CS212. Парадигмы и технологии программирования: часть 1, функциональное программирование

Лекция 2. Обработка списков и функции высших порядков

В. Н. Брагилевский

21 февраля 2019 г.

Направление «Фундаментальная информатика и информационные технологии»
Институт математики, механики и компьютерных наук имени И. И. Воровича
Южный федеральный университет

Списки в языке Haskell

Фрагмент исходного кода (lists.hs)

```
xs = [1,3,9]
ys = 5 : xs
zs = xs ++ ys
```

Сессия ghci

```
ghci> :load lists
ghci> xs
[1,3,9]
ghci> ys
[5,1,3,9]
ghci> zs
[1,3,9,5,1,3,9]
```

Простейшие функции для обработки списков

Доступ к элементам списка

```haskell
head :: [a] -> a        -- голова списка
tail :: [a] -> [a]      -- хвост списка
last :: [a] -> a        -- последний элемент списка
init :: [a] -> [a]      -- список без последнего
                        -- элемента

-- Первые n элементов
take :: Int -> [a] -> [a]
-- Все элементы кроме n первых
drop :: Int -> [a] -> [a]

splitAt :: Int -> [a] -> ([a], [a])
```

Простейший анализ содержимого списков

```haskell
length :: Foldable t => t a -> Int
-- пуст ли список?
null :: Foldable t => t a -> Bool
-- содержится ли элемент в списке?
elem :: (Eq a, Foldable t) => a -> t a -> Bool

sum, product :: (Num a, Foldable t) => t a -> a

maximum :: (Ord a, Foldable t) => t a -> a
minimum :: (Ord a, Foldable t) => t a -> a

and, or ::  Foldable t => t Bool -> Bool
```

Формирование списков

```haskell
reverse :: [a] -> [a]  -- обращение списка
-- соединение списков в один
concat :: Foldable t => t [a] -> [a]
-- бесконечное повторение элемента
repeat :: a -> [a]
-- n-кратное повторение элемента
replicate :: Int -> a -> [a]
-- бесконечное повторение списка
cycle :: [a] -> [a]
-- спаривание элементов списков
zip :: [a] -> [b] -> [(a, b)]
-- разъединение пар
unzip :: [(a, b)] -> ([a], [b])
```

Функции на списках символов (строках)

```haskell
-- разделение на строки (по \n)
lines :: String -> [String]

--разбиение на слова (по пробелам)
words :: String -> [String]

unlines :: [String] -> String
unwords :: [String] -> String
```

ФВП для работы со списками

Функция map: определение и примеры

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

```
ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
ghci> map (++ "!") ["БУХ", "БАХ", "ПАФ"]
["БУХ!","БАХ!","ПАФ!"]
ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]

ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]
```

Функция filter: определение и примеры

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

ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
ghci> filter (\x -> x `mod` 3 == 0) [1..10]
[3,6,9]
ghci> filter (<15) $ filter even [1..20]
[2,4,6,8,10,12,14]
```

Пример: «быстрая» сортировка

```haskell
quicksort :: Ord a => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = lower ++ [x] ++ upper
  where
    lower = quicksort $ filter (<= x) xs
    upper = quicksort $ filter (> x) xs
```

Пример: сумма квадратов

Задача

Найти сумму квадратов чисел из диапазона от 1 до 100, делящихся на 3 или 5.

Алгоритм решения

1. Формируем список чисел.
2. Оставляем числа, делящиеся на 3 или 5.
3. Возводим каждое число в квадрат.
4. Вычисляем сумму.

