Модуль 2 · Часть 2

Функции высшего порядка

map, filter, fold и композиция функций

Курс «Функциональное и логическое программирование на Haskell»

Что мы изучим сегодня

Функции высшего порядка — ключевой инструмент функционального программирования

Переписать императивные алгоритмы

Сортировка, поиск максимума, подсчёт — через map, filter, fold

Реализовать свои map, filter, foldl

Собственные версии стандартных функций через рекурсию

Функции высшего порядка принимают другие функции как аргументы или возвращают их как результат

Что такое функции высшего порядка?

Higher-Order Functions — основа функционального стиля

Определение

Функция высшего порядка (HOF) — функция, которая принимает другую функцию как аргумент и/или возвращает функцию как результат.

Знакомые примеры

`map` — применить функцию к каждому элементу

`filter` — отобрать элементы по предикату

`foldl/foldr` — свернуть список в значение

Сигнатуры типов

```
-- Типы функций высшего порядка
map    :: (a -> b) -> [a] -> [b]
filter :: (a -> Bool) -> [a] -> [a]
foldl  :: (b -> a -> b) -> b -> [a] -> b
foldr  :: (a -> b -> b) -> b -> [a] -> b
```

В Haskell функции — значения первого класса: их можно передавать и возвращать как любые другие данные

Задача 1: map — преобразование списков

Применение функции к каждому элементу

Императивный подход

```
result = []
for x in xs:
    result.append(f(x))
return result
```

Функциональный подход

```
map f xs
```

Реализация на Haskell

```
-- Примеры использования map
map (*2) [1,2,3,4,5]          -- [2,4,6,8,10]
map toUpper "hello"            -- "HELLO"
map (\x -> x*x + 1) [1..5]    -- [2,5,10,17,26]
map show [1,2,3]              -- ["1","2","3"]

-- Вложенный map
map (map (*2)) [[1,2],[3,4]]  -- [[2,4],[6,8]]
```

map сохраняет структуру списка: длина результата всегда равна длине аргумента

Задача 2: filter — фильтрация списков

Отбор элементов по предикату

Как работает filter

`filter p xs` оставляет только те элементы
`x`, для которых предикат
`p x == True`

Императивный аналог

```
result = []
for x in xs:
    if p(x):
        result.append(x)
```

Реализация на Haskell

```
-- Примеры использования filter
filter even [1..10]              -- [2,4,6,8,10]
filter (>3) [1,5,2,8,3]          -- [5,8]
filter (\s -> length s > 3)
  ["hi","hello","ok","world"]
-- ["hello","world"]

-- Комбинация map и filter
map (*2) (filter even [1..10])    -- [4,8,12,16,20]
```

filter не изменяет элементы — только решает, включать их в результат или нет

Задача 3: foldl и foldr — свёртка списков

Сведение списка к одному значению

foldl — свёртка слева

```
foldl f z [x1,x2,x3]

= f (f (f z x1) x2) x3
```

Порядок: (((z ⊕ x1) ⊕ x2) ⊕ x3)
Накапливает результат слева направо

foldr — свёртка справа

```
foldr f z [x1,x2,x3]

= f x1 (f x2 (f x3 z))
```

Порядок: (x1 ⊕ (x2 ⊕ (x3 ⊕ z)))
Строит выражение справа налево

Реализация на Haskell

```haskell
-- Примеры свёрток
foldl (+) 0 [1,2,3,4,5]     -- 15 (сумма)
foldl (*) 1 [1,2,3,4,5]     -- 120 (факториал)
foldl max 0 [3,1,4,1,5,9]   -- 9 (максимум)
foldr (:) [] [1,2,3]         -- [1,2,3] (копия списка)
foldr (++) "" ["a","b","c"]  -- "abc" (конкатенация)
```

Используйте foldl' (строгий) вместо foldl для больших списков — избежите утечек памяти

Переписываем алгоритмы: сортировка

Quicksort через filter — элегантность ФП

Идея quicksort на Haskell

1. Базовый случай: пустой список
2. Выбираем опорный элемент (pivot)
3. filter (<= pivot) — левая часть
4. filter (> pivot) — правая часть
5. Рекурсивно сортируем и соединяем

Реализация

```haskell
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x:xs) =
  qsort smaller ++ [x] ++ qsort bigger
  where
    smaller = filter (<= x) xs
    bigger  = filter (>  x) xs
```

Поиск максимума и подсчёт элементов

```haskell
-- Поиск максимума через foldl1
maximum' :: Ord a => [a] -> a
maximum' = foldl1 max

-- Подсчёт элементов, удовлетворяющих условию
countIf :: (a -> Bool) -> [a] -> Int
countIf p = length . filter p

-- Пример: сколько чётных?
countIf even [1..10]  -- 5
```

