

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

Лекция 3. Алгебраические типы данных  
и структуры данных

---

В. Н. Брагилевский

28 февраля 2019 г.

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

## Сканирование списка

---

## Сканирование списка

```
foldl :: Foldable t =>
```

```
    (b -> a -> b) -> b -> t a -> b
```

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

```
foldr :: Foldable t =>
```

```
    (a -> b -> b) -> b -> t a -> b
```

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

```
ghci> scanl (+) 0 [3,5,2,1]
```

```
[0,3,8,10,11]
```

```
ghci> scanr (+) 0 [3,5,2,1]
```

```
[11,8,3,1,0]
```

```
ghci> scanl (flip (:)) [] [3,2,1]
```

```
[[],[3],[2,3],[1,2,3]]
```

- Сканирование – это свёртка с сохранением промежуточных значений.
- Имеются версии `scanl1` и `scanr1` без начального значения:

```
scanl1 :: (a -> a -> a) -> [a] -> [a]
```

```
scanr1 :: (a -> a -> a) -> [a] -> [a]
```

## Пример

### Задача

Определить наибольшее количество последовательных натуральных чисел, сумма квадратных корней которых не превосходит 1000.

## Пример

### Задача

Определить наибольшее количество последовательных натуральных чисел, сумма квадратных корней которых не превосходит 1000.

```
answer = length $ takeWhile (<= 1000)
           $ scanl1 (+) $ map sqrt [1..]
```

```
ghci> answer
```

```
130
```

```
ghci> sum (map sqrt [1..130])
```

```
993.6486803921487
```

```
ghci> sum (map sqrt [1..131])
```

```
1005.0942035344083
```

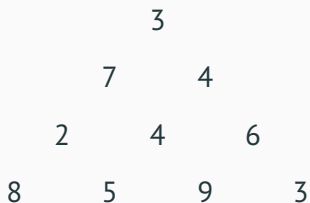
## Сканирование списка

---

Пример: путь в треугольнике

## Постановка задачи

Имеется числовой треугольник:

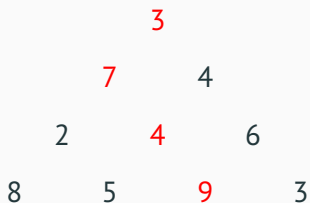


Необходимо вычислить максимальную сумму для пути от вершины к основанию. Допустим спуск либо влево, либо вправо.



## Постановка задачи

Имеется числовой треугольник:



Необходимо вычислить максимальную сумму для пути от вершины к основанию. Допустим спуск либо влево, либо вправо.



## Идея решения

10      7  
2      4      6  
8      5      9      3



## Идея решения

		10		7	
	2		4		6
8		5		9	3
	12		14		13
8		5		9	3



## Идея решения

	10		7		
	2		4	6	
8		5		9	3
	12		14		13
8		5		9	3
	20	19	23	16	

- Каждый уровень – список чисел.
- Треугольник в целом – список уровней.
- Сворачиваем все уровни в один уровень с максимальными путями.
- Находим максимум.



## Переход от одного уровня к другому

	12	14	13	
8	5	9	3	
	12		14	13
8+12	5+max{12,14}	9+max{14,13}	3+13	

## Переход от одного уровня к другому

	12	14	13	
8	5	9	3	
	12		14	13
$8+\max\{0,12\}$		$5+\max\{12,14\}$	$9+\max\{14,13\}$	$3+\max\{13,0\}$

## Переход от одного уровня к другому

	12	14	13	
8		5	9	3

		12		14		13	
	$8+\max\{0,12\}$		$5+\max\{12,14\}$		$9+\max\{14,13\}$		$3+\max\{13,0\}$

	0	12	14	13
	12	14	13	0

---

max

	12	14	14	13
--	----	----	----	----

## Переход от одного уровня к другому

```
downstep :: [Int] -> [Int] -> [Int]
downstep upper lower =
  zipWith (+)
    lower
    (zipWith max (0:upper) (upper ++ [0]))
```

```
answer :: [[Int]] -> Int  
answer = maximum . foldl1 downstep
```

```
downstep :: [Int] -> [Int] -> [Int]
downstep upper lower =
    zipWith (+)
        lower
        (zipWith max (0:upper) (upper ++ [0]))
```

```
answer :: [[Int]] -> Int
answer = maximum . foldl1 downstep
```

```
ghci> answer [[3],[7,4],[2,4,6],[8,5,9,3]]
23
```

