

# Лекция 6: работа со списками

Функциональное программирование на Haskell

---

Алексей Романов

15 марта 2024 г.

МИЭТ

# Списки в Haskell

- Поговорим о списках подробнее.
- Это структура данных, которая постоянно встречается в программах.
- Но по поведению и характеристикам *очень сильно* отличается от списков в Java и C# (или `std::vector` в C++).

# Списки в Haskell

- Поговорим о списках подробнее.
- Это структура данных, которая постоянно встречается в программах.
- Но по поведению и характеристикам *очень сильно* отличается от списков в Java и C# (или `std::vector` в C++).
- Списки в Haskell и множестве других языков, начиная с Lisp — односвязные.

# Списки в Haskell

- Поговорим о списках подробнее.
- Это структура данных, которая постоянно встречается в программах.
- Но по поведению и характеристикам *очень сильно* отличается от списков в Java и C# (или `std::vector` в C++).
- Списки в Haskell и множестве других языков, начиная с Lisp — односвязные.
- То есть вместе с каждым элементом хранится ссылка на остальную часть списка.
- Вспомним определение:  

```
data [a] = [] | a : [a]
```

## Некоторые функции первого порядка над списками

- `(++) :: [a] -> [a] -> [a]`

## Некоторые функции первого порядка над списками

- `(++) :: [a] -> [a] -> [a]` — объединение.
- `(!!) :: [a] -> Int -> a`

## Некоторые функции первого порядка над списками

- `(++) :: [a] -> [a] -> [a]` — объединение.
- `(!!) :: [a] -> Int -> a` — элемент по индексу.
- `null :: a -> Bool`

## Некоторые функции первого порядка над списками

- `(++) :: [a] -> [a] -> [a]` — объединение.
- `(!!) :: [a] -> Int -> a` — элемент по индексу.
- `null :: a -> Bool` — проверка на пустоту (лучше, чем `length xs == 0`! Но сопоставление ещё лучше).
- `elem :: Eq a => a -> [a] -> Bool`



## Некоторые функции первого порядка над списками

- `(++) :: [a] -> [a] -> [a]` — объединение.
- `(!!) :: [a] -> Int -> a` — элемент по индексу.
- `null :: a -> Bool` — проверка на пустоту (лучше, чем `length xs == 0`! Но сопоставление ещё лучше).
- `elem :: Eq a => a -> [a] -> Bool` — проверка на вхождение элемента.
- `and, or :: [Bool] -> Bool`

## Некоторые функции первого порядка над списками

- `(++) :: [a] -> [a] -> [a]` — объединение.
- `(!!) :: [a] -> Int -> a` — элемент по индексу.
- `null :: a -> Bool` — проверка на пустоту (лучше, чем `length xs == 0`! Но сопоставление ещё лучше).
- `elem :: Eq a => a -> [a] -> Bool` — проверка на вхождение элемента.
- `and, or :: [Bool] -> Bool` — все ли элементы `True` (или есть ли такой).

## Некоторые функции первого порядка над списками

- `(++) :: [a] -> [a] -> [a]` — объединение.
- `(!!) :: [a] -> Int -> a` — элемент по индексу.
- `null :: a -> Bool` — проверка на пустоту (лучше, чем `length xs == 0`! Но сопоставление ещё лучше).
- `elem :: Eq a => a -> [a] -> Bool` — проверка на вхождение элемента.
- `and, or :: [Bool] -> Bool` — все ли элементы `True` (или есть ли такой).
- У части из них тип на самом деле более общий, но пока нам не нужен.

## Некоторые функции первого порядка над списками

- `(++) :: [a] -> [a] -> [a]` — объединение.
- `(!!) :: [a] -> Int -> a` — элемент по индексу.
- `null :: a -> Bool` — проверка на пустоту (лучше, чем `length xs == 0`! Но сопоставление ещё лучше).
- `elem :: Eq a => a -> [a] -> Bool` — проверка на вхождение элемента.
- `and, or :: [Bool] -> Bool` — все ли элементы `True` (или есть ли такой).
- У части из них тип на самом деле более общий, но пока нам не нужен.
- Все основные в `Data.List`.