Композиция (.) позволяет строить конвейеры обработки данных без промежуточных переменных

Композиция функций

Оператор (.) и оператор ($) — конвейеры обработки данных

Оператор (.)

```
(.) :: (b -> c) -> (a -> b) -> a -> c

(f . g) x = f (g x)
```

Читается справа налево:
сначала применяем g, затем f

Оператор ($)

```
($) :: (a -> b) -> a -> b

f $ x = f x
```

Устраняет скобки:
f $ g $ h x вместо f (g (h x))

Реализация на Haskell

```haskell
-- Сравните:
map (\x -> negate (abs x)) [1,-2,3,-4]    -- [-1,-2,-3,-4]
map (negate . abs) [1,-2,3,-4]            -- [-1,-2,-3,-4]

-- Конвейер обработки данных - "взять список, отфильтровать чётные, удвоить, посчитать сумму"
sumDoubleEvens :: [Int] -> Int
sumDoubleEvens = sum . map (*2) . filter even

sumDoubleEvens [1..10]                    -- 60
```

Композиция позволяет писать в «pointfree» стиле — определять функции без явного упоминания аргументов

Ещё примеры: от циклов к fold

Переписываем типичные императивные паттерны

Примеры переписывания императивных алгоритмов

```haskell
-- Сумма квадратов чётных чисел от 1 до n
sumSqEven :: Int -> Int
sumSqEven n = sum . map (^2) . filter even $
[1..n]
-- sumSqEven 10 = 220

-- Реверс списка через foldl
reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []
-- или короче: foldl (flip (:)) []
```

```haskell
-- Подсчёт вхождений элемента
count :: Eq a => a -> [a] -> Int
count x = length . filter (== x)
-- count 'a' "banana" = 3

-- Все ли элементы удовлетворяют условию?
all' :: (a -> Bool) -> [a] -> Bool
all' p = foldl (\acc x -> acc && p x) True
```

Паттерн: Любой цикл с аккумулятором можно заменить на `foldl`. Цикл с условием — на `filter`. Цикл с преобразованием — на `map`.

Сводка: императивный → функциональный

Основные паттерны замены

Императивный паттерн	Функция	Пример на Haskell
for + transform	`map`	`map (*2) [1..5]`
for + if + append	`filter`	`filter even [1..10]`
for + accumulator	`foldl`	`foldl (+) 0 [1..10]`
построение списка справа	`foldr`	`foldr (:) [] xs`
цепочка преобразований	`(.) / ($)`	`sum . map (*2) . filter even`
max через цикл	`foldl1 max`	`foldl1 max [3,1,4,9]`

Функциональный стиль: описываем ЧТО делать, а не КАК — компилятор оптимизирует сам

Лабораторное задание

Реализуем свои map, filter, foldl

Собственные версии через рекурсию

Реализуем свой map'

Применение функции к каждому элементу через рекурсию

Рассуждение

Базовый случай: пустой список → возвращаем пустой список

Рекурсивный случай: применяем f к голове, рекурсивно обрабатываем хвост

Паттерн структурной рекурсии

1. Деконструируем список: (x:xs)
2. Обрабатываем голову: f x
3. Рекурсивно обрабатываем хвост
4. Собираем результат: (:)

Реализация на Haskell

```
-- Собственная реализация map
map' :: ???
```

Шаблон «_ []» означает: неважно какая функция — для пустого списка результат всегда []

Реализуем свой map'

Применение функции к каждому элементу через рекурсию

Рассуждение

Базовый случай: пустой список → возвращаем пустой список

Рекурсивный случай: применяем f к голове, рекурсивно обрабатываем хвост

Паттерн структурной рекурсии

1. Деконструируем список: (x:xs)
2. Обрабатываем голову: f x
3. Рекурсивно обрабатываем хвост
4. Собираем результат: (:)

Реализация на Haskell

```haskell
-- Собственная реализация map
map' :: (a -> b) -> [a] -> [b]
map' _ []     = []                    -- базовый случай
map' f (x:xs) = f x : map' f xs       -- рекурсивный случай

-- Проверка:
-- map' (*2) [1,2,3]  =  [2,4,6]  ✓
-- map' show [1,2,3]  =  ["1","2","3"]  ✓
```

Шаблон «_ []» означает: неважно какая функция — для пустого списка результат всегда []

Реализуем свой filter'

Отбор элементов по предикату через рекурсию

Рассуждение

Базовый случай: `[] → []`

Рекурсивный случай: проверяем голову предикатом. Если True — включаем, иначе — пропускаем.