```
ghci> xs = [1..100]
ghci> ys = filter (\n -> (mod n 3) * (mod n 5) == 0) xs
ghci> sum $ map (^2) ys
164036
```

Нотация для формирования списков

```
ghci> xs = [1..100]
ghci> ys = filter (\n -> (mod n 3) * (mod n 5) == 0) xs
ghci> sum $ map (^2) ys
164036
```

Генераторы списков (list comprehensions)

```
ghci> sum [ n^2 | n <- [1..100],
                  (mod n 3) * (mod n 5) == 0]
164036
```

$$\{n^2 \mid n \in [1..100], n \text{ делится на } 3 \text{ или } 5\}$$

Пример: поиск числа

Задача

Найти наибольшее число, меньшее 100000, которое делится на 3829.

Алгоритм решения

1. Формируем список кандидатов (в порядке убывания).
2. Фильтруем список, оставляя только те, которые делятся на 3829.
3. Берём первый элемент получившегося списка (head).

Пример: поиск числа

Решение

```
largestDivisible :: Integer -> Integer
                            -> Integer
largestDivisible divisor lim =
     head $ filter (\x -> mod x divisor == 0)
                    [lim, lim-1..]

ghci> largestDivisible 3829 100000
99554
```

Пример: поиск числа

Версия решения с использованием сечений и композиции

```
largestDivisible divisor lim =
    head $ filter ((==0).(`mod` divisor))
                  [lim,lim-1..]
```

И ещё одна версия

```
largestDivisible divisor =
    head . filter ((==0).(`mod` divisor))
         . iterate (subtract 1)
```

Функция iterate

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

ghci> take 6 $ iterate (^2) 2
[2,4,16,256,65536,4294967296]
ghci> take 7 $ iterate ('x':) ""
["","x","xx","xxx","xxxx","xxxxx", "xxxxxx"]
```

Метод Ньютона: решение на списках

Общие идеи метода Ньютона

- Начальное приближение.
- Имеется способ улучшать приближение (improve) $\Longrightarrow$ список приближений, вообще говоря бесконечный.
- Имеется способ находить подходящее приближение (goodEnough).

```
newton improve check eps y =
                        findFirst goodEnough (guesses 1)
 where
  findFirst pred xs = ... -- найти первый элемент,
                          -- удовлетворяющий предикату
  goodEnough x = ...   -- подходит ли приближение?
  guesses x = ...      -- список приближений
```

Метод Ньютона: решение на списках

Общие идеи метода Ньютона

- Начальное приближение.
- Имеется способ улучшать приближение (improve) $\Longrightarrow$ список приближений, вообще говоря бесконечный.
- Имеется способ находить подходящее приближение (goodEnough).

```
newton improve check eps y =
          head $ filter goodEnough guesses
 where
   goodEnough x = abs (check x - y) < eps
   guesses = iterate improve 1
```

Функции takeWhile и dropWhile

Определение takeWhile

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

Определение dropWhile

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

Функции takeWhile и dropWhile: примеры

```
ghci> takeWhile (<10) [2,4..]
[2,4,6,8]
ghci> dropWhile (<10) [2,4..20]
[10,12,14,16,18,20]
ghci> takeWhile (/=' ') "hello world"
"hello"
ghci> dropWhile (`elem` ['a'..'z']) "hello world"
" world"
```

Сравнение filter и takeWhile на бесконечных списках

```
ghci> takeWhile (<10) [2,4..]
[2,4,6,8]
ghci> filter (<10) [2,4..]
????

ghci> filter (<10) [2,4..]
[2,4,6,8
```

Функция zipWith

Определение

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

Функция zipWith

```
ghci> zipWith (+) [4,2,5,6] [2,6,2,3]
[6,8,7,9]
ghci> zipWith max [6,3,2,1] [7,3,1,5]
[7,3,2,5]
ghci> zipWith (*) [2,2,2,2,2] [1..]
[2,4,6,8,10]
ghci> zipWith (++) ["Hypno ", "Flareon "]
                   ["Psychic", "Fire"]
["Hypno Psychic","Flareon Fire"]
ghci> zipWith (zipWith (*))
              [[1,2,3],[3,5,6],[2,3,4]]
              [[3,2,2],[3,4,5],[5,4,3]]
[[3,4,6],[9,20,30],[10,12,12]]
```

Функция zipWith

Применение

```haskell
zip' :: [a] -> [b] -> [(a,b)]
zip' = zipWith (,)

sup :: Ord a => [a] -> [a] -> [a]
sup = zipWith max
```

Примеры

```
ghci> zip' [1,2,3] [2,4..]
[(1,2),(2,4),(3,6)]
ghci> sup [10, 2, 5] [3, 5, 7]
[10,5,7]
```