## **Синонимы типов**

---

```
evalSmth :: (Double, Double) -> (Double, Double)
                                     -> Double
evalSmth = ...
```



```
evalSmth :: (Double, Double) -> (Double, Double)  
          -> Double
```

```
evalSmth = ...
```

```
type Point = (Double, Double)
```

```
evalSmth :: Point -> Point -> Double
```

```
evalSmth = ...
```

```
evalSmth :: (Double, Double) -> (Double, Double)  
          -> Double
```

```
evalSmth = ...
```

```
type Point = (Double, Double)
```

```
evalSmth :: Point -> Point -> Double
```

```
evalSmth = ...
```

```
ghci> evalSmth (0,0) (3,4)  
5.0
```

```
ghci> :info String
type String = [Char]      -- Defined in `GHC.Base`
```

# Алгебраические типы данных

---

# Алгебраические типы данных

---

## Типы-перечисления

## Примеры типов-перечислений

- Дни недели, месяцы.
- Логические значения.
- Пол (мужской/женский).
- Масть и значение в картах.
- Жанр.
- ...

## Тип для пола (мужской—женский)

```
data Sex = Male | Female
```

```
pensionAge :: Sex -> Int
```

```
pensionAge Male = 60
```

```
pensionAge Female = 55
```

```
ghci> pensionAge Male
```

```
60
```

```
ghci> pensionAge Female
```

```
55
```

```
ghci> map pensionAge [Male, Male, Female, Male]
```

```
[60,60,55,60]
```

```
data Sex = Male | Female
```

- Sex – тип
- Male, Female – конструкторы значений



## Проблема: обработка значений

```
ghci> Male
```

```
No instance for (Show Sex)
```

```
arising from a use of 'print'
```

```
Possible fix: add an instance
```

```
declaration for (Show Sex)
```

```
In a stmt of an interactive GHCi command: print it
```

```
ghci> Female == Female
```

```
No instance for (Eq Sex)
```

```
arising from a use of '=='
```

```
Possible fix: add an instance declaration for (Eq Sex)
```

```
In the expression: Female == Female
```

```
In an equation for 'it': it = Female == Female
```

```
data Sex = Male | Female deriving (Show, Eq)
```

```
ghci> Male
```

```
Male
```

```
ghci> [Male, Male, Female, Male]
```

```
[Male, Male, Female, Male]
```

```
ghci> Female == Female
```

```
True
```

## Пример: игральные карты

```
data Suit = Spades | Clubs | Diamonds | Hearts
  deriving (Show, Eq, Ord)
```

```
data Value = Seven | Eight | Nine | Ten
           | Jack | Queen | King | Ace
  deriving (Show, Eq, Ord)
```

```
type Card = (Value, Suit)
type Pack = [Card]
```

```
mpack :: Pack
```

```
mpack = [(Eight, Clubs), (Ten, Spades),
         (Ace, Diamonds)]
```

## Сравнение значений

```
ghci> Nine < Ace
```

```
True
```

```
ghci> (Nine, Spades) < (Ace, Clubs)
```

```
True
```

```
ghci> (Queen, Spades) < (Queen, Hearts)
```

```
True
```

## Примеры функций

```
isRed :: Suit -> Bool
isRed Diamonds = True
isRed Hearts   = True
isRed _        = False
```

```
isPicture :: Value -> Bool
isPicture v
  | v >= Jack = True
  | otherwise = False
```

## Пример использования

```
ghci> mpack
[(Eight, Clubs), (Ten, Spades), (Ace, Diamonds)]
ghci> all (isPicture.fst) mpack
False
ghci> any (isRed.snd) mpack
True
```

## Стандартный тип Ordering

```
data Ordering = LT | EQ | GT
```

```
compare :: Ord a => a -> a -> Ordering
```

Для сравнения: в языке C используются значения -1, 0, 1.

```
ghci> compare 5 8
```

```
LT
```

```
ghci> compare "abc" "aba"
```

```
GT
```

```
ghci> compare (Queen, Spades) (Queen, Spades)
```

```
EQ
```

```
ghci> compare (1,2) (5, 1)
```

```
LT
```

## Функция Data.Ord.comparing

Тип

```
comparing :: Ord a =>  
           (b -> a) -> b -> b -> Ordering
```



## Функция Data.Ord.comparing

Тип

```
comparing :: Ord a =>  
           (b -> a) -> b -> b -> Ordering
```

```
ghci> compare (1,2) (5, 1)
```

```
LT
```

```
ghci> import Data.Ord
```

```
ghci> comparing snd (1,2) (5, 1)
```

```
GT
```

