

Pattern matching & list combinators

Михаил Беляев

14 октября 2015

Pattern matching: простые примеры

-- Списки

```
cons h t = h:t
```

```
car (h:t) = h
```

```
car [] = error "empty list"
```

```
cdr (h:t) = t
```

```
cdr [] = []
```

-- Кортежи

```
pair a b = (a,b)
```

```
first (a,b) = a
```

```
second (a,b) = b
```

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

Pattern matching: внутри let

```
-- 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

```
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)

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
```

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
```

String 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 должен
```

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 $f \ \$ \ x = f \ x$
- `(\cdot)` — композиция функций: $(f \ \cdot \ g) \ x == f \ (g \ x)$

```
foo x = f (g (h x))
```

-- $\$$ правоассоциативен и имеет самый низкий приоритет

```
foo x = f $ g $ h x
```

```
foo = f . g . h
```

Y-комбинатор, в haskell называется `fix`:

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

Зачем он нужен?

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

```
-- 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

*-- 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)
```

-- выкинуть из списка все элементы, не соответствующие пред

```
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

```
-- 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
```

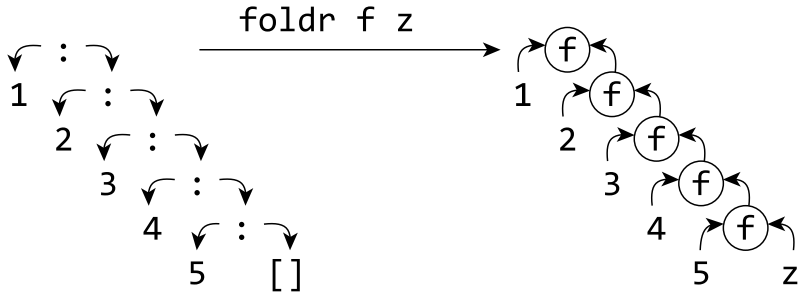


Рис. 1:foldr

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
```

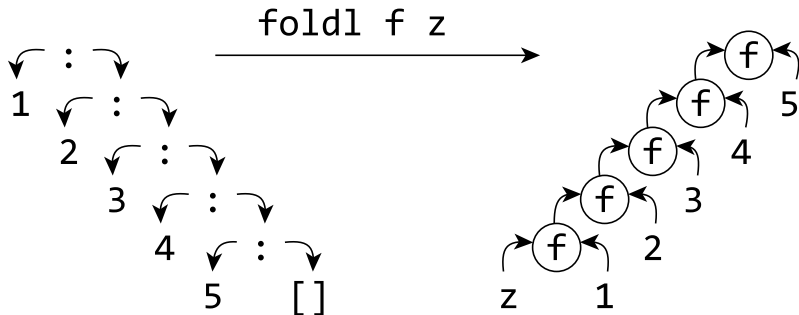


Рис. 2:foldl

Свёртки — это непривычная замена циклам foreach

```
-- 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
```