Модуль 2 · Часть 1

Рекурсия: обычная и хвостовая

Классические рекурсивные задачи на Haskell

Курс «Функциональное и логическое программирование на Haskell»

Что мы изучим сегодня

Задачи 2 и 3: рекурсия и классические задачи

1

Обычная рекурсия

Как работает стандартная рекурсия и почему она потребляет много памяти

2

Хвостовая рекурсия

Оптимизация через аккумулятор — константное использование стека

3

Классические задачи

Ханойские башни, обход дерева, бинарный поиск — рекурсия на практике

Рекурсия — основной механизм повторения в функциональном программировании

Рекурсия

Вместо циклов — функция вызывает сама себя

Почему рекурсия?

В Haskell нет циклов for/while. Рекурсия — единственный способ повторного выполнения.

Каждый рекурсивный вызов:

- Базовый случай (остановка)
- Рекурсивный случай (шаг)

Структура рекурсии

```
f базовый_случай = результат
f x = ... f (уменьшение x)
```

Пример: длина списка

```
length' []     = 0
length' (_:xs) = 1 + length' xs
```

Пример: сумма списка — обычная рекурсия

```
sumList :: ???
```

Рекурсия

Вместо циклов — функция вызывает сама себя

Почему рекурсия?

В Haskell нет циклов for/while. Рекурсия — единственный способ повторного выполнения.

Каждый рекурсивный вызов:

- Базовый случай (остановка)
- Рекурсивный случай (шаг)

Структура рекурсии

```
f базовый_случай = результат
f x = ... f (уменьшение x)
```

Пример: длина списка

```
length' []     = 0
length' (_:xs) = 1 + length' xs
```

Пример: сумма списка — обычная рекурсия

```
sumList :: [Int] -> Int
sumList []     = 0                -- базовый случай
sumList (x:xs) = x + sumList xs   -- рекурсивный случай

-- sumList [1,2,3] = 1 + (2 + (3 + 0)) = 6
```

Обычная рекурсия: стек вызовов

Как растёт стек при вычислении sumList [1,2,3,4,5]

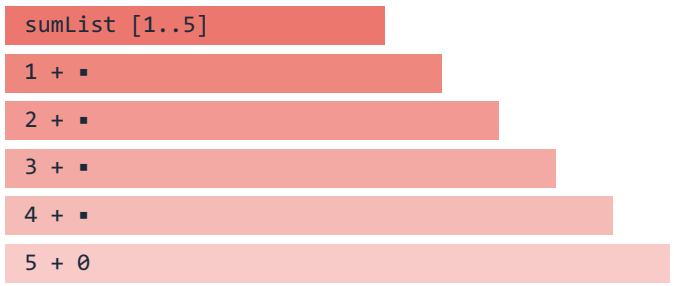
```
sumList [1,2,3,4,5]          →  1 + sumList [2,3,4,5]
  sumList [2,3,4,5]          →  2 + sumList [3,4,5]
    sumList [3,4,5]          →  3 + sumList [4,5]
      sumList [4,5]          →  4 + sumList [5]
        sumList [5]          →  5 + sumList []
          sumList []         →  0  ← базовый случай
```

Разворачивание: 0 → 5+0=5 → 4+5=9 → 3+9=12 → 2+12=14 → 1+14 = **15**

Проблема: потребление памяти

Обычная рекурсия накапливает отложенные вычисления в стеке

Обычная рекурсия: O(n) памяти



Что происходит?

Каждый вызов добавляет кадр в стек.
Вычисление отложено до базового случая.

Для списка из N элементов нужно
N кадров стека одновременно.

На больших данных (10^6+ элементов)
это приводит к Stack Overflow.

Проверка в GHCi: потребление памяти

```
-- Обычная рекурсия: переполнение стека на больших списках
> sumList [1..1000000]
*** Exception: stack overflow

-- :set +s  -- включить статистику времени и памяти в GHCi
```

Решение: хвостовая рекурсия

Tail recursion — рекурсивный вызов является последней операцией

Ключевая идея

Результат накапливается в аккумуляторе (дополнительном параметре), а не в стеке.

Рекурсивный вызов — последняя операция → компилятор может оптимизировать (tail call optimization).

Сравнение

	Обычная	Хвостовая
Память	O(n)	O(1)
Стек	Растёт	Const
Вычисл.	Отложено	Сразу

Обычная рекурсия

```
sumList [] = 0
sumList (x:xs) =
  x + sumList xs
  --  ^^ после рекурсии
  --  ещё нужно сложить
```

Хвостовая рекурсия

```
sumTail xs = go 0 xs
  where
    go acc []     = acc
    go acc (x:xs) = go (acc + x) xs
```

Хвостовая рекурсия: трассировка

sumTail [1,2,3,4,5] — аккумулятор вместо стека

Шаг	Вызов	acc	Список
1	go 0 [1,2,3,4,5]	0	[1,2,3,4,5]
2	go 1 [2,3,4,5]	0+1 = 1	[2,3,4,5]
3	go 3 [3,4,5]	1+2 = 3	[3,4,5]
4	go 6 [4,5]	3+3 = 6	[4,5]
5	go 10 [5]	6+4 = 10	[5]
6	**go 15 []**	**10+5 = 15**	[] ✓

Обратите внимание: на каждом шаге стек не растёт. Предыдущий кадр заменяется новым. Результат
накапливается в acc, а не в отложенных операциях.

