Pattern matching & list combinators

Михаил Беляев

October 19, 2016

Pattern matching: основы

```haskell
-- Базовые паттерны
-- Переменная
foo x = x + 2
-- Игнорирование
bar _ = 5
-- Комбинации
fee x y _ z = x + y * z
```

Pattern matching: простые примеры

```haskell
-- Списки
len [] = 0
len (h:t) = 1 + len t
-- Кортежи
pair a b = (a,b)
first (a,b) = a
second (a,b) = b
```

Pattern matching: пользовательские типы

```
data Maybe a = Nothing | Just a
isEmpty (Just _) = False
isEmpty (Nothing) = True

plus (Just x) (Just y) = Just(x + y)
plus Nothing _ = Nothing
plus _ Nothing = Nothing
```

Pattern matching: внутри `let` (или `where`)

```haskell
-- вернуть число изнутри Maybe или 42
elementOr42 x =
    if isEmpty x
    then 42
    else
        let (Just y) = x in
        y
```

Pattern matching: `case ... of`

Можно проверять паттерны прямо в середине функции

```
x = case <expression> of <pattern> -> <value>
                         <pattern> -> <value>
                       ...
```

Pattern matching: `case ... of`

```haskell
data Either a b = Left a | Right b
bothToString ei =
    case ei of
        (Left x)  -> show x
        (Right x) -> show x
```

Pattern matching: pattern guards

При объявлении функции можно вставлять проверки условий, при которых паттерн «подходит»

```
function <pattern> | <condition>       = <body>
                   | <other condition> = <body>
                   | otherwise         = <body>
```

Pattern matching: pattern guards

```
data BinaryTree = EmptyTree
                | Leaf Integer
                | Node Integer BinaryTree BinaryTree
containsElement EmptyTree _ = False
containsElement (Leaf x)  y | (x == y) = True
                            | otherwise = False
containsElement (Node c l r) y | (c == y) = True
                               | (c < y) =
                                   containsElement l y
                               | (c > y) =
                                   containsElement r y
```

(На самом деле `otherwise` это просто `True`)

Pattern matching: вложенные паттерны

Паттерны можно объединять друг с другом в очень мощные конструкции

```
-- Из списка Maybe найти первый элемент,
-- содержащий значение
firstJust (Just x : _) = x
firstJust (Nothing: t) = firstJust t
firstJust []           = error "Not found"
```

Pattern matching: вложенные паттерны

Паттерны можно объединять друг с другом в очень мощные конструкции

```
data Term = Constant Integer
          | Variable String
          | SumTerm Term Term
simplify (SumTerm (Constant x) (Constant y)) =
                                    Constant (x + y)
simplify (SumTerm x             (Constant 0)) =
                                    simplify x
simplify (SumTerm (Constant 0) x)             =
                                    simplify x
simplify x                                    = x
```

Pattern matching: as-patterns

Иногда нужно применить несколько паттернов к одному выражению

```
-- Если список начинается с 0,
-- вернуть его, иначе добавить 0
addZeroToHead list @ (0 : _) = list
addZeroToHead list = 0 : list
```

Pattern matching: более редкие конструкции

Strict pattern matching — всегда строгий
Lazy pattern matching — всегда ленивый

```
func !x = x -- всегда вычисляет x
func ~(h:t) = h:t -- не вычисляет h и t,
-- даже если список пустой
```

Pattern matching: summary

- Одна из самых мощных особенностей функциональных языков
- Вся мощность заключается в комбинаторных возможностях
- Было несколько попыток затащить в mainstream языки, получилось не очень
    - В Scala есть
    - В Rust были, сейчас порезаны
    - В C# есть вечно ждущий одобрения патч
    - В Python/Kotlin урезаны до destructurin assignment

Point-free Notation

Один из основных принципов «хорошего» кода на Haskell — point-free notation, или избегание скобок в любом их виде

Основные средства:

- Каррирование и секции
- `let`
- `($)` — применение функции к аргументу `f $ x = f x`
- `(.)` — композиция функций: `(f . g) x === f (g x)`

Point-free Notation

```haskell
foo x = f (g (h x))
foo x = let x' = h x
            x'' = g x'
        in f x''
-- $ правоассоциативен и имеет
-- самый низкий приоритет
foo x = f $ g $ h x
foo = f . g . h
```

Функциональные комбинаторы: рекурсия

Y-комбинатор, в haskell называется `fix`:

```haskell
fix :: (t -> t) -> t
fix f = f (fix f)
```

Зачем он нужен?