## Функции sortBy и maximumBy (Data.List)

```
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
maximumBy :: Foldable t =>
             (a -> a -> Ordering) -> t a -> a
```

## Функции `sortBy` и `maximumBy` (`Data.List`)

```
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
maximumBy :: Foldable t =>
             (a -> a -> Ordering) -> t a -> a
```

```
ghci> xs = [(1, 'x'), (2, 'c'), (3, 'a')]
```

```
ghci> sort xs
```

```
[(1, 'x'), (2, 'c'), (3, 'a')]
```

```
ghci> :m +Data.List Data.Ord
```

```
ghci> sortBy (comparing snd) xs
```

```
[(3, 'a'), (2, 'c'), (1, 'x')]
```

```
ghci> maximumBy (comparing snd) xs
```

```
(1, 'x')
```

# Алгебраические типы данных

---

Типы-контейнеры

## Тип с персональной информацией

```
type Name = String
type Age = Int
data Person = Person Name Age
    deriving (Show)
```

```
name :: Person -> String
name (Person nm _) = nm
```

```
ghci> p1 = Person "Фредди Крюгер" 43
ghci> p1
Person "Фредди Крюгер" 43
ghci> name p1
"Фредди Крюгер"
```

```
data Person = Person {  
    firstName :: String  
    , lastName :: String  
    , age :: Int  
    , height :: Float  
    , phoneNumber :: String  
} deriving (Show)
```

```
freddy :: Person
```

```
freddy = Person {firstName="Фредди",  
    lastName="Крюгер", age=43, height=190,  
    phoneNumber="22223232"}
```

```
ghci> :t height
height :: Person -> Float
ghci> :t firstName
firstName :: Person -> String
```

```
ghci> :t height
height :: Person -> Float
ghci> :t firstName
firstName :: Person -> String

ghci> phoneNumber freddy
"22223232"
ghci> firstName freddy
"Фредди"
ghci> age freddy
43
ghci> freddy' = freddy {age=44}
ghci> age freddy'
44
```



## Сопоставление с образцом vs функции-аксессоры

```
process :: Person -> ...  
process (Person n a) = ... n ... a ...  
  
process :: Person -> ...  
process p = ... name p ... age p ...
```

## Сопоставление с образцом vs функции-аксессоры

```
process :: Person -> ...  
process (Person n a) = ... n ... a ...
```

```
process :: Person -> ...  
process p = ... name p ... age p ...
```

- А что если структура значения типа поменяется, например, добавится новое поле?

## Пример: индекс массы тела

```
data Person = Person {name :: String, age :: Int,  
                      weight :: Double, height :: Double}
```

```
bmi1 :: Person -> Double
```

```
bmi1 p = weight p / (height p)^2
```

```
bmi2 :: Person -> Double
```

```
bmi2 (Person _ _ w h) = w / h ^ 2
```

```
bmi3 :: Person -> Double
```

```
bmi3 Person {weight=w, height=h} = w / h ^ 2
```

## Пример: список покемонов

### Постановка задачи

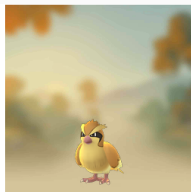
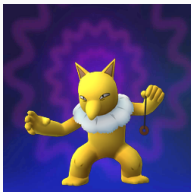
Дана строка следующего вида с информацией о покемонах и их СР:

Нурпо 152

Eevee 300

Pidgey 10

Разработать АТД для хранения информации о покемоне и преобразовать заданную строку в список значений этого типа.



```
data Pokemon = Pokemon String Int
  deriving Show
```

```
data Pokemon = Pokemon String Int
    deriving Show
```

```
parsePokemon :: String -> Pokemon
```

```
parsePokemon s = case words s of
    [ty, cp] -> Pokemon ty (read cp)
    _ -> error "неверный формат"
```

```
data Pokemon = Pokemon String Int
    deriving Show
```

```
parsePokemon :: String -> Pokemon
```

```
parsePokemon s = case words s of
    [ty, cp] -> Pokemon ty (read cp)
    _ -> error "неверный формат"
```

## Функция read

```
ghci> :t read
```

```
read :: Read a => String -> a
```

```
data Pokemon = Pokemon String Int
    deriving Show
```

```
parsePokemon :: String -> Pokemon
```

```
parsePokemon s = case words s of
    [ty, cp] -> Pokemon ty (read cp)
    _ -> error "неверный формат"
```

