Pattern matching и функциональные комбинаторы

Михаил Беляев

25 сентября 2018

Pattern matching

```
foo x = x + 2
  --^ это паттерн
```

Паттерны решают две задачи:

- Проверка применимости к аргументу
- Разбор аргумента на части

Pattern matching

Множественные определения функции с разными паттернами

```
factorial 0 = 1
factorial 1 = 1
factorial n = n * factorial (n - 1)
```

Базовые паттерны

```haskell
-- Переменная
foo x = x + 2
-- Игнорирование
bar _ = 5
-- Константа
fee 3 5 = 15
-- Комбинации
fuu x y _ 4 = x + y
```

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

```haskell
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: записи

Для записей есть специальный синтаксис:

```
data Coords = Coords { x::Int, y::Int }
xUnit Coords { x = x } = Coords x 0
isAtXAxis Coords { y = 0 } = True
isAtXAxis _ = False
```

Pattern matching: объявления функций

- Каждое объявление функции с новым паттерном обходится в порядке объявления
- Поэтому порядок важен
- Общее правило: сначала частные случаи, потом всё остальное

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: вложенные паттерны

Паттерны можно объединять друг с другом в очень мощные конструкции

```
data Line = Line { start::Coords, end::Coords }
isVertical Line {
                start = Coords{x = x1},
                end = Coords{x = x2}
            } | x1 == x2 = True
isVertical _ = False
```

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 Swift есть
    - В Rust были, сейчас порезаны
    - В C# есть ~~вечно ждущий одобрения патч~~ просто есть!
    - В Java есть JEP
    - В Python/Kotlin урезаны до destructuring assignment

Pattern matching: расширения

`-XViewPatterns`

Позволяет использовать как паттерн любую функцию

`(function -> pattern)`

```haskell
startsWithTwo :: Int -> Int
startsWithTwo (show -> ('2':_)) = True
startsWithTwo _ = False
```

Pattern matching: расширения

`-XPatternSynonyms`

Вариант 1:

- Позволяет делать свои паттерны на основе имеющихся
- Рекурсию нельзя =(
- Паттерны с пропусками нельзя

```
pattern MAXINT = 9223372036854775807
pattern SingletonList x = [x]

foo (SingletonList x) = x
foo _ = error ""
```

Pattern matching: расширения

`-XPatternSynonyms`

Вариант 2:

- Паттерны с пропусками можно

```
pattern StartsWithA <- 'a' : _
          where StartsWithA = ['a']
```

Pattern matching: другие языки

- В языках семейства ML популярны т.н. or-паттерны

```
// это код на F#
foo x = match x with
        | (0, y) | (y, 0) -> y
        | _ -> error ""
```

- Сравнение с несколькими паттернами по очереди
- Набор переменных во всех должен совпадать
- В haskell их нет из-за сложности восприятия

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` и `where`
- `($)` — 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
```

Point-free Notation

```haskell
factorial 0 = 1
factorial 1 = 1
factorial n = n * factorial (n - 1)

-- Читать числа построчно и выводить их факториалы
show.factorial.read
```

Point-free Notation

- В других языках (например, F#), популярен оператор `(|>)` (конвейерный оператор)
- Почему? Потому что IDE

```
(|>) x f = f x
infixl 0 |>

[1..5] |> (!! 3) |> (+1) |> (*2) |> show -- "10"
```

Функциональные комбинаторы: самые простые

```
const x _ = x
id x = x
flip f a b = f b a
```

