

Кортежи и списъци

Трифон Трифонов

Функционално програмиране, 2017/18 г.

14 декември 2017 г. – 4 януари 2018 г.

Кортежи (tuples)

Кортежите са наредени n -торки от данни от произволен тип.

- Примери: (1, 2), (3.5, 'A', False),
(("square", (^2)), 1.0)

Кортежи (tuples)

Кортежите са наредени n -торки от данни от произволен тип.

- Примери: `(1, 2), (3.5, 'A', False),
("square", (^2)), 1.0)`
- Тип кортеж от n елемента: (t_1, t_2, \dots, t_n)

Кортежи (tuples)

Кортежите са наредени n -торки от данни от произволен тип.

- Примери: `(1, 2), (3.5, 'A', False),
("square", (^2)), 1.0`
- Тип кортеж от n елемента: (t_1, t_2, \dots, t_n)
- Стойности: наредени n -торки от вида (x_1, x_2, \dots, x_n) , където x_i е от тип t_i

Кортежи (tuples)

Кортежите са наредени n -торки от данни от произволен тип.

- Примери: `(1, 2), (3.5, 'A', False),
("square", (^2)), 1.0`
- Тип кортеж от n елемента: (t_1, t_2, \dots, t_n)
- Стойности: наредени n -торки от вида (x_1, x_2, \dots, x_n) , където x_i е от тип t_i
- Позволяват “пакетиране“ на няколко стойности в една

Кортежи (tuples)

Кортежите са наредени n -торки от данни от произволен тип.

- Примери: `(1, 2), (3.5, 'A', False),
("square", (^2)), 1.0`
- Тип кортеж от n елемента: (t_1, t_2, \dots, t_n)
- Стойности: наредени n -торки от вида (x_1, x_2, \dots, x_n) , където x_i е от тип t_i
- Позволяват “пакетиране” на няколко стойности в една
- Операции за наредени двойки:

Кортежи (tuples)

Кортежите са наредени n -торки от данни от произволен тип.

- Примери: `(1, 2), (3.5, 'A', False),
("square", (^2)), 1.0`
- Тип кортеж от n елемента: (t_1, t_2, \dots, t_n)
- Стойности: наредени n -торки от вида (x_1, x_2, \dots, x_n) , където x_i е от тип t_i
- Позволяват “пакетиране“ на няколко стойности в една
- Операции за наредени двойки:
 - `(,) :: a -> b -> (a,b)` — конструиране на наредена двойка

Кортежи (tuples)

Кортежите са наредени n -торки от данни от произволен тип.

- Примери: `(1, 2), (3.5, 'A', False),
("square", (^2)), 1.0`
- Тип кортеж от n елемента: (t_1, t_2, \dots, t_n)
- Стойности: наредени n -торки от вида (x_1, x_2, \dots, x_n) , където x_i е от тип t_i
- Позволяват “пакетиране“ на няколко стойности в една
- Операции за наредени двойки:
 - `(,)` :: $a \rightarrow b \rightarrow (a,b)$ — конструиране на наредена двойка
 - `fst` :: $(a,b) \rightarrow a$ — първа компонента на наредена двойка

Кортежи (tuples)

Кортежите са наредени n -торки от данни от произволен тип.

- Примери: `(1, 2), (3.5, 'A', False),
("square", (^2)), 1.0`
- Тип кортеж от n елемента: (t_1, t_2, \dots, t_n)
- Стойности: наредени n -торки от вида (x_1, x_2, \dots, x_n) , където x_i е от тип t_i
- Позволяват “пакетиране“ на няколко стойности в една
- Операции за наредени двойки:
 - `(,)` :: $a \rightarrow b \rightarrow (a,b)$ — конструиране на наредена двойка
 - `fst` :: $(a,b) \rightarrow a$ — първа компонента на наредена двойка
 - `snd` :: $(a,b) \rightarrow b$ — втора компонента на наредена двойка

Потребителски типове

- Типът (`String`, `Int`) може да означава:

Потребителски типове

- Типът (`String`, `Int`) може да означава:
 - име и ЕГН на човек

Потребителски типове

- Типът (`String`, `Int`) може да означава:
 - име и ЕГН на човек
 - продукт с описание и количество

Потребителски типове

- Типът (`String`, `Int`) може да означава:
 - име и ЕГН на човек
 - продукт с описание и количество
 - сонет на Шекспир и поредният му номер

Потребителски типове

- Типът (`String`, `Int`) може да означава:
 - име и ЕГН на човек
 - продукт с описание и количество
 - сонет на Шекспир и поредният му номер
- Удобно е да именуваме типовете, за да означаваме смисъла им

Потребителски типове

- Типът (`String`, `Int`) може да означава:
 - име и ЕГН на човек
 - продукт с описание и количество
 - сонет на Шекспир и поредният му номер
- Удобно е да именуваме типовете, за да означаваме смисъла им
- `type <конструктор> = <тип>`

Потребителски типове

- Типът (`String`, `Int`) може да означава:
 - име и ЕГН на човек
 - продукт с описание и количество
 - сонет на Шекспир и поредният му номер
- Удобно е да именуваме типовете, за да означаваме смисъла им
- `type <конструктор> = <тип>`
 - конструкторите са идентификатори, започващи с главна буква

Потребителски типове

- Типът (`String`, `Int`) може да означава:
 - име и ЕГН на човек
 - продукт с описание и количество
 - сонет на Шекспир и поредният му номер
- Удобно е да именуваме типовете, за да означаваме смисъла им
- `type <конструктор> = <тип>`
 - конструкторите са идентификатори, започващи с главна буква
- Примери:

Потребителски типове

- Типът (`String`, `Int`) може да означава:
 - име и ЕГН на човек
 - продукт с описание и количество
 - сонет на Шекспир и поредният му номер
- Удобно е да именуваме типовете, за да означаваме смисъла им
- `type <конструктор> = <тип>`
 - конструкторите са идентификатори, започващи с главна буква
- Примери:
 - `type Student = (String, Int, Double)`

Потребителски типове

- Типът (`String`, `Int`) може да означава:
 - име и ЕГН на човек
 - продукт с описание и количество
 - сонет на Шекспир и поредният му номер
- Удобно е да именуваме типовете, за да означаваме смисъла им
- `type <конструктор> = <тип>`
 - конструкторите са идентификатори, започващи с главна буква
- Примери:
 - `type Student = (String, Int, Double)`
 - `type Point = (Double, Double)`

Потребителски типове

- Типът (`String`, `Int`) може да означава:
 - име и ЕГН на човек
 - продукт с описание и количество
 - сонет на Шекспир и поредният му номер
- Удобно е да именуваме типовете, за да означаваме смисъла им
- `type <конструктор> = <тип>`
 - конструкторите са идентификатори, започващи с главна буква
- Примери:
 - `type Student = (String, Int, Double)`
 - `type Point = (Double, Double)`
 - `type Triangle = (Point, Point, Point)`

Потребителски типове

- Типът (`String`, `Int`) може да означава:
 - име и ЕГН на човек
 - продукт с описание и количество
 - сонет на Шекспир и поредният му номер
- Удобно е да именуваме типовете, за да означаваме смисъла им
- `type <конструктор> = <тип>`
 - конструкторите са идентификатори, започващи с главна буква
- Примери:
 - `type Student = (String, Int, Double)`
 - `type Point = (Double, Double)`
 - `type Triangle = (Point, Point, Point)`
 - `type Translation = Point -> Point`

Потребителски типове

- Типът (`String`, `Int`) може да означава:
 - име и ЕГН на човек
 - продукт с описание и количество
 - сонет на Шекспир и поредният му номер
- Удобно е да именуваме типовете, за да означаваме смисъла им
- `type <конструктор> = <тип>`
 - конструкторите са идентификатори, започващи с главна буква
- Примери:
 - `type Student = (String, Int, Double)`
 - `type Point = (Double, Double)`
 - `type Triangle = (Point, Point, Point)`
 - `type Translation = Point -> Point`
 - `type Vector = Point`

Потребителски типове

- Типът (`String`, `Int`) може да означава:
 - име и ЕГН на човек
 - продукт с описание и количество
 - сонет на Шекспир и поредният му номер
- Удобно е да именуваме типовете, за да означаваме смисъла им
- `type <конструктор> = <тип>`
 - конструкторите са идентификатори, започващи с главна буква
- Примери:
 - `type Student = (String, Int, Double)`
 - `type Point = (Double, Double)`
 - `type Triangle = (Point, Point, Point)`
 - `type Translation = Point -> Point`
 - `type Vector = Point`
 - `addVectors :: Vector -> Vector -> Vector`
 - `addVectors v1 v2 = (fst v1 + fst v2, snd v1 + snd v2)`

Особености на кортежите

- **fst** (1,2,3) → ?