## Функция read

```
ghci> :t read
```

```
read :: Read a => String -> a
```

```
parseInfo :: String -> [Pokemon]
```

```
parseInfo = map parsePokemon . lines
```



```
ghci> parseInfo "Hypno 152\nEevee 300\nPidgey 10"  
[Pokemon "Hypno" 152,Pokemon "Eevee" 300,  
      Pokemon "Pidgey" 10]
```

## Пример: поиск покемона с максимальным CP

```
import Data.List
import Data.Ord

-- ...

maxCP :: [Pokemon] -> Pokemon
maxCP = maximumBy
      (comparing $ \ (Pokemon _ cp) -> cp)
```

# Алгебраические типы данных

---

Общий случай

## Пример: геометрическая фигура

```
data Point = Point Double Double
           deriving (Show)
```

```
type Radius = Double
```

```
data Shape = Circle Point Radius
           | Rectangle Point Point
           deriving (Show)
```

## Пример: геометрическая фигура

```
area :: Shape -> Double
area (Circle _ r) = pi * r ^ 2
area (Rectangle (Point x1 y1) (Point x2 y2))
      = (abs $ x2 - x1) * (abs $ y2 - y1)

ghci> area (Rectangle (Point 0 0) (Point 100 100))
10000.0
ghci> area (Circle (Point 0 0) 24)
1809.5574
```

Почему они алгебраические???

## Почему они алгебраические???

- Типы-перечисления – типы-суммы ( $x \in T_1 + T_2$ ).
- Типы-контейнеры – типы-произведения ( $x \in T_1 \times T_2$ ).
- Общий случай – сумма произведений.

# Алгебраические типы данных

---

## Параметризованные типы



### Функция Data.List.find

```
find :: (a -> Bool) -> [a] -> Maybe a
```

### Функция Data.List.find

```
find :: (a -> Bool) -> [a] -> Maybe a
```

Возможные результаты: либо элемент найден, либо нет.

### Функция Data.List.find

```
find :: (a -> Bool) -> [a] -> Maybe a
```

Возможные результаты: либо элемент найден, либо нет.

```
ghci> find (<0) [5, 10, -3, 3, -4]
```

```
Just (-3)
```

```
ghci> find (<0) [5, 10, 3]
```

```
Nothing
```

### Функция Data.List.find

```
find :: (a -> Bool) -> [a] -> Maybe a
```

Возможные результаты: либо элемент найден, либо нет.

```
ghci> find (<0) [5, 10, -3, 3, -4]
```

```
Just (-3)
```

```
ghci> find (<0) [5, 10, 3]
```

```
Nothing
```

```
data Maybe a = Nothing | Just a
```

## Пример: реализация функции elem через find

```
elem' a xs = case find (==a) xs of  
             Just _ -> True  
             Nothing -> False
```

## Значения типа Maybe a

```
ghci> :t Nothing
Nothing :: Maybe a
ghci> :t Just 3
Just 3 :: Num a => Maybe a
ghci> :t Just 'a'
Just 'a' :: Maybe Char
```

```
data Maybe a = Nothing | Just a
```

- Maybe Integer – тип.
- Maybe – конструктор типа (функция, аргументом и значением которой являются типы).
- Just, Nothing – конструкторы значений.

## Аналогия между Maybe a и [a]

- Nothing – пустой список [ ].
- Just 5 – одноэлементный список [5].



## Некоторые функции из модуля Data.Maybe

`maybe :: b -> (a -> b) -> Maybe a -> b`

`isJust :: Maybe a -> Bool`

`isNothing :: Maybe a -> Bool`

`fromJust :: Maybe a -> a`

`fromMaybe :: a -> Maybe a -> a`

`listToMaybe :: [a] -> Maybe a`

`maybeToList :: Maybe a -> [a]`

`catMaybes :: [Maybe a] -> [a]`

`mapMaybe :: (a -> Maybe b) -> [a] -> [b]`

## Некоторые функции из модуля Data.Maybe

`maybe :: b -> (a -> b) -> Maybe a -> b`

`isJust :: Maybe a -> Bool`

`isNothing :: Maybe a -> Bool`

`fromJust :: Maybe a -> a`

`fromMaybe :: a -> Maybe a -> a`

`listToMaybe :: [a] -> Maybe a`

`maybeToList :: Maybe a -> [a]`

`catMaybes :: [Maybe a] -> [a]`

`mapMaybe :: (a -> Maybe b) -> [a] -> [b]`

- По типу функции можно понять о ней всё или почти всё!