Функциональные комбинаторы: рекурсия

Y-комбинатор, в haskell называется `fix`:

```
fix :: (t -> t) -> t
fix f = f (fix f)
```

Зачем он нужен?

Комбинатор наименьшей неподвижной точки:
применяет функцию до тех пор, пока она сама не решит «остановиться»

Функциональные комбинаторы: рекурсия

```haskell
-- Заметьте, функция factorial' не рекурсивна
factorial' recursion i =
    if i <= 1 then 1
              else i * recursion (i - 1)
-- Заметьте, функция factorial тоже не рекурсивна
factorial i = fix factorial'
```

Функциональные комбинаторы над списками

```haskell
-- применить функцию к каждому элементу списка,
-- вернуть список результатов
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (h : t) = (f h) : (map f t)
-- то же самое, но на выходе получается набор
-- списков, объединяем их вместе
flatMap :: (a -> [b]) -> [a] -> [b]
flatMap f [] = []
flatMap f (h : t) = (f h) `append` (flatMap f t)
```

Функциональные комбинаторы над списками

```haskell
-- выкинуть из списка все элементы,
-- не соответствующие предикату
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (h:t) | p h = h : ft
               | otherwise = ft
              where ft = filter p t
```

Функциональные комбинаторы над списками

```haskell
-- слепить два списка в список пар
zip :: [a] -> [b] -> [(a,b)]
zip (ah: at) (bh: bt) = (ah, bh) : zip at bt
-- Взять первые N элементов списка
take :: Integer -> [a] -> [a]
take 0 _ = []
take i [] = []
take i (h: t) = h : take (i-1) t
-- Выбросить первые N элементов списка
drop :: Integer -> [a] -> [a]
drop 0 lst = lst
drop i [] = []
drop i (_: t) = drop (i-1) t
```

Функциональные комбинаторы над списками: свёртки

Правая свёртка:

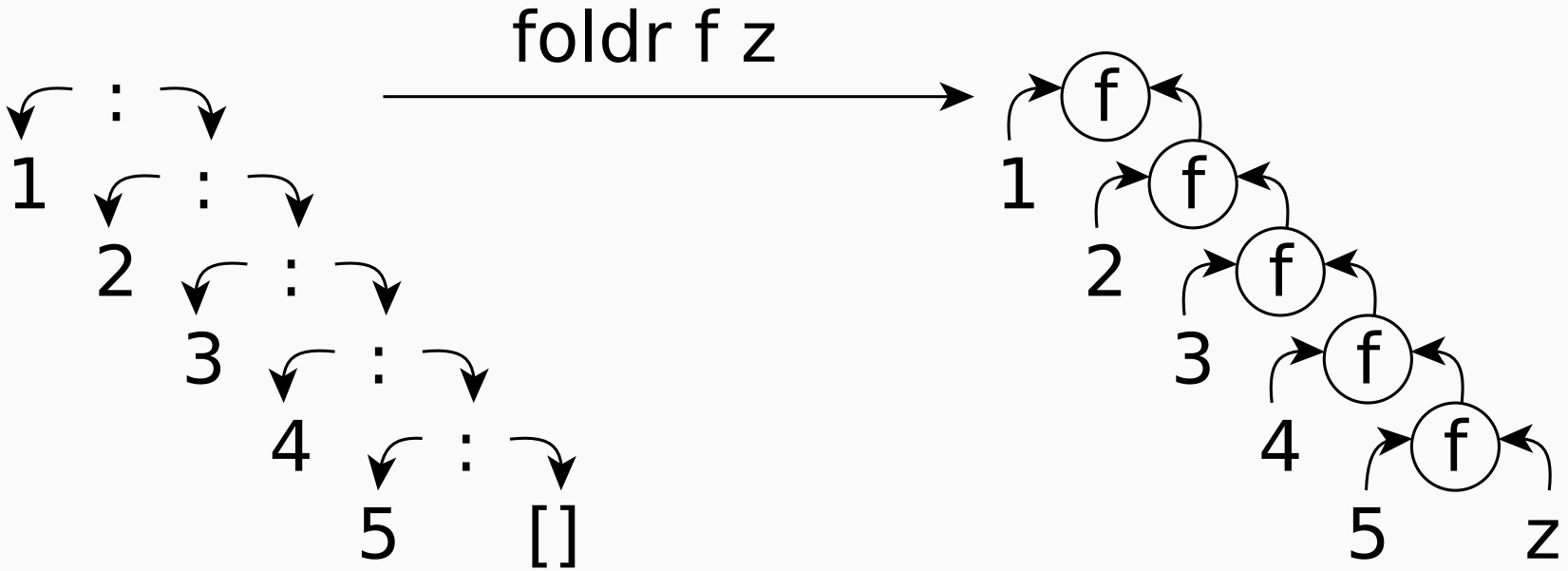
```
foldr :: (a -> b -> b) -> b -> [a] -> b


sum list = foldr (+) 0 list
```

