42
42
GHCi> seq ((+) undefined) 42
42
```

- Подобные «не редексы» объединяют одним термином – их называют *слабой головной нормальной формой* (weak head normal form, WHNF).

Аппликация с вызовом по значению

- Через `seq` определяется энергичная аппликация (с вызовом-по-значению)

```
infixr 0 $!  
($!) :: (a -> b) -> a -> b  
f $! x = x `seq` f x
```

- Форсирование приводит к «худшей определенности»

```
GHCi> ignore undefined  
42  
GHCi> ignore $ undefined  
42  
GHCi> ignore $! undefined  
*** Exception: Prelude.undefined
```

Пример использования seq

- Вспомним факториал с аккумулярующим параметром

```
factorial n = helper 1 n where
  helper acc k | k > 1 = helper (acc * k) (k - 1)
               | otherwise = acc
```

- Из-за ленивости acc будет содержать thunk вида
 $(\dots((1 * n) * (n - 1)) * (n - 2) * \dots * 2)$
- Оптимизатор GHC обычно справляется, имея встроенный *анализатор строгости*. Но можно, не полагаясь на него, написать

```
factorial n = helper 1 n where
  helper acc k | k > 1 = (helper $! acc * k) (k - 1)
               | otherwise = acc
```

- 1 Операторы и их сечения
- 2 Ленивость и строгость, форсирование
- 3 Стандартные списки и работа с ними

Стандартные списки

Имеют два конструктора

```
[] :: [a]
(:) :: a -> [a] -> [a]
infixr 5 :
```

Для удобства введён синтаксический сахар

```
[1,2,3] ≡ 1:(2:(3:[])) ≡ 1:2:3:[]
```

Пример определения функции

```
head      :: [a] -> a
head (x:_) = x
head []    = error "Prelude.head: empty list"
```

Это частичная функция, в современном Haskell использовать их не рекомендуется.

Основные функции из Data.List

```
tail      :: [a] -> [a]
tail (_:xs) = xs
tail []    = error "Prelude.tail: empty list"
```

```
GHCI> tail [1,2,3,4]
[2,3,4]
GHCI> tail "ABCD"
"BCD"
```

```
(++)      :: [a] -> [a] -> [a]
[] ++ ys  = ys
(x:xs) ++ ys = x : xs ++ ys
```

Какова сложность tail? конкатенации?

Основные функции из Data.List (2)

```
take                :: Int -> [a] -> [a]
take n _           | n <= 0 = []
take _ []          = []
take n (x:xs)      = x : take (n-1) xs
```

```
GHCI> take 3 "ABCDEFGH"
"ABC"
GHCI> take 10 "ABCDEFGH"
"ABCDEFGH"
```

```
drop                :: Int -> [a] -> [a]
drop                = undefined
```

Как через drop сделать тотальный эквивалент tail?

Функции высших порядков (HOF)

```
filter      :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
  | p x      = x : filter p xs
  | otherwise = filter p xs
```

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

```
GHCI> map length ["Good","bye","world"]
[4,3,5]
GHCI> map (^2) . map length $ ["Good","bye","world"]
[16,9,25]
```

Семейства zip и zipWith

```
zip :: [a] -> [b] -> [(a,b)]
zip []      _      = []
zip _      []      = []
zip (a:as) (b:bs) = (a,b) : zip as bs
```

```
zip3 :: [a] -> [b] -> [c] -> [(a,b,c)]
```

```
unzip :: [(a,b)] -> ([a],[b])
```

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ []      _      = []
zipWith _ _      []      = []
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
```

```
zipWith3 :: (a -> b -> c -> d)
          -> [a] -> [b] -> [c] -> [d]
```

«Бесконечные» структуры данных

«Бесконечные» структуры данных описываются рекурсией:

```
GHCi> ones = 1 : ones  
GHCi> :type ones  
ones :: Num a => [a]
```

Благодаря ленивости вычисляется только то, что требуется:

```
GHCi> numsFrom n = n : numsFrom (n+1)  
GHCi> squares = map (^2) (numsFrom 0)  
GHCi> take 10 squares  
[0,1,4,9,16,25,36,49,64,81]  
GHCi> fibs = 0 : 1 : zipWith (+) fibs (drop 1 fibs)  
GHCi> take 10 fibs  
[0,1,1,2,3,5,8,13,21,34]
```

Имеется компактный способ описывать большие «регулярные» списки:

```
GHCI> [1..10]
[1,2,3,4,5,6,7,8,9,10]
GHCI> [1,3..17]
[1,3,5,7,9,11,13,15,17]
GHCI> ['A'..'z']
"ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz
vwxyz"
GHCI> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,
...
```

Для формирования «нелинейных» последовательностей есть другая техника...

Выделение списков (List Comprehension)

Название происходит из аксиоматической теории множеств.

```
GHCi> digits = [0..9]
GHCi> [ x^2 | x <- digits ]
[0,1,4,9,16,25,36,49,64,81]
```

При нескольких генераторах чаще обновляется тот, что правее:

```
GHCi> [ [x,y] | x <- "ABC", y <- "de" ]
["Ad","Ae","Bd","Be","Cd","Ce"]
```

Выделение списков: дополнительные возможности

Дополнительные возможности, доступные при выделении:

- генераторы могут ссылаться на значения из предыдущих

```
GHCi> [ (x,y) | x <- [1..3], y <- [1..x] ]  
[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]
```

- можно использовать предикаты над этими значениями

```
GHCi> l = [1..19]  
GHCi> [ (a,b,c) | a<-l, b<-[1..a], c<-l, a^2 + b^2 == c^2 ]  
[(4,3,5),(8,6,10),(12,5,13),(12,9,15),(15,8,17)]
```

- можно использовать сопоставление с образцом

```
GHCi> tst = [(1,7),(2,4),(3,11)]  
GHCi> [ x + y | (x,y) <- tst ]  
[8,6,14]
```