## Применение Maybe: функция Data.List.unfoldr

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

## Применение Maybe: функция Data.List.unfoldr

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

```
arith a d n = unfoldr next (a, 0)
```

```
  where
```

```
    next (a, k)
```

```
      | k < n = Just (a, (a+d, k+1))
```

```
      | otherwise = Nothing
```

## Применение Maybe: функция Data.List.unfoldr

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

```
arith a d n = unfoldr next (a, 0)
```

```
  where
```

```
    next (a, k)
```

```
      | k < n = Just (a, (a+d, k+1))
```

```
      | otherwise = Nothing
```

```
fibs = 0:1:unfoldr next (0,1)
```

```
  where
```

```
    next (f1, f2) = let f3 = f1 + f2
```

```
                    in Just (f3, (f2, f3))
```

## Тип Either a b

```
data Either a b = Left a | Right b
```

```
ghci> Right 20
```

```
Right 20
```

```
ghci> Left "xxx"
```

```
Left "xxx"
```

```
ghci> :t Right 'a'
```

```
Right 'a' :: Either a Char
```

```
ghci> :t Left True
```

```
Left True :: Either Bool b
```

Обычное применение: Right (результат) или Left (Ошибка).

## Пример: параметризованные векторы

```
data Vector a = Vector a a a deriving (Show)
```

```
vplus :: Num a => Vector a -> Vector a -> Vector a  
(Vector i j k) `vplus` (Vector l m n)  
    = Vector (i+l) (j+m) (k+n)
```

```
scalarProd :: Num a => Vector a -> Vector a -> a  
(Vector i j k) `scalarProd` (Vector l m n)  
    = i*l + j*m + k*n
```

```
vmult :: Num a => a -> Vector a -> Vector a  
m `vmult` (Vector i j k) = Vector (m*i) (m*j) (m*k)
```

## Пример: параметризованные векторы

```
ghci> Vector 3 5 8 `vplus` Vector 9 2 8  
Vector 12 7 16
```

```
ghci> Vector 3 5 8 `vplus` Vector 9 2 8  
                `vplus` Vector 0 2 3  
Vector 12 9 19
```

```
ghci> 10 `vmult` Vector 3 9 7  
Vector 30 90 70
```

```
ghci> (Vector 4 9 5 `scalarProd` Vector 9 2 4)  
      `vmult` Vector 2 9 3  
Vector 148 666 222
```

```
ghci> Vector 4 9 5 `scalarProd` Vector 9.0 2.0 4.0  
74.0
```



# Алгебраические типы данных

---

## Рекурсивные типы

- Список — это либо пустой список, либо голова и хвост (тоже список).
- Бинарное дерево — это либо пустое дерево, либо узел с двумя поддеревьями.
- Арифметическое выражение — это либо число, либо сумма выражений, либо произведение выражений.
- Рекурсивные типы обычно параметризованы.

```
data List a = Nil | Cons a (List a)
```

```
Cons 1 (Cons 2 (Cons 3 Nil)) :: Num a => List a
```

```
data List a = Nil | Cons a (List a)
```

```
Cons 1 (Cons 2 (Cons 3 Nil)) :: Num a => List a
```

## Извлечение первого элемента

```
head' :: List a -> a
```

```
head' Nil = error "no such element"
```

```
head' (Cons a _) = a
```

# Бинарное дерево

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a)
    deriving (Show)
```

```
t :: Tree Char
```

```
t = Node 'g' (Node 'a' EmptyTree EmptyTree)
            EmptyTree
```

## Вставка в бинарное дерево поиска

### Создание одноэлементного дерева

```
singleton :: a -> Tree a
```

```
singleton x = Node x EmptyTree EmptyTree
```

## Вставка в бинарное дерево поиска

### Создание одноэлементного дерева

```
singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree
```

### Вставка элемента

```
treeInsert :: (Ord a, Eq a) =>
             a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a left right) =
  case compare x a of
    EQ -> Node a left right
    LT -> Node a (treeInsert x left) right
    GT -> Node a left (treeInsert x right)
```

```
treeElem :: (Ord a, Eq a) => a -> Tree a -> Bool
treeElem x EmptyTree = False
treeElem x (Node a left right) =
  case compare x a of
    EQ -> True
    LT -> treeElem x left
    GT -> treeElem x right
```



## Формирование БДП