Функциональные комбинаторы: рекурсия

Y-комбинатор, в haskell называется `fix`:

```
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 = 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
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f [] = []
concatMap f (h : t) = (f h) `append` (concatMap 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
zip [] [] = []
-- 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:

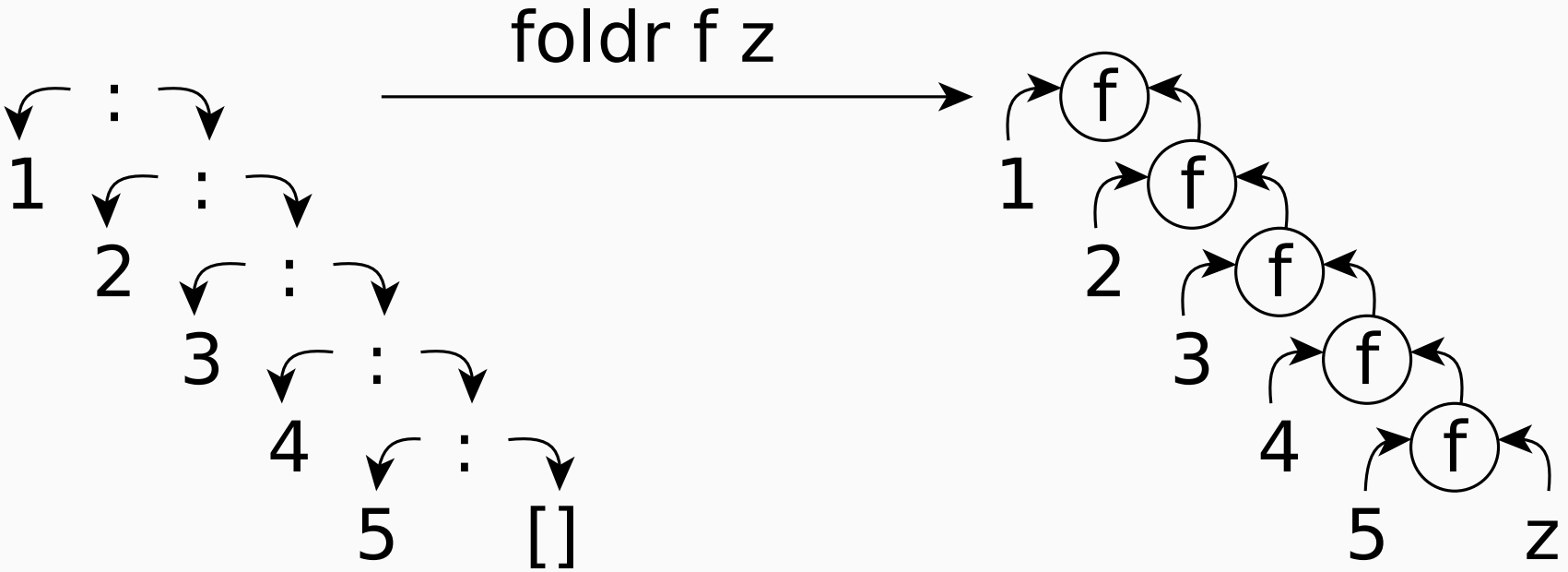
```
foldr :: (a -> b -> b) -> b -> [a] -> b