## Некоторые функции первого порядка над списками

- `(++) :: [a] -> [a] -> [a]` — объединение.
- `(!!) :: [a] -> Int -> a` — элемент по индексу.
- `null :: a -> Bool` — проверка на пустоту (лучше, чем `length xs == 0`! Но сопоставление ещё лучше).
- `elem :: Eq a => a -> [a] -> Bool` — проверка на вхождение элемента.
- `and, or :: [Bool] -> Bool` — все ли элементы `True` (или есть ли такой).
- У части из них тип на самом деле более общий, но пока нам не нужен.
- Все основные в `Data.List`. И ещё пакет `safe`.

## Некоторые функции второго порядка над списками

- `map :: (a -> b) -> [a] -> [b]`

## Некоторые функции второго порядка над списками

- `map :: (a -> b) -> [a] -> [b]` — применяет функцию ко всем элементам.
- `filter :: (a -> Bool) -> [a] -> [a]`

## Некоторые функции второго порядка над списками

- `map :: (a -> b) -> [a] -> [b]` — применяет функцию ко всем элементам.
- `filter :: (a -> Bool) -> [a] -> [a]` — выбирает элементы, удовлетворяющие условию.
- `concatMap :: (a -> [b]) -> [a] -> [b]`



## Некоторые функции второго порядка над списками

- `map :: (a -> b) -> [a] -> [b]` — применяет функцию ко всем элементам.
- `filter :: (a -> Bool) -> [a] -> [a]` — выбирает элементы, удовлетворяющие условию.
- `concatMap :: (a -> [b]) -> [a] -> [b]` — применяет функцию ко всем элементам и объединяет результаты.
- `all, any :: (a -> Bool) -> [a] -> Bool`

## Некоторые функции второго порядка над списками

- `map :: (a -> b) -> [a] -> [b]` — применяет функцию ко всем элементам.
- `filter :: (a -> Bool) -> [a] -> [a]` — выбирает элементы, удовлетворяющие условию.
- `concatMap :: (a -> [b]) -> [a] -> [b]` — применяет функцию ко всем элементам и объединяет результаты.
- `all, any :: (a -> Bool) -> [a] -> Bool` — все ли элементы удовлетворяют условию (или есть ли такой).

## Некоторые функции второго порядка над списками

- `map :: (a -> b) -> [a] -> [b]` — применяет функцию ко всем элементам.
- `filter :: (a -> Bool) -> [a] -> [a]` — выбирает элементы, удовлетворяющие условию.
- `concatMap :: (a -> [b]) -> [a] -> [b]` — применяет функцию ко всем элементам и объединяет результаты.
- `all, any :: (a -> Bool) -> [a] -> Bool` — все ли элементы удовлетворяют условию (или есть ли такой).
- Упражнение: реализуем какие-нибудь из них (рекурсивно и друг через друга).

## Сложность функций над списками

- Базовые операции — конструирование списков и их разбор через `:`.

## Сложность функций над списками

- Базовые операции — конструирование списков и их разбор через `:.`
- Поэтому `length` (а значит, и `length xs == 0`) проходит весь список целиком, а `null` нет.

## Сложность функций над списками

- Базовые операции — конструирование списков и их разбор через `:.`
- Поэтому `length` (а значит, и `length xs == 0`) проходит весь список целиком, а `null` нет.
- `xs !! n` — цена пропорциональна

## Сложность функций над списками

- Базовые операции — конструирование списков и их разбор через `:.`
- Поэтому `length` (а значит, и `length xs == 0`) проходит весь список целиком, а `null` нет.
- `xs !! n` — цена пропорциональна `n`.