```
nTree :: Tree Integer
```

```
nTree = foldr treeInsert EmptyTree [8,6,4,1,7,3,5]
```

```
ghci> nTree
```

```
Node 5
```

```
  (Node 3
```

```
    (Node 1 EmptyTree EmptyTree)
```

```
    (Node 4 EmptyTree EmptyTree)
```

```
  )
```

```
  (Node 7
```

```
    (Node 6 EmptyTree EmptyTree)
```

```
    (Node 8 EmptyTree EmptyTree)
```

```
  )
```

## Формирование БДП

```
nTree :: Tree Integer
```

```
nTree = foldr treeInsert EmptyTree [8,6,4,1,7,3,5]
```

```
ghci> nTree
```

```
Node 5
```

```
  (Node 3
```

```
    (Node 1 EmptyTree EmptyTree)
```

```
    (Node 4 EmptyTree EmptyTree)
```

```
  )
```

```
  (Node 7
```

```
    (Node 6 EmptyTree EmptyTree)
```

```
    (Node 8 EmptyTree EmptyTree)
```

```
  )
```

- Тип `treeInsert` случайно подошёл для `foldr`.

# Представление арифметических выражений

Тип

```
data IntExpr
  = I Int           -- константа
  | Add IntExpr IntExpr -- сумма
  | Mul IntExpr IntExpr -- произведение
```

## Представление арифметических выражений

Тип

```
data IntExpr
  = I Int           -- константа
  | Add IntExpr IntExpr -- сумма
  | Mul IntExpr IntExpr -- произведение
```

```
(I 5 'Add' I 1) 'Mul' I 7 :: IntExpr
```

# Представление арифметических выражений

## Тип

```
data IntExpr
  = I Int           -- константа
  | Add IntExpr IntExpr -- сумма
  | Mul IntExpr IntExpr -- произведение
```

```
(I 5 'Add' I 1) 'Mul' I 7 :: IntExpr
```

## Вычисление

```
eval :: IntExpr -> Int
eval (I n) = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Mul e1 e2) = eval e1 * eval e2
```

# Линейные структуры данных

---

# Линейные структуры данных

---

Список

# Список (Data.List)

## Определение

Список — это набор элементов (конечный или бесконечный) с быстрым доступом к началу ( $O(1)$ ) и линейной сложностью большинства операций.

## Основные приёмы использования

- Функции высшего порядка.
- Генераторы списков.



# Список (Data.List)

## Определение

Список — это набор элементов (конечный или бесконечный) с быстрым доступом к началу ( $O(1)$ ) и линейной сложностью большинства операций.

## Основные приёмы использования

- Функции высшего порядка.
- Генераторы списков.

```
[ x2 + y2 | x <- xs, y <- ys,  
          let x2 = x^2, let y2 = y^2,  
          x2 `mod` y2 == 0 ]
```

# Линейные структуры данных

---

Последовательность

## Последовательность (Data.Sequence)

### Определение

Последовательность — это конечный проиндексированный набор элементов с быстрым доступом к началу и концу ( $O(1)$ ) и логарифмической амортизированной сложностью многих операций.

# Последовательность (Data.Sequence)

## Определение

Последовательность — это конечный проиндексированный набор элементов с быстрым доступом к началу и концу ( $O(1)$ ) и логарифмической амортизированной сложностью многих операций.

## Типы операций

- Конструирование.
- Обращение к содержимому.
- Преобразование.

Последовательности обобщают очереди и деки (двусторонние очереди). В основе реализации «2-3 finger trees», аннотированные размерами.

## Конструирование и простые запросы

```
empty :: Seq a                -- 0(1)
singleton :: a -> Seq a      -- 0(1)
replicate :: Int -> a -> Seq a -- 0(log n)

(<|) :: a -> Seq a -> Seq a   -- 0(1)
(|>) :: Seq a -> a -> Seq a   -- 0(1)

(><) :: Seq a -> Seq a -> Seq a
      -- 0(log(min(n1, n2)))

fromList :: [a] -> Seq a      -- 0(n)

null :: Seq a -> Bool        -- 0(1)
length :: Seq a -> Int       -- 0(1)
```

## Пример конструирования и обращение к концам

```
ghci> let s = (4 <| 5 <| singleton 6)
           >< (empty |> 7 |> 8)
```

```
ghci> s
```

```
fromList [4,5,6,7,8]
```

```
ghci> let (x :< xs) = viewl s
```

```
ghci> x
```

```
4
```

```
ghci> xs
```

```
fromList [5,6,7,8]
```

```
ghci> let (ys :> y) = viewr s
```

```
ghci> y
```

```
8
```

```
ghci> ys
```

```
fromList [4,5,6,7]
```

## Обращение к концам: типы

```
data ViewL a = EmptyL | a :< Seq a
```

```
data ViewR a = EmptyR | Seq a :> a
```

```
viewl :: Seq a -> ViewL a      -- O(1)
```

```
viewr :: Seq a -> ViewR a      -- O(1)
```

## Пример

```
endsWith :: Eq a => Seq a -> a -> Bool
endsWith s x = let (_ :> x') = viewr s
                in x == x'
```