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
```

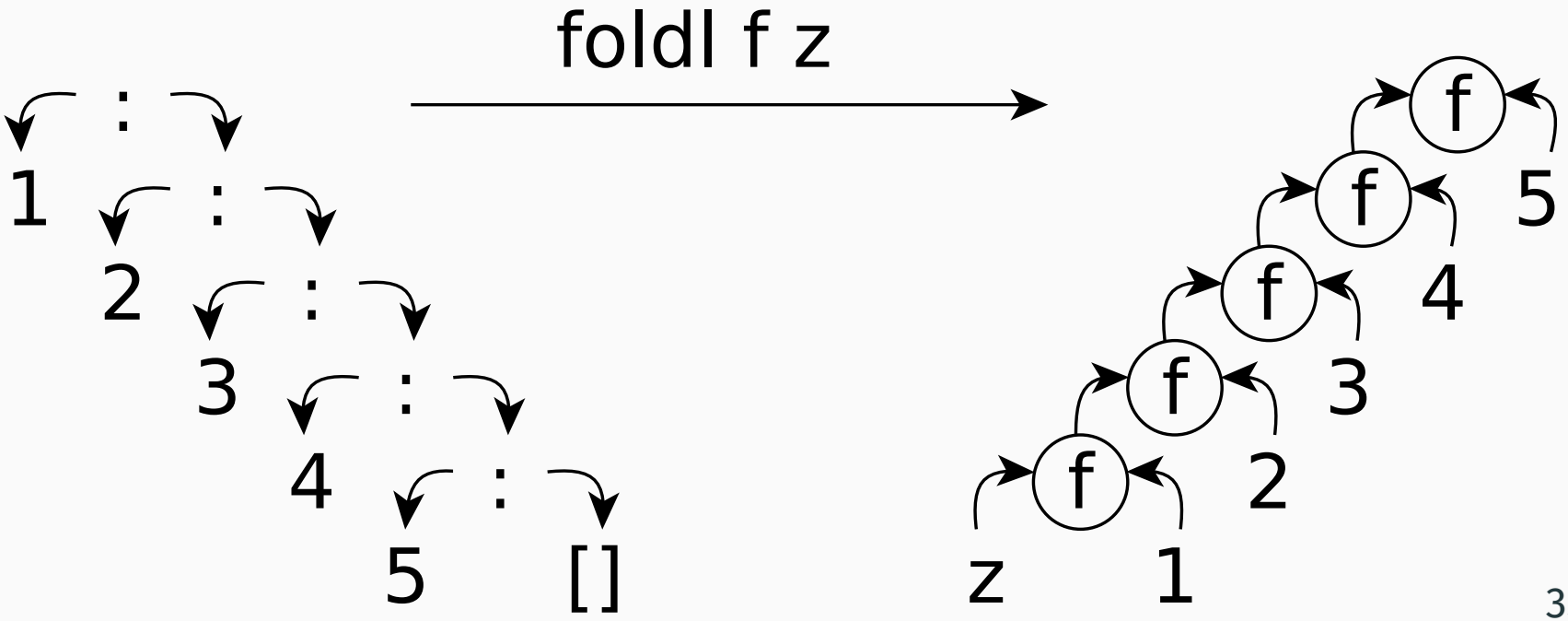
```
sum list = foldl (+) 0 list
```

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
```