Особености на кортежите

- **fst** (1,2,3) → Грешка!

Особености на кортежите

- `fst` (1,2,3) → Грешка!
- `fst` и `snd` работят само над наредени двойки!

Особености на кортежите

- **fst** $(1, 2, 3) \rightarrow$ Грешка!
 - **fst** и **snd** работят само над наредени двойки!
- $((a, b), c) \neq (a, (b, c)) \neq (a, b, c)$

Особености на кортежите

- **fst** $(1, 2, 3) \rightarrow$ Грешка!
- **fst** и **snd** работят само над наредени двойки!
- $((a, b), c) \neq (a, (b, c)) \neq (a, b, c)$
- Няма специален тип кортеж от един елемент . . .

Особености на кортежите

- **fst** (1,2,3) → Грешка!
 - **fst** и **snd** работят само над наредени двойки!
- $((a,b),c) \neq (a,(b,c)) \neq (a,b,c)$
- Няма специален тип кортеж от един елемент . . .
- . . . но има тип “празен кортеж” () с единствен елемент ()

Особености на кортежите

- `fst` (1,2,3) → Грешка!
 - `fst` и `snd` работят само над наредени двойки!
- $((a,b),c) \neq (a,(b,c)) \neq (a,b,c)$
- Няма специален тип кортеж от един елемент . . .
- . . . но има тип “празен кортеж” () с единствен елемент ()
 - в други езици такъв тип се нарича `unit`

Особености на кортежите

- **fst** (1,2,3) → Грешка!
 - **fst** и **snd** работят само над наредени двойки!
- $((a,b),c) \neq (a,(b,c)) \neq (a,b,c)$
- Няма специален тип кортеж от един елемент . . .
- . . . но има тип “празен кортеж” () с единствен елемент ()
 - в други езици такъв тип се нарича **unit**
 - използва се за означаване на липса на информация

Образци на кортежи

Образец на кортеж е конструкция от вида (p_1, p_2, \dots, p_n) .

Образци на кортежи

Образец на кортеж е конструкция от вида (p_1, p_2, \dots, p_n) .

Пасва на всеки кортеж от точно n елемента (x_1, x_2, \dots, x_n) , за който образецът p_i пасва на елемента x_i .

Образци на кортежи

Образец на кортеж е конструкция от вида (p_1, p_2, \dots, p_n) .

Пасва на всеки кортеж от точно n елемента (x_1, x_2, \dots, x_n) , за който образецът p_i пасва на елемента x_i .

- addVectors $(x_1, y_1) (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$

Образци на кортежи

Образец на кортеж е конструкция от вида (p_1, p_2, \dots, p_n) .

Пасва на всеки кортеж от точно n елемента (x_1, x_2, \dots, x_n) , за който образецът p_i пасва на елемента x_i .

- `addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)`
- `fst (x,_) = x`
- `snd (_,y) = y`

Образци на кортежи

Образец на кортеж е конструкция от вида (p_1, p_2, \dots, p_n) .

Пасва на всеки кортеж от точно n елемента (x_1, x_2, \dots, x_n) , за който образецът p_i пасва на елемента x_i .

- `addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)`
- `fst (x,_) = x`
- `snd (_,y) = y`
- `getFN :: Student -> Int`
- `getFN (_, fn, _) = fn`

Образци на кортежи

Образец на кортеж е конструкция от вида (p_1, p_2, \dots, p_n) .

Пасва на всеки кортеж от точно n елемента (x_1, x_2, \dots, x_n) , за който образецът p_i пасва на елемента x_i .

- `addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)`
- `fst (x,_) = x`
- `snd (_,y) = y`
- `getFN :: Student -> Int`
- `getFN (_, fn, _) = fn`
- образците на кортежи могат да се използват за “разглобяване” на кортежи при дефиниция

Образци на кортежи

Образец на кортеж е конструкция от вида (p_1, p_2, \dots, p_n) .

Пасва на всеки кортеж от точно n елемента (x_1, x_2, \dots, x_n) , за който образецът p_i пасва на елемента x_i .

- `addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)`
- `fst (x,_) = x`
- `snd (_,y) = y`
- `getFN :: Student -> Int`
- `getFN (_, fn, _) = fn`
- образците на кортежи могат да се използват за “разглобяване” на кортежи при дефиниция
- $(x,y) = (3.5, 7.8)$

Образци на кортежи

Образец на кортеж е конструкция от вида (p_1, p_2, \dots, p_n) .

Пасва на всеки кортеж от точно n елемента (x_1, x_2, \dots, x_n) , за който образецът p_i пасва на елемента x_i .

- `addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)`
- `fst (x,_) = x`
- `snd (_,y) = y`
- `getFN :: Student -> Int`
- `getFN (_, fn, _) = fn`
- образците на кортежи могат да се използват за “разглобяване” на кортежи при дефиниция
- $(x,y) = (3.5, 7.8)$
- `let (_, fn, grade) = student in (fn, min (grade + 1) 6)`

Именувани образци

- намиране на студент с по-висока оценка

```
betterStudent (name1, fn1, grade1) (name2, fn2, grade2)
| grade1 > grade2 = (name1, fn1, grade1)
| otherwise          = (name2, fn2, grade2)
```

Именувани образци

- намиране на студент с по-висока оценка

```
betterStudent (name1, fn1, grade1) (name2, fn2, grade2)
| grade1 > grade2 = (name1, fn1, grade1)
| otherwise          = (name2, fn2, grade2)
```

- ами ако имахме 10 полета?

Именувани образци

- намиране на студент с по-висока оценка

```
betterStudent (name1, fn1, grade1) (name2, fn2, grade2)
| grade1 > grade2 = (name1, fn1, grade1)
| otherwise         = (name2, fn2, grade2)
```

- ами ако имахме 10 полета?
- удобно е да използваме **именувани образци**

Именувани образци

- намиране на студент с по-висока оценка

```
betterStudent (name1, fn1, grade1) (name2, fn2, grade2)
| grade1 > grade2 = (name1, fn1, grade1)
| otherwise         = (name2, fn2, grade2)
```

- ами ако имахме 10 полета?
- удобно е да използваме **именувани образци**
- <име>@<образец>

Именувани образци

- намиране на студент с по-висока оценка

```
betterStudent (name1, fn1, grade1) (name2, fn2, grade2)
| grade1 > grade2 = (name1, fn1, grade1)
| otherwise          = (name2, fn2, grade2)
```

- ами ако имахме 10 полета?
- удобно е да използваме **именувани образци**
- <име>@<образец>

```
betterStudent s1@(_, _, grade1) s2@(_, _, grade2)
| grade1 > grade2 = s1
| otherwise          = s2
```

Списъци

Дефиниция

- ① Празният списък [] е списък от тип [a]
- ② Ако h е елемент от тип a и t е списък от тип [a] то (h : t) е списък от тип [a]
 - h — глава на списъка
 - t — опашка на списъка