```
sameEnds :: Eq a => Seq a -> Bool
sameEnds s = l == r
  where
    (l :< _) = viewl s
    (_ :> r) = viewr s
```

```
ghci> endsWith (1 <| 2 <| 3 <| empty) 3
True
ghci> sameEnds (1 <| 2 <| 3 <| empty)
False
```



## Разные операции с последовательностями

### Операции с индексами (логарифмическая сложность)

```
index :: Seq a -> Int -> a
adjust :: (a -> a) -> Int -> Seq a -> Seq a
update :: Int -> a -> Seq a -> Seq a
take :: Int -> Seq a -> Seq a
drop :: Int -> Seq a -> Seq a
splitAt :: Int -> Seq a -> (Seq a, Seq a)
```

### Обращение последовательности ( $O(n)$ )

```
reverse :: Seq a -> Seq a
```

Имеются также операции, аналогичные операциям со списками.

# Множества и отображения

---

# Множества и отображения

---

## Множества

## Определение

Множество — тип данных, поддерживающий эффективные операции вставки, удаления и проверки принадлежности для данных некоторого типа.

- Модуль `Data.Set`.
- Реализация — сбалансированные бинарные деревья поиска.
- Ограничение для элементов — класс типов `Ord`.

# Импорт модуля `Data.Set` и создание множества

## Импорт модуля

```
import qualified Data.Set as Set
```

## Создание множества

```
empty :: Set a  
singleton :: a -> Set a  
fromList :: Ord a => [a] -> Set a  
fromAscList :: Eq a => [a] -> Set a  
fromDistinctAscList :: [a] -> Set a
```

- Предусловия не проверяются!

## Основные операции с множествами

```
null :: Set a -> Bool
size :: Set a -> Int
member :: Ord a => a -> Set a -> Bool
notMember :: Ord a => a -> Set a -> Bool
isSubsetOf :: Ord a => Set a -> Set a -> Bool
isProperSubsetOf :: Ord a => Set a -> Set a -> Bool
insert :: Ord a => a -> Set a -> Set a
delete :: Ord a => a -> Set a -> Set a
union :: Ord a => Set a -> Set a -> Set a
unions :: Ord a => [Set a] -> Set a
difference :: Ord a => Set a -> Set a -> Set a
intersection :: Ord a => Set a -> Set a -> Set a
```

## Обработка элементов множества

```
filter :: (a -> Bool) -> Set a -> Set a
partition :: (a -> Bool) -> Set a
           -> (Set a, Set a)
split :: Ord a => a -> Set a -> (Set a, Set a)
splitMember :: Ord a => a -> Set a
            -> (Set a, Bool, Set a)
map :: Ord b => (a -> b) -> Set a -> Set b
```

В модуле `Data.Set` определён экземпляр класса типов `Foldable`, то есть можно использовать любые его операции (`fold`, `foldr`,...).

### Множество как очередь с приоритетом

```
findMin :: Set a -> a
findMax :: Set a -> a
deleteMin :: Set a -> Set a
deleteMax :: Set a -> Set a
deleteFindMin :: Set a -> (a, Set a)
deleteFindMax :: Set a -> (a, Set a)
```

### Преобразование множества в список

```
elems :: Set a -> [a]
toAscList :: Set a -> [a]
toDescList :: Set a -> [a]
```



## Пример: проверка принадлежности множеству

### Задача

Определить, сколько слов из данного набора являются женскими именами.

## Пример: проверка принадлежности множеству

### Задача

Определить, сколько слов из данного набора являются женскими именами.