Результат: **go 15 [] → acc = 15** (без переполнения стека!)

Практика: измерение в GHCi

Сравнение потребления памяти на больших списках

Обычная рекурсия

```
> :set +s
> sumList [1..100000]
5000050000
(0.08 secs, 18,523,456 bytes)

> sumList [1..10000000]
*** Exception: stack overflow
```

Хвостовая рекурсия

```
> :set +s
> sumTail [1..100000]
5000050000
(0.03 secs, 9,623,120 bytes)

> sumTail [1..10000000]
50000005000000  ✓
```

Совет: foldl' — встроенная строгая свёртка

В Haskell ленивый foldl может накапливать thunks (отложенные вычисления). Используйте строгий foldl' из Data.List для гарантированной оптимизации:

```
import Data.List (foldl')
sum' xs = foldl' (+) 0 xs   -- строгая свёртка, O(1) памяти
```

Задача 3

Классические рекурсивные задачи

Ханойские башни · Обход дерева · Бинарный поиск

Ханойские башни

Классическая задача на рекурсивную декомпозицию

Задача

Переместить n дисков с колышка A на колышек C, используя B как вспомогательный. Диски упорядочены.

Правила:
- Перемещать по одному диску
- Нельзя класть большой на малый

Рекурсивная идея

1. Переместить n-1 дисков A → B
2. Переместить 1 диск A → C
3. Переместить n-1 дисков B → C

Количество ходов: $2^n - 1$

```
type Peg = String;  type Move = (Peg, Peg)

hanoi :: Int -> Peg -> Peg -> Peg -> [Move]
hanoi 0 _ _ _ = []
hanoi n from to aux =
  hanoi (n-1) from aux to   -- n-1 дисков: from → aux
  ++ [(from, to)]           -- 1 диск: from → to
  ++ hanoi (n-1) aux to from -- n-1 дисков: aux → to
```

Ханойские башни: разбор для n = 3

7 ходов = $2^3 - 1$. Прослеживаем каждый рекурсивный вызов

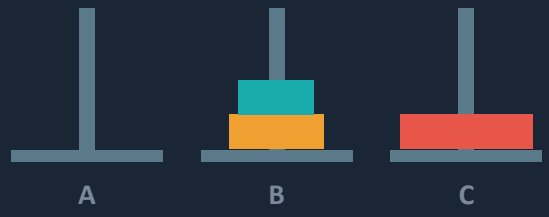
Дерево рекурсивных вызовов

```
hanoi 3 A C B
├─ hanoi 2 A B C
│  ├─ hanoi 1 A C B
│  │  └─ Move A → C          ①
│  ├─ Move A → B             ②
│  └─ hanoi 1 C B A
│     └─ Move C → B          ③
├─ Move A → C                ④
└─ hanoi 2 B C A
   ├─ hanoi 1 B A C
   │  └─ Move B → A          ⑤
   ├─ Move B → C             ⑥
   └─ hanoi 1 A C B
      └─ Move A → C          ⑦
```

Последовательность ходов

Ход	Перемещение	Диск
①	A → C	маленький диск
②	A → B	средний диск
③	C → B	маленький диск
④	**A → C**	**большой диск**
⑤	B → A	маленький диск
⑥	B → C	средний диск
⑦	A → C	маленький диск

После хода ④ — большой диск на месте:



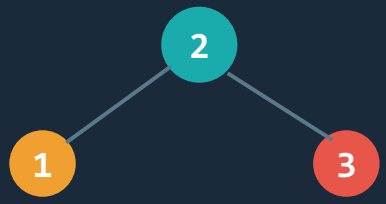
Обход бинарного дерева (DFS, not BFS)

Рекурсивная структура данных → рекурсивный обход

```haskell
-- Определение типа дерева
data Tree a = Leaf
  | Node (Tree a) a (Tree a)
  deriving (Show)

-- Node (Node Leaf 1 Leaf) 2 (Node Leaf 3 Leaf)
```

Визуализация дерева



Inorder

```haskell
inorder Leaf = []
inorder (Node l v r)
  = inorder l
    ++ [v]
    ++ inorder r
-- [1, 2, 3]
```

Preorder

```haskell
preorder Leaf = []
preorder (Node l v r)
  = [v]
    ++ preorder l
    ++ preorder r
-- [2, 1, 3]
```

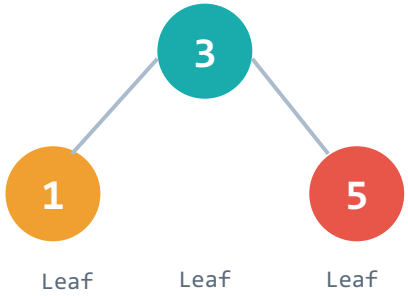
Postorder

```haskell
postorder Leaf = []
postorder (Node l v r)
  = postorder l
    ++ postorder r
    ++ [v]
-- [1, 3, 2]
```