Списъци

Дефиниция

- ① Празният списък [] е списък от тип [a]
 - ② Ако h е елемент от тип a и t е списък от тип [a] то $(h : t)$ е списък от тип [a]
 - h — глава на списъка
 - t — опашка на списъка
-
- Списъкът е последователност с **произволна дължина** от елементи от **еднакъв тип**

Списъци

Дефиниция

- ① Празният списък [] е списък от тип [a]
 - ② Ако h е елемент от тип a и t е списък от тип [a] то $(h : t)$ е списък от тип [a]
 - h — глава на списъка
 - t — опашка на списъка
-
- списъкът е последователност с **произволна дължина** от елементи от **еднакъв тип**
 - $(::)$:: $a \rightarrow [a] \rightarrow [a]$ е **дясноасоциативна** бинарна операция

Списъци

Дефиниция

- ① Празният списък [] е списък от тип [a]
 - ② Ако h е елемент от тип a и t е списък от тип [a] то $(h : t)$ е списък от тип [a]
 - h — глава на списъка
 - t — опашка на списъка
-
- списъкът е последователност с **произволна дължина** от елементи от **еднакъв тип**
 - $(::)$:: $a \rightarrow [a] \rightarrow [a]$ е **дясноасоциативна** бинарна операция
 - $(1:(2:(3:(4:[])))) = 1:2:3:4:[] \neq (((1:2):3):4):[]$

Списъци

Дефиниция

- ① Празният списък [] е списък от тип [a]
 - ② Ако h е елемент от тип a и t е списък от тип [a] то (h : t) е списък от тип [a]
 - h — глава на списъка
 - t — опашка на списъка
-
- списъкът е последователност с **произволна дължина** от елементи от **еднакъв тип**
 - $(::)$:: a \rightarrow [a] \rightarrow [a] е **дясноасоциативна** бинарна операция
 - $(1:(2:(3:(4:[])))) = 1:2:3:4:[] \neq (((1:2):3):4):[]$
 - $[a_1, a_2, \dots, a_n]$ е по-удобен запис за $a_1:(a_2:\dots(a_n:[])\dots)$

Списъци

Дефиниция

- ① Празният списък [] е списък от тип [a]
 - ② Ако h е елемент от тип a и t е списък от тип [a] то (h : t) е списък от тип [a]
 - h — глава на списъка
 - t — опашка на списъка
-
- списъкът е последователност с **произволна дължина** от елементи от **еднакъв тип**
 - $(::)$:: a \rightarrow [a] \rightarrow [a] е **дясноасоциативна** бинарна операция
 - $(1:(2:(3:(4:[])))) = 1:2:3:4:[] \neq (((1:2):3):4):[]$
 - $[a_1, a_2, \dots, a_n]$ е по-удобен запис за $a_1:(a_2:\dots(a_n:[])\dots)$
 - $[1,2,3,4] = 1:[2,3,4] = 1:2:[3,4] = 1:2:3:[4] = 1:2:3:4:[]$

Примери

- [False] :: ?

Примери

- [False] :: [Bool]

Примери

- [False] :: [Bool]
- ["Иван", 4.5] :: ?

Примери

- [False] :: [Bool]
- ["Иван", 4.5] :: \perp

Примери

- [False] :: [Bool]
- ["Иван", 4.5] :: \perp
- [1]:2 :: ?
 $\alpha : [\alpha]$

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`

Примери

- [False] :: [Bool]
- ["Иван", 4.5] :: \perp
- [1]:2 :: \perp
- [[1,2],[3],[4,5,6]] :: ?

Примери

- [False] :: [Bool]
- ["Иван", 4.5] :: \perp
- [1]:2 :: \perp
- [[1,2], [3], [4,5,6]] :: [[Int]]

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`
- `[[1,2],[3],[4,5,6]] :: [[Int]]`
- `([1,2],[3],[4,5,6]) :: ?`

Примери

- [False] :: [Bool]
- ["Иван", 4.5] :: \perp
- [1]:2 :: \perp
- [[1,2],[3],[4,5,6]] :: [[Int]]
- ([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])

Примери

- [False] :: [Bool]
- ["Иван", 4.5] :: \perp
- [1]:2 :: \perp
- [[1,2], [3], [4,5,6]] :: [[Int]]
- ([1,2], [3], [4,5,6]) :: ([Int], [Int], [Int])
- [(1,2), (3), (4,5,6)] :: ?
 $(\text{Int}, \text{Int}) \text{ Int } ((\text{Int}, \text{Int}), \text{Int})$

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`
- `[[1,2],[3],[4,5,6]] :: [[Int]]`
- `([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])`
- `[(1,2),(3),(4,5,6)] :: ⊥`

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`
- `[[1,2],[3],[4,5,6]] :: [[Int]]`
- `([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])`
- `[(1,2),(3),(4,5,6)] :: ⊥`
- `((1,2),(3),(4,5,6)) :: ?`

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`
- `[[1,2],[3],[4,5,6]] :: [[Int]]`
- `([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])`
- `[(1,2),(3),(4,5,6)] :: ⊥`
- `((1,2),(3),(4,5,6)) :: ((Int,Int),Int,(Int,Int,Int))`

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`
- `[[1,2],[3],[4,5,6]] :: [[Int]]`
- `([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])`
- `[(1,2),(3),(4,5,6)] :: ⊥`
- `((1,2),(3),(4,5,6)) :: ((Int,Int),Int,(Int,Int,Int))`
- `[[]] :: ?`

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`
- `[[1,2],[3],[4,5,6]] :: [[Int]]`
- `([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])`
- `[(1,2),(3),(4,5,6)] :: ⊥`
- `((1,2),(3),(4,5,6)) :: ((Int,Int),Int,(Int,Int,Int))`
- `[[]] :: [[a]]`

Примери

- `[False] :: [Bool]`
 - `["Иван", 4.5] :: ⊥`
 - `[1]:2 :: ⊥`
 - `[[1,2],[3],[4,5,6]] :: [[Int]]`
 - `([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])`
 - `[(1,2),(3),(4,5,6)] :: ⊥`
 - `((1,2),(3),(4,5,6)) :: ((Int,Int),Int,(Int,Int,Int))`
 - `[[]] :: [[a]]`
 - `[]:[] :: ?`
- $x:[\] \Leftrightarrow [\ x]$

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`
- `[[1,2],[3],[4,5,6]] :: [[Int]]`
- `([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])`
- `[(1,2),(3),(4,5,6)] :: ⊥`
- `((1,2),(3),(4,5,6)) :: ((Int,Int),Int,(Int,Int,Int))`
- `[[]] :: [[a]]`
- `[]:[[]] :: [[a]]`

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`
- `[[1,2],[3],[4,5,6]] :: [[Int]]`
- `([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])`
- `[(1,2),(3),(4,5,6)] :: ⊥`
- `((1,2),(3),(4,5,6)) :: ((Int,Int),Int,(Int,Int,Int))`
- `[] :: [[a]]`
- `[]:[] :: [[a]]`
- `[1]:[] :: ?`

$(:) :: b \rightarrow [b] \rightarrow [b]$

b $a = Int$
 b $[b] = [[Int]]$

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`
- `[[1,2],[3],[4,5,6]] :: [[Int]]`
- `([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])`
- `[(1,2),(3),(4,5,6)] :: ⊥`
- `((1,2),(3),(4,5,6)) :: ((Int,Int),Int,(Int,Int,Int))`
- `[[]] :: [[a]]`
- `[]:[[]] :: [[a]]`
- `[1]:[][] :: [[Int]]`