```
import qualified Data.Set as Set

names = ["анна", "юлия", "ольга", "мария",
         "дарья", "александра", "ирина", "наталья",
         "оксана", "галина", "виктория", "любовь",
         "вера", "алиса", "татьяна"]
names' = Set.fromList female_names

some_words = take 10000000 $
  cycle ["анна", "лето", "двор", "вера"]
```

```
answer = length $  
  filter ('elem' female_names) some_words  
answer' = length $  
  filter ('Set.member' female_names') some_words
```

```
answer = length $  
    filter ('elem' female_names) some_words  
answer' = length $  
    filter ('Set.member' female_names') some_words
```

```
ghci> length some_words  
10000000  
(0.28 secs, 561475440 bytes)
```

```
ghci> answer  
5000000  
(3.86 secs, 280926888 bytes)
```

```
ghci> answer'  
5000000  
(2.12 secs, 520981296 bytes)
```

```
import qualified Data.IntSet as IntSet
```

```
primes = IntSet.fromAscList [2,3,5,7,11,13]
```

```
ghci> 5 `IntSet.member` primes  
True
```

```
ghci> 4 `IntSet.member` primes  
False
```

# Множества и отображения

---

**Отображения: создание и основные операции**

## Определение

Отображение (словарь) – тип данных, позволяющий хранить пары вида (ключ, значение) и поддерживающий операции добавления пары, а также поиска и удаления пары по ключу.

## Модуль Data.Мар – отображения

```
import qualified Data.Map as Map

type Name = String
type PhoneNumber = String

phoneBook :: Map.Map Name PhoneNumber
phoneBook = Map.fromList
  [ ("оля", "555-29-38")
  , ("женя", "452-29-28")
  , ("катя", "493-29-28")
  , ("маша", "205-29-28")
  , ("надя", "939-82-82")
  , ("юля", "853-24-92")
  ]
```



```
lookup :: Ord k => k -> Map k a -> Maybe a
```

```
ghci> Map.lookup "оля" phoneBook
```

```
Just "555-29-38"
```

```
ghci> Map.lookup "надя" phoneBook
```

```
Just "939-82-82"
```

```
ghci> Map.lookup "таня" phoneBook
```

```
Nothing
```

## Добавление записи и определение размера

```
insert :: Ord k => k -> a -> Map k a -> Map k a  
size  :: Map k a -> Int
```

```
ghci> Map.lookup "таня" phoneBook  
Nothing
```

```
ghci> let newBook = Map.insert  
           "таня" "341-90-21" phoneBook
```

```
ghci> Map.lookup "таня" newBook  
Just "341-90-21"
```

```
ghci> Map.size phoneBook  
6
```

```
ghci> Map.size newBook  
7
```

## Некоторые функции из модуля Data.Map

```
member :: Ord k => k -> Map k a -> Bool
```

```
map :: (a -> b) -> Map k a -> Map k b
```

```
delete :: Ord k => k -> Map k a -> Map k a
```

```
update :: Ord k => (a -> Maybe a) -> k -> Map k a  
                                             -> Map k a
```

```
elems :: Map k a -> [a]
```

```
keys :: Map k a -> [k]
```

## Некоторые функции из модуля Data.Map (2)

`insertWith` :: **Ord** k =>

(a -> a -> a) -> k -> a -> **Map** k a -> **Map** k a

`insertWithKey` :: **Ord** k =>

(k -> a -> a -> a) -> k -> a -> **Map** k a  
-> **Map** k a

`insertLookupWithKey` :: **Ord** k =>

(k -> a -> a -> a) -> k -> a -> **Map** k a  
-> (**Maybe** a, **Map** k a)

# Множества и отображения

---

Пример: шкафчики для хранения

```
import qualified Data.Map as Map
```

```
data LockerState = Taken | Free  
  deriving (Show, Eq)
```

```
type Code = String
```

```
type LockerMap = Map.Map Int (LockerState, Code)
```

## Поиск шкафчика

```
lockerLookup :: Int -> LockerMap
              -> Either String Code
lockerLookup lockerNumber lockers =
  case Map.lookup lockerNumber lockers of
    Nothing -> Left $ locker ++ " отсутствует"
    Just (Taken, _ ) -> Left $ locker
                      ++ " занят"
    Just (Free, code) -> Right code
  where
    locker = "Шкафчик #" ++ show lockerNumber
```

```
lockers :: LockerMap
lockers = Map.fromList
  [(100,(Taken,"ZD39I"))
  ,(101,(Free,"JAH3I"))
  ,(103,(Free,"IQSA9"))
  ,(105,(Free,"QOTSA"))
  ,(109,(Taken,"893JJ"))
  ,(110,(Taken,"99292"))
  ]
```

```
ghci> lockerLookup 101 lockers
Right "JAH3I"
ghci> lockerLookup 100 lockers
Left "Шкафчик #100 занят"
ghci> lockerLookup 102 lockers
Left "Шкафчик #102 отсутствует"
ghci> lockerLookup 110 lockers
Left "Шкафчик #110 занят"
ghci> lockerLookup 105 lockers
Right "QOTSA"
```