## Сложность функций над списками

- Базовые операции — конструирование списков и их разбор через `:.`
- Поэтому `length` (а значит, и `length xs == 0`) проходит весь список целиком, а `null` нет.
- `xs !! n` — цена пропорциональна `n`.
- Чем определяется цена `xs ++ ys`?



## Сложность функций над списками

- Базовые операции — конструирование списков и их разбор через `:.`
- Поэтому `length` (а значит, и `length xs == 0`) проходит весь список целиком, а `null` нет.
- `xs !! n` — цена пропорциональна `n`.
- Чем определяется цена `xs ++ ys`? Длиной `xs`.

## Сложность функций над списками

- Базовые операции — конструирование списков и их разбор через `:.`
- Поэтому `length` (а значит, и `length xs == 0`) проходит весь список целиком, а `null` нет.
- `xs !! n` — цена пропорциональна `n`.
- Чем определяется цена `xs ++ ys`? Длиной `xs`.
- Добавление в конец списка `xs ++ [y]` получается дорогим.
- В цикле может сделать линейный алгоритм квадратичным.

## Сложность функций над списками

- Базовые операции — конструирование списков и их разбор через `:`.
- Поэтому `length` (а значит, и `length xs == 0`) проходит весь список целиком, а `null` нет.
- `xs !! n` — цена пропорциональна `n`.
- Чем определяется цена `xs ++ ys`? Длиной `xs`.
- Добавление в конец списка `xs ++ [y]` получается дорогим.
- В цикле может сделать линейный алгоритм квадратичным.
- Вместо добавления в конец часто лучше добавлять в начало, а потом развернуть.

## Сложность функций над списками

- Базовые операции — конструирование списков и их разбор через `:`.
- Поэтому `length` (а значит, и `length xs == 0`) проходит весь список целиком, а `null` нет.
- `xs !! n` — цена пропорциональна `n`.
- Чем определяется цена `xs ++ ys`? Длиной `xs`.
- Добавление в конец списка `xs ++ [y]` получается дорогим.
- В цикле может сделать линейный алгоритм квадратичным.
- Вместо добавления в конец часто лучше добавлять в начало, а потом развернуть.
- Всё это осложняется ленивостью, о ней позже.

## Выделения списков (list comprehensions)

- Есть удобный способ записи списков, похожий на выражения для множеств.

```
[x*x | x <- [1..5], even x] ==
```

## Выделения списков (list comprehensions)

- Есть удобный способ записи списков, похожий на выражения для множеств.

`[x*x | x <- [1..5], even x] == [4, 16].`

- Как  $\{x^2 \mid x \in \{1, \dots, 5\}, \text{even}(x)\}$ .
- `x <- [1..5]` — генератор, `even x` — фильтр.

## Выделения списков (list comprehensions)

- Есть удобный способ записи списков, похожий на выражения для множеств.

`[x*x | x <- [1..5], even x] == [4, 16].`

- Как  $\{x^2 \mid x \in \{1, \dots, 5\}, \text{even}(x)\}$ .
- `x <- [1..5]` — генератор, `even x` — фильтр.
- Тех и других может быть много.
- Генераторы могут использовать переменные, введённые в предыдущих. Пример:

`[(x, y) | x <- [1..5], y <- [1..x], odd (x - y)] ==`

## Выделения списков (list comprehensions)

- Есть удобный способ записи списков, похожий на выражения для множеств.

`[x*x | x <- [1..5], even x] == [4, 16].`

- Как  $\{x^2 \mid x \in \{1, \dots, 5\}, \text{even}(x)\}$ .
- `x <- [1..5]` — генератор, `even x` — фильтр.
- Тех и других может быть много.

- Генераторы могут использовать переменные, введённые в предыдущих. Пример:

`[(x, y) | x <- [1..5], y <- [1..x], odd (x - y)] == [(3, 2), (5, 2), (5, 4)]`

- Эти выражения преобразуются в комбинации `concatMap`, `map` и `filter`:

`[x*x | x <- [1..5], even x] ==`



## Выделения списков (list comprehensions)

- Есть удобный способ записи списков, похожий на выражения для множеств.

```
[x*x | x <- [1..5], even x] == [4, 16].
```

- Как  $\{x^2 \mid x \in \{1, \dots, 5\}, \text{even}(x)\}$ .
- `x <- [1..5]` — генератор, `even x` — фильтр.
- Тех и других может быть много.