Отличие от map'

`map'` всегда включает элемент (преобразованный) в результат.

`filter'` включает элемент только если предикат возвращает True.

Реализация на Haskell

```
-- Собственная реализация filter
filter' :: ???
```

Guards (|) — удобный способ условного ветвления в Haskell, аналог if-else

Реализуем свой filter'

Отбор элементов по предикату через рекурсию

Рассуждение

Базовый случай: `[] → []`

Рекурсивный случай: проверяем голову предикатом. Если True — включаем, иначе — пропускаем.

Отличие от map'

`map'` всегда включает элемент (преобразованный) в результат.

`filter'` включает элемент только если предикат возвращает True.

Реализация на Haskell

```haskell
-- Собственная реализация filter
filter' :: (a -> Bool) -> [a] -> [a]
filter' _ []     = []
filter' p (x:xs)
  | p x       = x : filter' p xs      -- включаем x
  | otherwise = filter' p xs          -- пропускаем x

-- Проверка:
-- filter' even [1..10]  =  [2,4,6,8,10]  ✓
-- filter' (>3) [1,5,2,8] =  [5,8]  ✓
```

Guards (|) — удобный способ условного ветвления в Haskell, аналог if-else

Реализуем свой foldl'

Левая свёртка с аккумулятором через рекурсию

Рассуждение

Базовый случай: пустой список → возвращаем аккумулятор z
Рекурсивный случай: обновляем аккумулятор: (f z x), продолжаем с хвостом
Это хвостовая рекурсия!

Трассировка вычислений

```
foldl' (+) 0 [1,2,3]
= foldl' (+) (0+1) [2,3]
= foldl' (+) 1 [2,3]
= foldl' (+) 3 [3]
= foldl' (+) 6 []
= 6
```

Реализация на Haskell

```haskell
-- Собственная реализация foldl
foldl' :: ???
```

foldl' — самая мощная функция: через неё можно выразить map, filter, length, reverse и многие другие

Реализуем свой foldl'

Левая свёртка с аккумулятором через рекурсию

Рассуждение

Базовый случай: пустой список → возвращаем аккумулятор z
Рекурсивный случай: обновляем аккумулятор: (f z x), продолжаем с хвостом
Это хвостовая рекурсия!

Трассировка вычислений

```
foldl' (+) 0 [1,2,3]
= foldl' (+) (0+1) [2,3]
= foldl' (+) 1 [2,3]
= foldl' (+) 3 [3]
= foldl' (+) 6 []
= 6
```

Реализация на Haskell

```haskell
-- Собственная реализация foldl
foldl' :: (b -> a -> b) -> b -> [a] -> b
foldl' _ z []     = z                  -- вернуть аккумулятор
foldl' f z (x:xs) = foldl' f (f z x) xs -- обновить и продолжить

-- Проверка:
-- foldl' (+) 0 [1..5]  =  15  ✓
-- foldl' (*) 1 [1..5]  =  120  ✓
-- foldl' (\acc x -> x:acc) [] [1,2,3]  =  [3,2,1]  ✓
```

foldl' — самая мощная функция: через неё можно выразить map, filter, length, reverse и многие другие

Бонус: map и filter через foldr

Свёртка — универсальный оператор над списками

Универсальность fold

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

-- filter через foldr
filterF :: (a -> Bool) -> [a] -> [a]
filterF p = foldr (\x acc -> if p x then x:acc else acc) []

-- length через foldl
length' :: [a] -> Int
length' = foldl (\acc _ -> acc + 1) 0

-- concat через foldr
concat' :: [[a]] -> [a]
concat' = foldr (++) []
```

Теорема: любая функция, обрабатывающая список рекурсивно, может быть выражена через `foldr`. Это означает, что fold — это фундаментальная операция над списками, а map и filter — лишь частные случаи.

Итоги занятия

- Функции высшего порядка — ключевой инструмент ФП, принимают/возвращают функции
- map преобразует каждый элемент, filter отбирает по условию, fold сворачивает список
- Любой императивный цикл с аккумулятором заменяется на foldl или foldr
- Композиция (.) и ($) позволяют строить элегантные конвейеры обработки данных
- map, filter и foldl реализуются через структурную рекурсию (паттерн [], (x:xs))
- fold — универсальная операция: через неё выражаются map, filter, length, reverse

Следующее занятие: **Лабораторная работа №1 — сортировка слиянием, quicksort, BFS/DFS в функциональном стиле**

Вопросы?

Модуль 2, Часть 2: Функции высшего порядка