Функциональные комбинаторы над списками: свёртки

Правая свёртка:

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

```
sum list = foldr (+) 0 list
```



Функциональные комбинаторы над списками: свёртки

Левая свёртка:

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

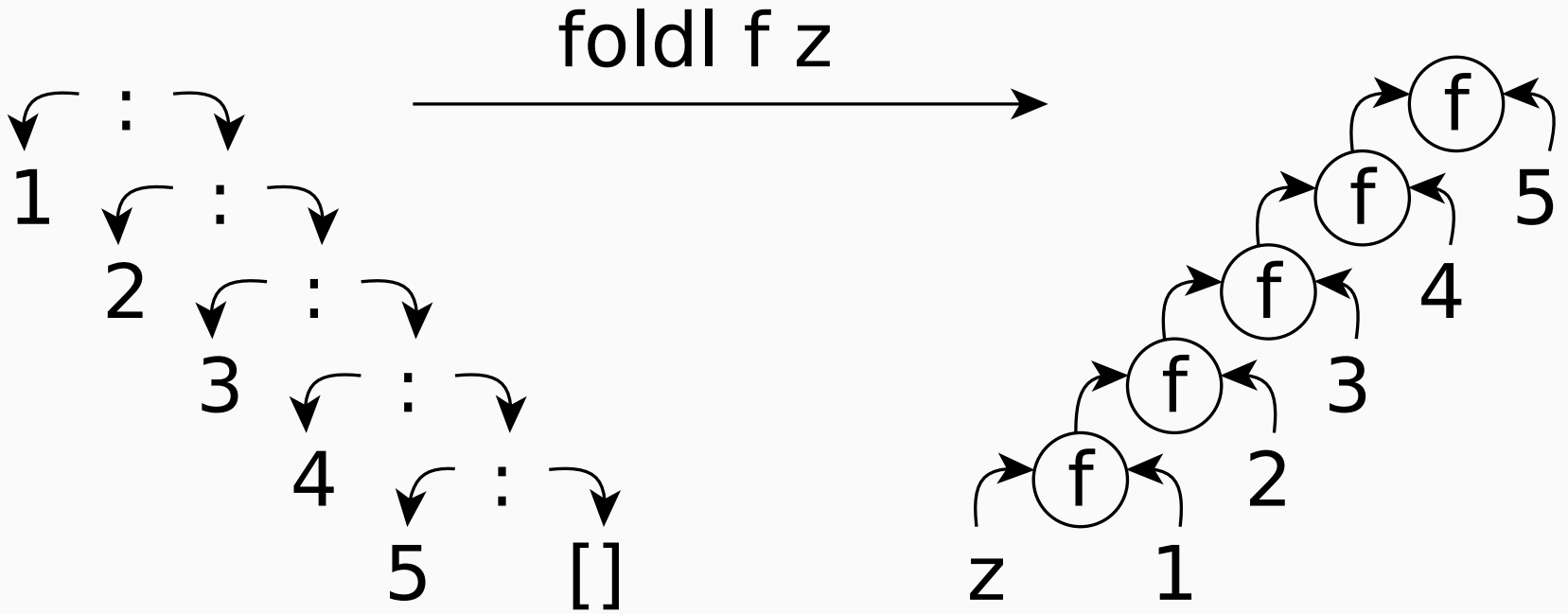
```
sum list = foldl (+) 0 list
```

Функциональные комбинаторы над списками: свёртки

Левая свёртка:

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

```
sum list = foldl (+) 0 list
```



Свёртки — это непривычная замена циклам `foreach`

```haskell
-- поиск длины списка
lstLength :: [a] -> Integer
-- аккумулятор имеет тип Integer
lstLength = foldl
              action -- действие над аккумулятором
              0 -- начальное значение аккумулятора
            where action acc element = acc + 1
```

Другие комбинаторы над списками

```
unfoldr :: (b -> Maybe(a, b)) -> b -> [a]
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
tails :: [a] -> [[a]]
```

Мораль