- Генераторы могут использовать переменные, введённые в предыдущих. Пример:

```
[(x, y) | x <- [1..5], y <- [1..x], odd (x - y)] ==  
  [(3, 2), (5, 2), (5, 4)]
```

- Эти выражения преобразуются в комбинации `concatMap`, `map` и `filter`:

```
[x*x | x <- [1..5], even x] ==  
  map (\x -> x*x) (filter even [1..5])
```

## Правая свёртка списков и структурная рекурсия

- При *структурной рекурсии* рекурсивный вызов делается на подтерме одного из аргументов функции (например, хвосте списка).
- Большинство наших рекурсивных определений выглядели именно так.

# Правая свёртка списков и структурная рекурсия

- При *структурной рекурсии* рекурсивный вызов делается на подтерме одного из аргументов функции (например, хвосте списка).
- Большинство наших рекурсивных определений выглядели именно так.
- Оказывается, что есть универсальная функция, через которую выражаются такие определения:  

```
foldr :: (a -> b -> b) -> b -> [a] -> b  
foldr f z [x1, x2, ..., xn] ==  
  x1 `f` (x2 `f` ... (xn `f` z) ...)
```
- Правая свёртка проходит по структуре списка и заменяет `[]` на `z` и `(:)` на `f`.
- Например, `sum` получится, если взять `f =`

# Правая свёртка списков и структурная рекурсия

- При *структурной рекурсии* рекурсивный вызов делается на подтерме одного из аргументов функции (например, хвосте списка).
- Большинство наших рекурсивных определений выглядели именно так.
- Оказывается, что есть универсальная функция, через которую выражаются такие определения:  

```
foldr :: (a -> b -> b) -> b -> [a] -> b  
foldr f z [x1, x2, ..., xn] ==  
  x1 `f` (x2 `f` ... (xn `f` z) ...)
```
- Правая свёртка проходит по структуре списка и заменяет `[]` на `z` и `(:)` на `f`.
- Например, `sum` получится, если взять `f = (+)` и `z =`

# Правая свёртка списков и структурная рекурсия

- При *структурной рекурсии* рекурсивный вызов делается на подтерме одного из аргументов функции (например, хвосте списка).
- Большинство наших рекурсивных определений выглядели именно так.
- Оказывается, что есть универсальная функция, через которую выражаются такие определения:  

```
foldr :: (a -> b -> b) -> b -> [a] -> b  
foldr f z [x1, x2, ..., xn] ==  
  x1 `f` (x2 `f` ... (xn `f` z) ...)
```
- Правая свёртка проходит по структуре списка и заменяет `[]` на `z` и `(:)` на `f`.
- Например, `sum` получится, если взять `f = (+)` и `z = 0`.
- Упражнение: реализовать `length`.

## Левая свёртка и хвостовая рекурсия

- При *хвостовой рекурсии* рекурсивный вызов делается в самом конце вычисления. Например,
- `sum xs = sum' 0 xs where`  
    `sum' acc [] = acc`  
    `sum' acc (x:xs) = sum'`

## Левая свёртка и хвостовая рекурсия

- При *хвостовой рекурсии* рекурсивный вызов делается в самом конце вычисления. Например,
- ```
sum xs = sum' 0 xs where  
  sum' acc [] = acc  
  sum' acc (x:xs) = sum' (acc + x) xs
```
- Универсальная функция для таких определений:  

```
foldl :: (b -> a -> b) -> b -> [a] -> b  
foldl _f z [] = z  
foldl f z (x:xs) = foldl f
```