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`
- `[[1,2],[3],[4,5,6]] :: [[Int]]`
- `([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])`
- `[(1,2),(3),(4,5,6)] :: ⊥`
- `((1,2),(3),(4,5,6)) :: ((Int,Int),Int,(Int,Int,Int))`
- `[[]] :: [[a]]`
- `[]:[[]] :: [[a]]`
- `[1]: [[]] :: [[Int]]`
- `[]:[1] :: ?`

b [b]
[a] b=Int

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`
- `[[1,2],[3],[4,5,6]] :: [[Int]]`
- `([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])`
- `[(1,2),(3),(4,5,6)] :: ⊥`
- `((1,2),(3),(4,5,6)) :: ((Int,Int),Int,(Int,Int,Int))`
- `[[]] :: [[a]]`
- `[]:[[]] :: [[a]]`
- `[1]: [[]] :: [[Int]]`
- `[]:[1] :: ⊥`

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`
- `[[1,2],[3],[4,5,6]] :: [[Int]]`
- `([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])`
- `[(1,2),(3),(4,5,6)] :: ⊥`
- `((1,2),(3),(4,5,6)) :: ((Int,Int),Int,(Int,Int,Int))`
- `[[]] :: [[a]]`
- `[]:[[]] :: [[a]]`
- `[1]: [[]] :: [[Int]]`
- `[]:[1] :: ⊥`
- `[[1,2,3],[]] :: ?`

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`
- `[[1,2],[3],[4,5,6]] :: [[Int]]`
- `([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])`
- `[(1,2),(3),(4,5,6)] :: ⊥`
- `((1,2),(3),(4,5,6)) :: ((Int,Int),Int,(Int,Int,Int))`
- `[[]] :: [[a]]`
- `[]:[[]] :: [[a]]`
- `[1]: [[]] :: [[Int]]`
- `[]:[1] :: ⊥`
- `[[1,2,3],[]] :: [[Int]]`

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`
- `[[1,2],[3],[4,5,6]] :: [[Int]]`
- `([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])`
- `[(1,2),(3),(4,5,6)] :: ⊥`
- `((1,2),(3),(4,5,6)) :: ((Int,Int),Int,(Int,Int,Int))`
- `[[]] :: [[a]]`
- `[]:[[]] :: [[a]]`
- `[1]: [[]] :: [[Int]]`
- `[]:[1] :: ⊥`
- `[[1,2,3],[]] :: [[Int]]`
- `[[1,2,3],[[]]] :: ?`

[Int] [[a]] $\text{Int} \neq [\text{a}]$

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`
- `[[1,2],[3],[4,5,6]] :: [[Int]]`
- `([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])`
- `[(1,2),(3),(4,5,6)] :: ⊥`
- `((1,2),(3),(4,5,6)) :: ((Int,Int),Int,(Int,Int,Int))`
- `[[]] :: [[a]]`
- `[]:[[]] :: [[a]]`
- `[1]: [[]] :: [[Int]]`
- `[]:[1] :: ⊥`
- `[[1,2,3],[]] :: [[Int]]`
- `[[1,2,3], [[]]] :: ⊥`

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`
- `[[1,2],[3],[4,5,6]] :: [[Int]]`
- `([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])`
- `[(1,2),(3),(4,5,6)] :: ⊥`
- `((1,2),(3),(4,5,6)) :: ((Int,Int),Int,(Int,Int,Int))`
- `[[]] :: [[a]]`
- `[]:[[]] :: [[a]]`
- `[1]: [[]] :: [[Int]]`
- `[]:[1] :: ⊥`
- `[[1,2,3],[]] :: [[Int]]`
- `[[1,2,3],[[]]] :: ⊥`
- `[1,2,3]:[4,5,6]: [[]] :: ?`

Примери

- `[False] :: [Bool]`
- `["Иван", 4.5] :: ⊥`
- `[1]:2 :: ⊥`
- `[[1,2],[3],[4,5,6]] :: [[Int]]`
- `([1,2],[3],[4,5,6]) :: ([Int],[Int],[Int])`
- `[(1,2),(3),(4,5,6)] :: ⊥`
- `((1,2),(3),(4,5,6)) :: ((Int,Int),Int,(Int,Int,Int))`
- `[[]] :: [[a]]`
- `[]:[[]] :: [[a]]`
- `[1]: [[]] :: [[Int]]`
- `[]:[1] :: ⊥`
- `[[1,2,3],[]] :: [[Int]]`
- `[[1,2,3],[[]]] :: ⊥`
- `[1,2,3]:[4,5,6]: [[]] :: [[Int]]`

Низове

- В Haskell низовете са представени като списъци от символи

Низове

- В Haskell низовете са представени като списъци от символи
- `type String = [Char]`

Низове

- В Haskell низовете са представени като списъци от символи
- `type String = [Char]`
- Всички операции над списъци важат и над низове

Низове

- В Haskell низовете са представени като списъци от символи
- `type String = [Char]`
- Всички операции над списъци важат и над низове
- Примери:

Низове

- В Haskell низовете са представени като списъци от символи
- `type String = [Char]`
- Всички операции над списъци важат и над низове
- **Примери:**
 - `['H', 'e', 'l', 'l', 'o'] = "Hello"`

Низове

- В Haskell низовете са представени като списъци от символи
- `type String = [Char]`
- Всички операции над списъци важат и над низове
- **Примери:**
 - `['H', 'e', 'l', 'l', 'o'] = "Hello"`
 - `'H':'e':'l':'l':'o':[] = "Hello"`

Низове

- В Haskell низовете са представени като списъци от символи
- `type String = [Char]`
- Всички операции над списъци важат и над низове
- **Примери:**
 - `[’H’, ’e’, ’l’, ’l’, ’o’] = "Hello"`
 - `’H’:’e’:’l’:’l’:’o’:[] = "Hello"`
 - `’H’:’e’:”llo” = "Hello"`

Низове

- В Haskell низовете са представени като списъци от символи
- `type String = [Char]`
- Всички операции над списъци важат и над низове
- **Примери:**
 - `[’H’, ’e’, ’l’, ’l’, ’o’] = "Hello"`
 - `’H’:’e’:’l’:’l’:’o’:[] = "Hello"`
 - `’H’:’e’: "llo" = "Hello"`
 - `"" = [] :: [Char]`

Низове

- В Haskell низовете са представени като списъци от символи
- `type String = [Char]`
- Всички операции над списъци важат и над низове
- Примери:
 - `['H', 'e', 'l', 'l', 'o'] = "Hello"`
 - `'H':'e':'l':'l':'o':[] = "Hello"`
 - `'H':'e':"llo" = "Hello"`
 - `"" = [] :: [Char]`
 - `[[1,2,3],"]" :: ?`

[Int] [Char]

Низове

- В Haskell низовете са представени като списъци от символи
- `type String = [Char]`
- Всички операции над списъци важат и над низове
- **Примери:**
 - `[’H’, ’e’, ’l’, ’l’, ’o’] = "Hello"`
 - `’H’:’e’:’l’:’l’:’o’:[] = "Hello"`
 - `’H’:’e’:”llo” = "Hello"`
 - `"" = [] :: [Char]`
 - `[[1,2,3],"] :: _`

Низове

- В Haskell низовете са представени като списъци от символи
- `type String = [Char]`
- Всички операции над списъци важат и над низове
- Примери:
 - `['H', 'e', 'l', 'l', 'o'] = "Hello"`
 - `'H':'e':'l':'l':'o':[] = "Hello"`
 - `'H':'e':"llo" = "Hello"`
 - `"" = [] :: [Char]`
 - `[[1,2,3], ""] :: _`
 - `["12", '3', []] :: ?`

String [Char] · [α]

Низове

- В Haskell низовете са представени като списъци от символи
- `type String = [Char]`
- Всички операции над списъци важат и над низове
- Примери:
 - `['H', 'e', 'l', 'l', 'o'] = "Hello"`
 - `'H':'e':'l':'l':'o':[] = "Hello"`
 - `'H':'e':"llo" = "Hello"`
 - `"" = [] :: [Char]`
 - `[[1,2,3],"]"] :: [String]`
 - `["12",['3'],[]] :: [String] = [[Char]]`

Основни функции за списъци

- **head** :: [a] -> a — връща главата на (непразен) списък

Основни функции за списъци

- **head** :: [a] -> a — връща главата на (непразен) списък
 - **head** [[1,2],[3,4]] → ?