Обход дерева: пошаговая трассировка

Разбираем inorder для дерева Node (Node Leaf 1 Leaf) 3 (Node Leaf 5 Leaf)

Дерево



inorder: левое → корень → правое

Трассировка inorder

```
inorder (Node (Node Leaf 1 Leaf) 3 (Node Leaf 5 Leaf))
= inorder (Node Leaf 1 Leaf) ++ [3] ++ inorder (Node Leaf 5
Leaf)

  inorder (Node Leaf 1 Leaf)  =  [] ++ [1] ++ []  =  [1]
  inorder (Node Leaf 5 Leaf)  =  [] ++ [5] ++ []  =  [5]

= [1] ++ [3] ++ [5]  =  [1, 3, 5]  ✓
```

Обход	Порядок	Трассировка	Результат
Inorder	Лево → Корень → Право	`inorder L ++ [3] ++ inorder R`	`[1, 3, 5]`
Preorder	Корень → Лево → Право	`[3] ++ preorder L ++ preorder R`	`[3, 1, 5]`
Postorder	Лево → Право → Корень	`postorder L ++ postorder R ++ [3]`	`[1, 5, 3]`

Inorder на BST (дереве поиска) всегда даёт отсортированный список!

Бинарный поиск: пошаговый разбор

Ищем число 11 в массиве [1, 3, 5, 7, 9, 11, 13, 15, 17]

Шаг 1

9 < 11 → ищем справа

Шаг 2

13 > 11 → ищем слева

Шаг 3

11 == 11 → Найдено! ✓

3 шага вместо 9 (линейный поиск). Для массива из *n* элементов бинарный поиск делает не более **$\lceil \log_2 n \rceil$** сравнений. Для 1 000 000 — всего 20 шагов!

Бинарный поиск

Поиск элемента в отсортированном списке за O(log n)

Рекурсивная идея

Отсортированный массив делим пополам.
Сравниваем средний элемент с целью:

- Равен → нашли!
- Меньше → ищем в правой половине
- Больше → ищем в левой половине

Каждый шаг уменьшает область вдвое.

Пример: поиск 7

```
[1, 3, 5, 7, 9, 11, 13]
```

`mid = arr[3] = 7`

7 == 7 → Найдено! Индекс 3

Для поиска 3: 7>3 → левая часть
[1, 3, 5] → mid=3 → Найдено!

Реализация на Haskell (с использованием Array)

```haskell
import Data.Array
binarySearch :: (Ord a) => Array Int a -> a -> Maybe Int
binarySearch arr target = go lo hi
  where
    (lo, hi) = bounds arr
    go l h
      | l > h     = Nothing
      | mid == target = Just m
      | mid <  target = go (m+1) h
      | otherwise     = go l (m-1)
      where m = (l + h) `div` 2; mid = arr ! m
```

Бинарный поиск: простой вариант

Версия для списков — менее эффективная, но нагляднее

Бинарный поиск по списку

```haskell
bsearch :: (Ord a) => [a] -> a -> Bool
bsearch [] _ = False
bsearch xs target
  | mid == target = True
  | mid <  target = bsearch right target
  | otherwise     = bsearch left  target
  where
    n     = length xs
    half  = n `div` 2
    left  = take half xs
    right = drop (half + 1) xs
    mid   = xs !! half
```

Важно: разница в эффективности

Версия с Array: O(log n) — прямой доступ по индексу (arr ! i за O(1))
Версия со списком: O(n log n) — length, take, drop, (!!) работают за O(n)

Паттерны рекурсии в Haskell

Обобщение подходов, которые мы изучили

Паттерн	Когда использовать	Пример
Обычная рекурсия	Простые задачи, небольшие данные, наглядность важнее скорости	`factorial, length`
Хвостовая рекурсия	Большие данные, нужна O(1) память, аккумулятор очевиден	`sum, reverse, foldl'`
Разделяй и властвуй	Задача разбивается на подзадачи, результаты объединяются	`mergesort, hanoi, bsearch`
Структурная рекурсия	Обход рекурсивных типов данных (деревья, списки)	`inorder, depth, treeMap`

Полезные рекурсивные функции для практики

```haskell
reverse' []     = []                    -- обычная рекурсия
reverse' (x:xs) = reverse' xs ++ [x]    -- O(n²)

reverse'' xs = go [] xs                  -- хвостовая рекурсия
  where go acc []     = acc
        go acc (x:xs) = go (x:acc) xs    -- O(n)
```

Итоги занятия

- Рекурсия — основной способ повторения в Haskell (вместо циклов)
- Обычная рекурсия проста, но потребляет O(n) памяти на стеке
- Хвостовая рекурсия с аккумулятором — O(1) памяти, используйте foldl'
- Ханойские башни — пример рекурсивной декомпозиции (2^n-1 ходов)
- Обход дерева — структурная рекурсия (inorder, preorder, postorder)
- Бинарный поиск — «разделяй и властвуй», O(log n) с Array

Следующее занятие: **Функции высшего порядка — map, filter, fold, композиция**

Вопросы?

Модуль 2, Часть 1: Рекурсия и классические задачи