# Левая свёртка и хвостовая рекурсия

- При *хвостовой рекурсии* рекурсивный вызов делается в самом конце вычисления. Например,
- ```
sum xs = sum' 0 xs where  
  sum' acc [] = acc  
  sum' acc (x:xs) = sum' (acc + x) xs
```
- Универсальная функция для таких определений:  

```
foldl :: (b -> a -> b) -> b -> [a] -> b  
foldl _f z [] = z  
foldl f z (x:xs) = foldl f (f z x) xs
```
- Она проходит по списку слева и получается  
 $(\dots((z \text{ `f` } x_1) \text{ `f` } x_2) \text{ `f` } \dots) \text{ `f` } x_n$
- Хвостовая рекурсия может быть скомпилирована в цикл и в других языках ФП очень важна. В Haskell ленивость всё усложняет.



## Свёртки других типов данных

- Сравним объявление списка с типом `foldr`:  
`data [a] = a : [a] | []`  
`foldr :: (a -> b -> b) -> b -> [a] -> b`
- Видим, что у нас по одному аргументу для каждого конструктора: каждый принимает аргументы типов полей этого конструктора (а если там `[a]`, то `b`) и возвращает `b`.
- Последний аргумент — разбираемый список.
- Для `Maybe`:

```
data Maybe a = Nothing | Just a
foldMaybe :: ? -> ? -> ? -> b
```

## Свёртки других типов данных

- Сравним объявление списка с типом `foldr`:  
`data [a] = a : [a] | []`  
`foldr :: (a -> b -> b) -> b -> [a] -> b`
- Видим, что у нас по одному аргументу для каждого конструктора: каждый принимает аргументы типов полей этого конструктора (а если там `[a]`, то `b`) и возвращает `b`.
- Последний аргумент — разбираемый список.
- Для `Maybe`:

```
data Maybe a = Nothing | Just a
foldMaybe :: ? -> ? -> ? -> b
foldMaybe :: b -> (a -> b) -> Maybe a -> b
```

## Свёртки других типов данных

- Сравним объявление списка с типом `foldr`:  

```
data [a] = a : [a]      | []  
foldr :: (a -> b -> b) -> b -> [a] -> b
```
- Видим, что у нас по одному аргументу для каждого конструктора: каждый принимает аргументы типов полей этого конструктора (а если там `[a]`, то `b`) и возвращает `b`.
- Последний аргумент — разбираемый список.
- Для `Maybe`:  

```
data Maybe a = Nothing | Just a  
foldMaybe :: ?          -> ?          -> ?          -> b  
foldMaybe :: b          -> (a -> b) -> Maybe a -> b
```
- Это `Data.Maybe.maybe` с обратным порядком аргументов.

# Свёртка деревьев

- Для деревьев возьмём пример, где данные хранятся в листьях, но не в ветвях:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
foldTree    :: ? ->          ?                -> Tree a -> b
```

# Свёртка деревьев

- Для деревьев возьмём пример, где данные хранятся в листьях, но не в ветвях:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
foldTree    :: ? ->          ?                -> Tree a -> b
foldTree    :: (a -> b) -> (b -> b -> b) -> Tree a -> b
```

# Свёртка деревьев

- Для деревьев возьмём пример, где данные хранятся в листьях, но не в ветвях:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
foldTree    :: ? ->          ?                -> Tree a -> b
foldTree    :: (a -> b) -> (b -> b -> b) -> Tree a -> b

sumTree = foldTree
```

# Свёртка деревьев

- Для деревьев возьмём пример, где данные хранятся в листьях, но не в ветвях:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
foldTree   :: ? ->          ?                -> Tree a -> b
foldTree   :: (a -> b) -> (b -> b -> b) -> Tree a -> b

sumTree = foldTree id (+)
```

- Применение обобщённой свёртки для обработки синтаксических деревьев
- A tutorial on the universality and expressiveness of fold
- Getting a Fix from the Right Fold (реализация `dropWhile` через `foldr`)