Основни функции за списъци

- **head** :: [a] -> a — връща главата на (непразен) списък
 - **head** [[1,2],[3,4]] → [1,2]

Основни функции за списъци

- **head** :: [a] -> a — връща главата на (непразен) списък
 - **head** [[1,2],[3,4]] → [1,2]
 - **head** [] → Грешка!

Основни функции за списъци

- **head** :: [a] -> a — връща главата на (непразен) списък
 - **head** [[1,2],[3,4]] → [1,2]
 - **head** [] → Грешка!
- **tail** :: [a] -> [a] — връща опашката на (непразен) списък

Основни функции за списъци

- **head** :: [a] → a — връща главата на (непразен) списък
 - **head** [[1,2],[3,4]] → [1,2]
 - **head** [] → Грешка!
- **tail** :: [a] → [a] — връща опашката на (непразен) списък
 - **tail** [[1,2],[3,4]] → ?

Основни функции за списъци

- **head** :: [a] → a — връща главата на (непразен) списък
 - **head** [[1,2],[3,4]] → [1,2]
 - **head** [] → Грешка!
- **tail** :: [a] → [a] — връща опашката на (непразен) списък
 - **tail** [[1,2],[3,4]] → [[3,4]]

$\text{[1,2]} : \text{[3,4]} : []$
 head tail

Основни функции за списъци

- **head** :: [a] → a — връща главата на (непразен) списък
 - **head** [[1,2],[3,4]] → [1,2]
 - **head** [] → Грешка!
- **tail** :: [a] → [a] — връща опашката на (непразен) списък
 - **tail** [[1,2],[3,4]] → [[3,4]]
 - **tail** [] → Грешка!

Основни функции за списъци

- **head** :: [a] -> a — връща главата на (непразен) списък
 - **head** [[1,2],[3,4]] → [1,2]
 - **head** [] → Грешка!
- **tail** :: [a] -> [a] — връща опашката на (непразен) списък
 - **tail** [[1,2],[3,4]] → [[3,4]]
 - **tail** [] → Грешка!
- **null** :: [a] -> Bool — проверява дали списък е празен

Основни функции за списъци

- `head` :: [a] -> a — връща главата на (непразен) списък
 - `head` [[1,2],[3,4]] → [1,2]
 - `head` [] → Грешка!
- `tail` :: [a] -> [a] — връща опашката на (непразен) списък
 - `tail` [[1,2],[3,4]] → [[3,4]]
 - `tail` [] → Грешка!
- `null` :: [a] -> Bool — проверява дали списък е празен
- `length` :: [a] -> Int — дължина на списък

Генератори на списъци

Можем да генерираме списъци от последователни елементи

- $[a..b] \rightarrow [a, a+1, a+2, \dots, b]$
- Пример: $[1..5] \rightarrow [1, 2, 3, 4, 5]$
- Пример: $['a'..'e'] \rightarrow "abcde"$
- Синтактична захар за enumFromTo from to

Генератори на списъци

Можем да генерираме списъци от последователни елементи

- $[a..b] \rightarrow [a, a+1, a+2, \dots, b]$
- Пример: $[1..5] \rightarrow [1, 2, 3, 4, 5]$
- Пример: $['a'..'e'] \rightarrow \text{"abcde"}$
- Синтактична захар за enumFromTo from to
- $[a, a + \Delta x \dots b] \rightarrow [a, a + \Delta x, a + 2\Delta x, \dots, b']$, където b' е най-голямото число $\leq b$, за което $b' = a + k\Delta x$
- Пример: $[1, 4..15] \rightarrow [1, 4, 7, 10, 13]$
- Пример: $['a', 'e'..'z'] \rightarrow \text{"aeimquy"}$
- Синтактична захар за enumFromThenTo from then to

Рекурсивни функции над списъци

- $(++) :: [a] \rightarrow [a] \rightarrow [a]$ — слепва два списъка
 - $[1..3] ++ [5..7] \longrightarrow [1, 2, 3, 5, 6, 7]$

Рекурсивни функции над списъци

- $(++) :: [a] \rightarrow [a] \rightarrow [a]$ — слепва два списъка
 - $[1..3] ++ [5..7] \longrightarrow [1,2,3,5,6,7]$
- $a ++ b = \text{if } \text{null } a \text{ then } b \text{ else head } a : \text{tail } a ++ b$

Рекурсивни функции над списъци

- `(++) :: [a] -> [a]` -> `[a]` — слепва два списъка
 - `[1..3] ++ [5..7] —> [1,2,3,5,6,7]`
- `a ++ b = if null a then b else head a : tail a ++ b`
- `reverse :: [a] -> [a]` — обръща списък
 - `reverse [1..5] —> [5,4,3,2,1]`

Рекурсивни функции над списъци

- $(++) :: [a] \rightarrow [a] \rightarrow [a]$ — слепва два списъка
 - $[1..3] ++ [5..7] \longrightarrow [1,2,3,5,6,7]$
- $a ++ b = \text{if } \text{null } a \text{ then } b \text{ else } \text{head } a : \text{tail } a ++ b$
- $\text{reverse} :: [a] \rightarrow [a]$ — обръща списък
 - $\text{reverse } [1..5] \longrightarrow [5,4,3,2,1]$

```
reverse a
| null a      = a
| otherwise = reverse (tail a) ++ [head a]
```

Рекурсивни функции над списъци

- `(++) :: [a] -> [a]` — слепва два списъка
 - `[1..3] ++ [5..7] → [1,2,3,5,6,7]`
- `a ++ b = if null a then b else head a : tail a ++ b`
- `reverse :: [a] -> [a]` — обръща списък
 - `reverse [1..5] → [5,4,3,2,1]`

```
reverse a
| null a      = a
| otherwise = reverse (tail a) ++ [head a]
```

- `(!!) :: [a] -> Int -> a` — елемент с пореден номер (от 0)
 - "Haskell" !! 2 → 's'

Рекурсивни функции над списъци

- `(++) :: [a] -> [a] -> [a]` — слепва два списъка
 - `[1..3] ++ [5..7] → [1,2,3,5,6,7]`
- `a ++ b = if null a then b else head a : tail a ++ b`
- `reverse :: [a] -> [a]` — обръща списък
 - `reverse [1..5] → [5,4,3,2,1]`

```
reverse a
| null a      = a
| otherwise = reverse (tail a) ++ [head a]
```

- `(!!) :: [a] -> Int -> a` — елемент с пореден номер (от 0)
 - `"Haskell" !! 2 → 's'`
- `elem :: Eq a => a -> [a] -> Bool` — проверка за принадлежност на елемент към списък
 - `3 `elem` [1..5] → True`

Образци и списъци

Много удобно е да използваме образци на списъци:

- $p_h : p_t$ — пасва на всеки непразен списък l , за който:

Образци и списъци

Много удобно е да използваме образци на списъци:

- $p_h : p_t$ — пасва на всеки непразен списък l , за който:
 - образецът p_h пасва на главата на l

Образци и списъци

Много удобно е да използваме образци на списъци:

- $p_h : p_t$ — пасва на всеки непразен списък l , за който:
 - образецът p_h пасва на главата на l
 - образецът p_t пасва на опашката на l

Образци и списъци

Много удобно е да използваме образци на списъци:

- $p_h : p_t$ — пасва на всеки непразен списък $/$, за който:
 - образецът p_h пасва на главата на $/$
 - образецът p_t пасва на опашката на $/$
- **Внимание:** обикновено слагаме скоби (`h:t`), понеже операцията `:` е с много нисък приоритет

Образци и списъци

Много удобно е да използваме образци на списъци:

- $p_h : p_t$ — пасва на всеки непразен списък l , за който:
 - образецът p_h пасва на главата на l
 - образецът p_t пасва на опашката на l
- **Внимание:** обикновено слагаме скоби (`h:t`), понеже операцията : е с много нисък приоритет
- $[p_1, p_2, \dots, p_n]$ — пасва на всеки списък от точно n елемента $[x_1, x_2, \dots, x_n]$, за който образецът p_i пасва на елемента x_i ;

Образци и списъци

Много удобно е да използваме образци на списъци:

- $p_h : p_t$ — пасва на всеки непразен списък l , за който:
 - образецът p_h пасва на главата на l
 - образецът p_t пасва на опашката на l
- **Внимание:** обикновено слагаме скоби (`h:t`), понеже операцията : е с много нисък приоритет
- $[p_1, p_2, \dots, p_n]$ — пасва на всеки списък от точно n елемента $[x_1, x_2, \dots, x_n]$, за който образецът p_i пасва на элемента x_i ;
- **Примери:**

Образци и списъци

Много удобно е да използваме образци на списъци:

- $p_h : p_t$ — пасва на всеки непразен списък l , за който:
 - образецът p_h пасва на главата на l
 - образецът p_t пасва на опашката на l
- **Внимание:** обикновено слагаме скоби ($h:t$), понеже операцията : е с много нисък приоритет
- $[p_1, p_2, \dots, p_n]$ — пасва на всеки списък от точно n елемента $[x_1, x_2, \dots, x_n]$, за който образецът p_i пасва на елемента x_i ;
- **Примери:**
 - `head (h:_)` = `h`

Образци и списъци

Много удобно е да използваме образци на списъци:

- $p_h : p_t$ — пасва на всеки непразен списък l , за който:
 - образецът p_h пасва на главата на l
 - образецът p_t пасва на опашката на l
- **Внимание:** обикновено слагаме скоби ($h:t$), понеже операцията : е с много нисък приоритет
- $[p_1, p_2, \dots, p_n]$ — пасва на всеки списък от точно n елемента $[x_1, x_2, \dots, x_n]$, за който образецът p_i пасва на елемента x_i ;
- **Примери:**
 - `head (h:_)` = h
 - `tail (_:t)` = t

Образци и списъци

Много удобно е да използваме образци на списъци:

- $p_h : p_t$ — пасва на всеки непразен списък $/$, за който:
 - образецът p_h пасва на главата на $/$
 - образецът p_t пасва на опашката на $/$
- **Внимание:** обикновено слагаме скоби ($h:t$), понеже операцията : е с много нисък приоритет
- $[p_1, p_2, \dots, p_n]$ — пасва на всеки списък от точно n елемента $[x_1, x_2, \dots, x_n]$, за който образецът p_i пасва на елемента x_i ;
- **Примери:**
 - `head (h:_)` = `h`
 - `tail (_:t)` = `t`
 - `null []` = `True`
 - `null _` = `False`

Образци и списъци

Много удобно е да използваме образци на списъци:

- $p_h : p_t$ — пасва на всеки непразен списък $/$, за който:
 - образецът p_h пасва на главата на $/$
 - образецът p_t пасва на опашката на $/$
- **Внимание:** обикновено слагаме скоби $(h:t)$, понеже операцията : е с много нисък приоритет
- $[p_1, p_2, \dots, p_n]$ — пасва на всеки списък от точно n елемента $[x_1, x_2, \dots, x_n]$, за който образецът p_i пасва на елемента x_i ;
- **Примери:**
 - `head (h:_)` = h
 - `tail (_:t)` = t
 - `null []` = `True`
 - `null _` = `False`
 - `length []` = 0
 - `length (_:t)` = $1 + \text{length } t$

Случаи по образци (case)

- case <израз> of { <образец> -> <израз> }⁺

Случаи по образци (case)

- `case <израз> of { <образец> -> <израз> }+`
- `case <израз> of <образец1> -> <израз1>`
...

`<образецn> -> <изразn>`

Случаи по образци (case)

- case <израз> of { <образец> -> <израз> }⁺
- case <израз> of <образец₁> -> <израз₁>
 - ...
 - <образец_n> -> <израз_n>
- ако <израз> пасва на <образец₁>, връща <израз₁>, иначе:
- ...
- ако <израз> пасва на <образец_n>, връща <израз_n>, иначе:
- Грешка!

Случаи по образци (case)

- `case <израз> of { <образец> -> <израз> }+`
- `case <израз> of <образец1> -> <израз1>`
...
`<образецn> -> <изразn>`
- ако `<израз>` пасва на `<образец1>`, връща `<израз1>`, иначе:
- ...
- ако `<израз>` пасва на `<образецn>`, връща `<изразn>`, иначе:
- **Грешка!**
- Използването на образци в дефиниции всъщност е синтактична захар за конструкцията `case`!

Случаи по образци (case)

- `case <израз> of { <образец> -> <израз> }+`
- `case <израз> of <образец1> -> <израз1>`
...
`<образецn> -> <изразn>`
- ако `<израз>` пасва на `<образец1>`, връща `<израз1>`, иначе:
- ...
- ако `<израз>` пасва на `<образецn>`, връща `<изразn>`, иначе:
- **Грешка!**
- Използването на образци в дефиниции всъщност е синтактична захар за конструкцията `case`!
- `case` може да се използва навсякъде, където се очаква израз

Полиморфни функции

Функциите `head`, `tail`, `null`, `length`, `reverse` и операциите `++` и `!!` са полиморфни

- работят над списъци с елементи от произволен тип `[t]`

Полиморфни функции

Функциите `head`, `tail`, `null`, `length`, `reverse` и операциите `++` и `!!` са полиморфни

- работят над списъци с елементи от произволен тип `[t]`
- `t` се нарича **типова променлива**

Полиморфни функции

Функциите `head`, `tail`, `null`, `length`, `reverse` и операциите `++` и `!!` са полиморфни

- работят над списъци с елементи от произволен тип `[t]`
- `t` се нарича **типова променлива**
- свойството се нарича **параметричен типов полиморфизъм**

Полиморфни функции

Функциите `head`, `tail`, `null`, `length`, `reverse` и операциите `++` и `!!` са полиморфни

- работят над списъци с елементи от произволен тип `[t]`
- `t` се нарича **типова променлива**
- свойството се нарича **параметричен типов полиморфизъм**
- подобно на шаблоните в C++

Полиморфни функции

Функциите `head`, `tail`, `null`, `length`, `reverse` и операциите `++` и `!!` са полиморфни

- работят над списъци с елементи от произволен тип `[t]`
- `t` се нарича **типова променлива**
- свойството се нарича **параметричен типов полиморфизъм**
- подобно на шаблоните в C++
- да не се бърка с **подтипов полиморфизъм**, реализиран с виртуални функции!