Некоторые функции обработки списков

```
??? :: (a -> Bool) -> [a] -> Bool
??? :: (a -> Bool) -> [a] -> ([a], [a])
??? :: (a -> Bool) -> [a] -> [Int]
??? :: (a -> a -> Bool) -> [a] -> [a]
??? :: (a -> a -> Bool) -> [a] -> [[a]]
??? :: (a -> b -> a) -> a -> [b] -> a
??? :: (a -> b -> a) -> a -> [b] -> [a]
```

Hoogle — поиск по функциям

URL: `http://hoogle.haskell.org`

Примеры поисковых запросов:

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

Свёртки

Pascal

```
(* arr - массив*)
s := 0;
for i:= 1 to 10 do
  s := s + arr[i];
```

C#

```
// numbers - IEnumerable
s = 0;
foreach (n in numbers)
  s += n;
```

Основные компоненты кода

- Проход по структуре данных или диапазону (цикл).
- Аккумулирующая переменная.
- Текущее значение.
- Вычисление в теле цикла.

Свёртки

Левая и правая свёртки

Определение

Свёртки — это семейство ФВП, обрабатывающих все компоненты рекурсивной структуры данных и вычисляющих в результате некоторое значение (аккумулятор). Обычно свёртка задаётся комбинирующей функцией, структурой данных и, возможно, начальным значением аккумулятора.

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

foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
foldr _ z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Порядок сворачивания списка

Левая свёртка

```
foldl g z [3,4,5,6] === g (g (g (g z 3) 4) 5) 6
```

Правая свёртка

```
foldr f z [3,4,5,6] === f 3 (f 4 (f 5 (f 6 z)))
```

Простейшие примеры

```haskell
sum' :: Num a => [a] -> a
sum' xs = foldl (+) 0 xs

sum'' :: Num a => [a] -> a
sum'' = foldl (+) 0

product' :: Num a => [a] -> a
product' = foldl (*) 1

elem' :: Eq a => a -> [a] -> Bool
elem' y ys =
    foldl (\acc x -> if x == y then True else acc)
          False
          ys
```

Примеры

Количество положительных элементов списка

```
countPositive =
  foldl (\c x -> c + if x > 0 then 1 else 0) 0
```

Сумма и произведение элементов списка

```
sumprod = foldl (\(s,p) x -> (s+x, p*x)) (0, 1)
```

Среднее арифметическое элементов списка

```
mean xs = sum / len
  where
    step (s, l) x = (s+x, l+1)
    (sum, len) = foldl step (0, 0) xs
```

Построение списков свёртками

```haskell
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr (\x acc -> f x : acc) [] xs

map'' :: (a -> b) -> [a] -> [b]
map'' f xs = foldr ((:).f) [] xs

filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr step []
  where
    step x acc
      | p x = x : acc
      | otherwise = acc
```

Построение списков свёртками: reverse

```haskell
reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []

reverse'' :: [a] -> [a]
reverse'' = foldl (flip (:)) []
-- flip :: (a -> b -> c) -> b -> a -> c
```

Функции foldl1 и foldr1

Определение

```haskell
foldl1 :: Foldable t => (a -> a -> a) -> t a -> a
foldl1 f (x:xs) =  foldl f x xs
foldl1 _ [] =  error "foldl1: empty list"

foldr1 :: Foldable t => (a -> a -> a) -> t a -> a
foldr1 _ [x] =  x
foldr1 f (x:xs) =  f x (foldr1 f xs)
foldr1 _ [] =  error "foldr1: empty list"
```

Простейшие примеры

```haskell
maximum' :: (Ord a) => [a] -> a
maximum' = foldl1 max

last' :: [a] -> a
last' = foldl1 (\ _ x -> x)
```

Свёртка бесконечных списков

Функция and'

```
and' :: [Bool] -> Bool
and' xs = foldr (&&) True xs
```

Порядок вычисления

```
and' [True,False,True]
    === True && (False && (True && True))
```

Случай бесконечного списка

```
and' (repeat False)
     === False && (False && (False && (False ...
```

Как запомнить?

- «Северный ветер дует с севера»

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