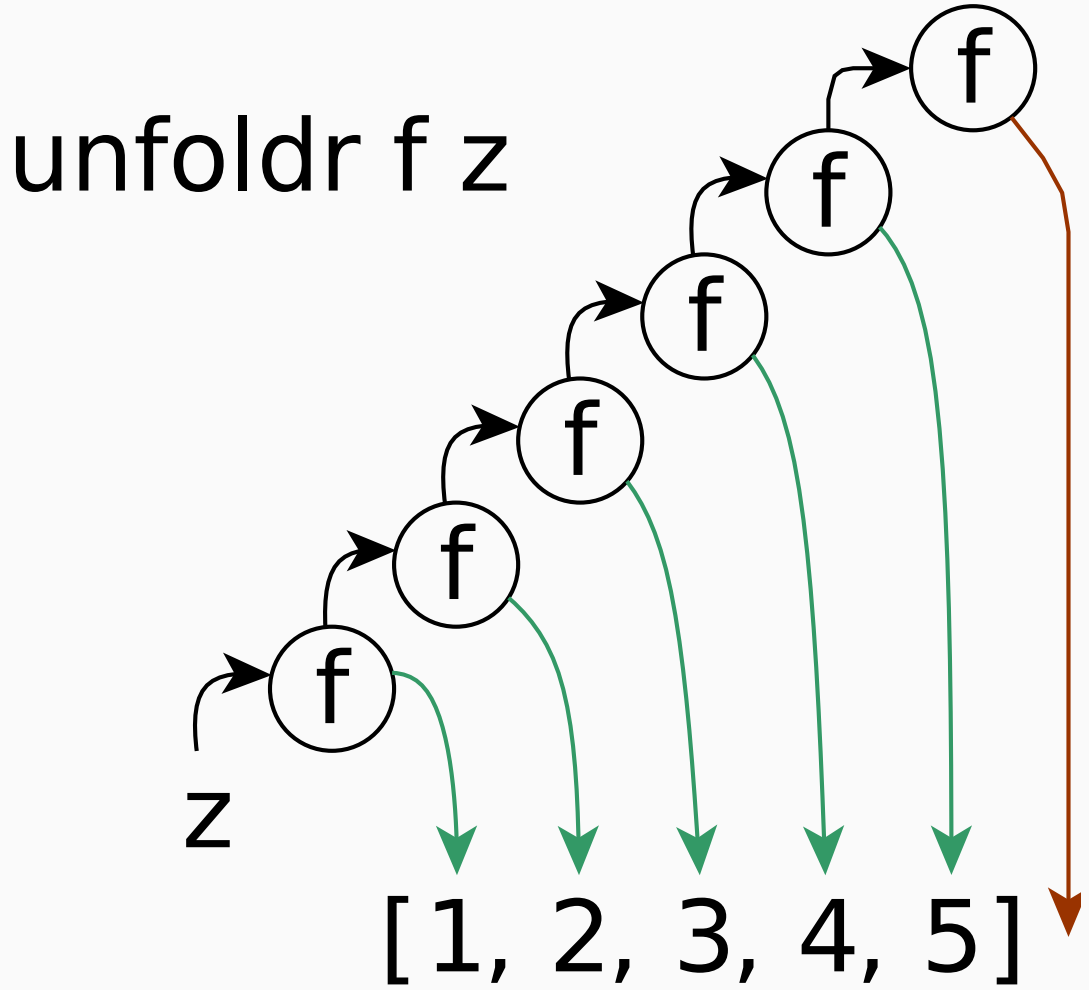
Свёртки

На самом деле, используя свёртки и лямбды, можно выразить любые другие функции над списками из этой лекции

См. домашнее задание

Развёртки: генераторы списков

```
unfoldr :: (b -> Maybe(a, b)) -> b -> [a]
```



Развёртки: генераторы списков

```
unfoldr :: (b -> Maybe(a, b)) -> b -> [a]

take 5 $ unfoldr (\n -> Just (show n, n + 1)) 0
["0", "1", "2", "3", "4"]
```

Origami programming

Есть строгое доказательство, что используя только `fold`, `unfold` и лямбды, можно выразить любой базовый обработчик списков

Этот принцип называется оригами-программированием

Другие комбинаторы над списками

```
scanl :: (b -> a -> b) -> b -> [a] -> [b]
scanr :: (a -> b -> b) -> b -> [a] -> [b]
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
tails :: [a] -> [[a]]

scanl (+) 1 [1..10]
> [1,1,2,6,24,120,720,5040,40320,362880,3628800]
```

Всё вместе

```haskell
-- Напишите функцию, которая находит N-ю цифру
-- в ряду квадратов, записанных в строку
squareChar n =
    let square x = x * x
        squares = concatMap (show.square) [1..]
    in squares !! n
```

Комбинаторы над списками: summary

- Один из базовых принципов ФП — комбинаторность
  - Т.е. сборка программы из универсальных кирпичиков
  - Паттерн матчинг
  - Комбинаторы
  - Лямбда-выражения
  - Каррирование и секции

http://kspt.icc.spbstu.ru/course/lang

belyaev@kspt.icc.spbstu.ru

КСПТ | КАФЕДРА КОМПЬЮТЕРНЫХ СИСТЕМ И ПРОГРАММНЫХ ТЕХНОЛОГИЙ СПбПУ

ПОЛИТЕХ
Санкт-Петербургский политехнический университет Петра Великого