Полиморфни функции

Функциите `head`, `tail`, `null`, `length`, `reverse` и операциите `++` и `!!` са полиморфни

- работят над списъци с елементи от произволен тип `[t]`
- `t` се нарича **типова променлива**
- свойството се нарича **параметричен типов полиморфизъм**
- подобно на шаблоните в C++
- да не се бърка с **подтипов полиморфизъм**, реализиран с **виртуални функции!**
- `[]` е **полиморфна константа**

Класове от типове (typeclasses)

Функцията `elem` има специални изисквания към типа на елементите на списъка: трябва да могат да бъдат сравнявани с `==` или `/=`

- `elem :: Eq t => t -> [t] -> Bool`

Класове от типове (typeclasses)

Функцията `elem` има специални изисквания към типа на елементите на списъка: трябва да могат да бъдат сравнявани с `==` или `/=`

- `elem :: Eq t => t -> [t] -> Bool`
- `Eq` е клас от типове

$$A = \{n \in \mathbb{N} \mid 2/n \in \mathbb{N}\} \quad A \subseteq \mathbb{N}$$

$$\alpha(n) = n \bmod 1 \quad 2 = 0$$

$$A = \{n \in \mathbb{N} \mid \alpha(n) = \text{True}\}$$

$$\alpha(n) = "n \in A"$$

Класове от типове (typeclasses)

Функцията `elem` има специални изисквания към типа на елементите на списъка: трябва да могат да бъдат сравнявани с `==` или `/=`

- `elem :: Eq t => t -> [t] -> Bool`
- `Eq` е **клас от типове**
- `Eq` е **класът на тези типове**, за които има операции `==` и `/=`

Класове от типове (typeclasses)

Функцията `elem` има специални изисквания към типа на елементите на списъка: трябва да могат да бъдат сравнявани с `==` или `/=`

- `elem :: Eq t => t -> [t] -> Bool`
- `Eq` е **клас от типове**
- `Eq` е **класът на тези типове**, за които има операции `==` и `/=`
 - можем да си мислим за класовете от типове като за “интерфейси”

Класове от типове (typeclasses)

Функцията `elem` има специални изисквания към типа на елементите на списъка: трябва да могат да бъдат сравнявани с `==` или `/=`

- `elem :: Eq t => t -> [t] -> Bool`
- `Eq` е **клас от типове**
- `Eq` е класът на тези типове, за които има операции `==` и `/=`
 - можем да си мислим за класовете от типове като за “интерфейси”
- `Eq t` наричаме **класово ограничение** за типа `t` (class constraint)

Класове от типове (typeclasses)

Функцията `elem` има специални изисквания към типа на елементите на списъка: трябва да могат да бъдат сравнявани с `==` или `/=`

- `elem :: Eq t => t -> [t] -> Bool`
- `Eq` е **клас от типове**
- `Eq` е класът на тези типове, за които има операции `==` и `/=`
 - можем да си мислим за класовете от типове като за “интерфейси”
- `Eq t` наричаме **класово ограничение** за типа `t` (class constraint)
- множеството от всички класови ограничения наричаме **контекст**

Класове от типове (typeclasses)

Функцията `elem` има специални изисквания към типа на елементите на списъка: трябва да могат да бъдат сравнявани с `==` или `/=`

- `elem :: Eq t => t -> [t] -> Bool`
- **Eq** е **клас от типове**
- **Eq** е класът на тези типове, за които има операции `==` и `/=`
 - можем да си мислим за класовете от типове като за “интерфейси”
- **Eq t** наричаме **класово ограничение** за типа **t** (class constraint)
- множеството от всички класови ограничения наричаме **контекст**
- **инстанция** на клас от типове наричаме всеки тип, за който са реализирани операциите зададени в класа

Класове от типове (typeclasses)

Функцията `elem` има специални изисквания към типа на елементите на списъка: трябва да могат да бъдат сравнявани с `==` или `/=`

- `elem :: Eq t => t -> [t] -> Bool`
- `Eq` е **клас от типове**
- `Eq` е класът на тези типове, за които има операции `==` и `/=`
 - можем да си мислим за класовете от типове като за “интерфейси”
- `Eq t` наричаме **класово ограничение** за типа `t` (class constraint)
- множеството от всички класови ограничения наричаме **контекст**
- **инстанция** на клас от типове наричаме всеки тип, за който са реализирани операциите зададени в класа
- инстанции на `Eq` са:

Класове от типове (typeclasses)

Функцията `elem` има специални изисквания към типа на елементите на списъка: трябва да могат да бъдат сравнявани с `==` или `/=`

- `elem :: Eq t => t -> [t] -> Bool`
- `Eq` е **клас от типове**
- `Eq` е класът на тези типове, за които има операции `==` и `/=`
 - можем да си мислим за класовете от типове като за “интерфейси”
- `Eq t` наричаме **класово ограничение** за типа `t` (class constraint)
- множеството от всички класови ограничения наричаме **контекст**
- **инстанция** на клас от типове наричаме всеки тип, за който са реализирани операциите зададени в класа
- инстанции на `Eq` са:
 - `Bool, Char`, всички числови типове (`Int, Integer, Float, Double`)

Класове от типове (typeclasses)

Функцията `elem` има специални изисквания към типа на елементите на списъка: трябва да могат да бъдат сравнявани с `==` или `/=`

- `elem :: Eq t => t -> [t] -> Bool`
- `Eq` е **клас от типове**
- `Eq` е класът на тези типове, за които има операции `==` и `/=`
 - можем да си мислим за класовете от типове като за “интерфейси”
- `Eq t` наричаме **класово ограничение** за типа `t` (class constraint)
- множеството от всички класови ограничения наричаме **контекст**
- **инстанция** на клас от типове наричаме всеки тип, за който са реализирани операциите зададени в класа
- инстанции на `Eq` са:
 - `Bool, Char`, всички числови типове (`Int, Integer, Float, Double`)
 - списъчните типове `[t]`, за които `t` е инстанция на `Eq`

Класове от типове (typeclasses)

Функцията `elem` има специални изисквания към типа на елементите на списъка: трябва да могат да бъдат сравнявани с `==` или `/=`

- `elem :: Eq t => t -> [t] -> Bool`
- `Eq` е **клас от типове**
- `Eq` е класът на тези типове, за които има операции `==` и `/=`
 - можем да си мислим за класовете от типове като за “интерфейси”
- `Eq t` наричаме **класово ограничение** за типа `t` (class constraint)
- множеството от всички класови ограничения наричаме **контекст**
- **инстанция** на клас от типове наричаме всеки тип, за който са реализирани операциите зададени в класа
- инстанции на `Eq` са:
 - `Bool, Char`, всички числови типове (`Int, Integer, Float, Double`)
 - списъчните типове `[t]`, за които `t` е инстанция на `Eq`
 - кортежните типове `(t1, ..., tn)`, за които `ti` са инстанции на `Eq`

Стандартни класове

Някои от по-често използваните класове на Haskell:

- **Eq** — типове с равенство

Стандартни класове

Някои от по-често използваните класове на Haskell:

- `Eq` — типове с равенство
- `Ord` — типове с (линейна) наредба
 - операциите `==`, `/=`, `>=`, `<=`, `<`, `>`
 - специалната функция `compare`, която сравнява два елемента и връща `LT`, `GT` или `EQ` в зависимост от резултата
 - функциите `min` и `max`

Стандартни класове

Някои от по-често използваните класове на Haskell:

- `Eq` — типове с равенство
- `Ord` — типове с (линейна) наредба
 - операциите `==`, `/=`, `>=`, `<=`, `<`, `>`
 - специалната функция `compare`, която сравнява два елемента и връща `LT`, `GT` или `EQ` в зависимост от резултата
 - функциите `min` и `max`
- `Show` — типове, чиито елементи могат да бъдат извеждани в низ
 - функция `show :: a -> String`

Стандартни класове

Някои от по-често използваните класове на Haskell:

- **Eq** — типове с равенство
- **Ord** — типове с (линейна) наредба
 - операциите `==`, `/=`, `>=`, `<=`, `<`, `>`
 - специалната функция `compare`, която сравнява два елемента и връща `LT`, `GT` или `EQ` в зависимост от резултата
 - функциите `min` и `max`
- **Show** — типове, чиито елементи могат да бъдат извеждани в низ
 - функция `show :: a -> String`
- **Read** — типове, чиито елементи могат да бъдат въвеждани от низ
 - функция `read :: String -> a`

Стандартни класове

Някои от по-често използваните класове на Haskell:

- **Eq** — типове с равенство
- **Ord** — типове с (линейна) наредба
 - операциите `==`, `/=`, `>=`, `<=`, `<`, `>`
 - специалната функция `compare`, която сравнява два елемента и връща `LT`, `GT` или `EQ` в зависимост от резултата
 - функциите `min` и `max`
- **Show** — типове, чиито елементи могат да бъдат извеждани в низ
 - функция `show :: a -> String`
- **Read** — типове, чиито елементи могат да бъдат въвеждани от низ
 - функция `read :: String -> a`
- **Num** — числови типове
- **Integral** — целочислени типове
- **Floating** — типове с плаваща запетая

Стандартни класове

Някои от по-често използваните класове на Haskell:

- **Eq** — типове с равенство
- **Ord** — типове с (линейна) наредба
 - операциите `==`, `/=`, `>=`, `<=`, `<`, `>`
 - специалната функция `compare`, която сравнява два елемента и връща `LT`, `GT` или `EQ` в зависимост от резултата
 - функциите `min` и `max`
- **Show** — типове, чиито елементи могат да бъдат извеждани в низ
 - функция `show :: a -> String`
- **Read** — типове, чиито елементи могат да бъдат въвеждани от низ
 - функция `read :: String -> a`
- **Num** — числови типове
- **Integral** — целочислени типове
- **Floating** — типове с плаваща запетая
- **числата в Haskell са полиморфни константи!**

Отделяне на списъци (list comprehension)

Отделянето на списъци е удобен начин за дефиниране на нови списъци чрез използване на дадени такива

- [<израз> | <генератор> {, <генератор>} {, <условие>}]

$$\begin{aligned} \{x \in A \mid p(x)\} &\subseteq A \\ \{x^2 \mid x \in \{-\ell, \dots, r\}\} & \end{aligned}$$

Отделяне на списъци (list comprehension)

Отделянето на списъци е удобен начин за дефиниране на нови списъци чрез използване на дадени такива

- [<израз> | <генератор> {, <генератор>} {, <условие>}]
- <генератор> е от вида <образец> <-> <израз>, където

Отделяне на списъци (list comprehension)

Отделянето на списъци е удобен начин за дефиниране на нови списъци чрез използване на дадени такива

- [<израз> | <генератор> {, <генератор>} {, <условие>}]
- <генератор> е от вида <образец> <-> <израз>, където
 - <израз> е от тип списък [a]

Отделяне на списъци (list comprehension)

Отделянето на списъци е удобен начин за дефиниране на нови списъци чрез използване на дадени такива

- [<израз> | <генератор> {, <генератор>} {, <условие>}]
- <генератор> е от вида <образец> <-> <израз>, където
 - <израз> е от тип списък [a]
 - <образец> пасва на елементи от тип a

Отделяне на списъци (list comprehension)

Отделянето на списъци е удобен начин за дефиниране на нови списъци чрез използване на дадени такива

- [<израз> | <генератор> {, <генератор>} {, <условие>}]
- <генератор> е от вида <образец> <-> <израз>, където
 - <израз> е от тип списък [a]
 - <образец> пасва на елементи от тип a
- <условие> е произволен израз от тип Bool

Отделяне на списъци (list comprehension)

Отделянето на списъци е удобен начин за дефиниране на нови списъци чрез използване на дадени такива

- [<израз> | <генератор> {, <генератор>} {, <условие>}]
- <генератор> е от вида <образец> <- <израз>, където
 - <израз> е от тип списък [a]
 - <образец> пасва на елементи от тип a
- <условие> е произволен израз от тип Bool
- За всеки от елементите генериран от <генератор>, които удовлетворяват **всички** <условие>, пресмята <израз> и натрупва резултатите в списък

Примери за отделяне на списъци

Брой Структура

- [2 * x | x <- [1..5]] → ?

Брой Структура

Примери за отделяне на списъци

- [2 * x | x <- [1..5]] \longrightarrow [2,4,6,8,10]

$\{2x \mid x \in [1;5]\}$

Примери за отделяне на списъци

- $[2 * x \mid x <- [1..5]] \longrightarrow [2,4,6,8,10]$
- $[x^2 \mid x <- [1..10], \text{odd } x] \longrightarrow ?$

Примери за отделяне на списъци

- $[2 * x \mid x <- [1..5]] \longrightarrow [2,4,6,8,10]$
- $[x^2 \mid x <- [1..10], \text{odd } x] \longrightarrow [1,9,25,49,81]$

Примери за отделяне на списъци

- $[2 * x \mid x <- [1..5]] \rightarrow [2,4,6,8,10]$
- $[x^2 \mid x <- [1..10], \text{odd } x] \rightarrow [1,9,25,49,81]$
- $[fn \mid (_, fn, grade) \leftarrow \text{students}, grade \geq 2]$

SELECT

FROM

WHERE

Примери за отделяне на списъци

- $[2 * x \mid x <- [1..5]] \longrightarrow [2, 4, 6, 8, 10]$
- $[x^2 \mid x <- [1..10], \text{odd } x] \longrightarrow [1, 9, 25, 49, 81]$
- $[fn \mid (_, fn, grade) <- \text{students}, grade \geq 2]$
- $[x^2 + y^2 \mid (x, y) <- \text{vectors}, x \geq 0, y \geq 0]$

Примери за отделяне на списъци

- $[2 * x \mid x <- [1..5]] \longrightarrow [2, 4, 6, 8, 10]$
- $[x^2 \mid x <- [1..10], \text{odd } x] \longrightarrow [1, 9, 25, 49, 81]$
- $[fn \mid (_, fn, grade) <- \text{students}, grade \geq 2]$
- $[x^2 + y^2 \mid (x, y) <- \text{vectors}, x \geq 0, y \geq 0]$
- Ако имаме повече от един генератор, се генерират всички възможни комбинации от елементи (декартово произведение)

Примери за отделяне на списъци

- `[2 * x | x <- [1..5]]` → `[2,4,6,8,10]`
- `[x^2 | x <- [1..10], odd x]` → `[1,9,25,49,81]`
- `[fn | (_, fn, grade) <- students, grade >= 2]`
- `[x^2 + y^2 | (x, y) <- vectors, x >= 0, y >= 0]`
- Ако имаме повече от един генератор, се генерират всички възможни комбинации от елементи (декартово произведение)
- `[x++(,':y) | x<-["green","blue"], y<-["sky","grass"]]`
→ ?

Примери за отделяне на списъци

- `[2 * x | x <- [1..5]]` → `[2,4,6,8,10]`
 - `[x^2 | x <- [1..10], odd x]` → `[1,9,25,49,81]`
 - `[fn | (_, fn, grade) <- students, grade >= 2]`
 - `[x^2 + y^2 | (x, y) <- vectors, x >= 0, y >= 0]`
 - Ако имаме повече от един генератор, се генерират всички възможни комбинации от елементи (декартово произведение)
 - `[x++(, :y) | x<-["green", "blue"], y<-["sky", "grass"]]`
→ `["green sky", "green grass", "blue sky", "blue grass"]`
- X++" "+y

Примери за отделяне на списъци

- $[2 * x \mid x <- [1..5]] \rightarrow [2, 4, 6, 8, 10]$
- $[x^2 \mid x <- [1..10], \text{odd } x] \rightarrow [1, 9, 25, 49, 81]$
- $[fn \mid (_, fn, grade) <- \text{students}, grade \geq 2]$
- $[x^2 + y^2 \mid (x, y) <- \text{vectors}, x \geq 0, y \geq 0]$
- Ако имаме повече от един генератор, се генерират всички възможни комбинации от елементи (декартово произведение)
 - $[x++(, :y) \mid x <- ["green", "blue"], y <- ["sky", "grass"]] \rightarrow ["green sky", "green grass", "blue sky", "blue grass"]$
 - $[(x, y) \mid x <- [1, 2, 3], y <- [5, 6, 7], x + y \leq 8] \rightarrow ?$

Примери за отделяне на списъци

- `[2 * x | x <- [1..5]]` → `[2,4,6,8,10]`
- `[x^2 | x <- [1..10], odd x]` → `[1,9,25,49,81]`
- `[fn | (_, fn, grade) <- students, grade >= 2]`
- `[x^2 + y^2 | (x, y) <- vectors, x >= 0, y >= 0]`
- Ако имаме повече от един генератор, се генерират всички възможни комбинации от елементи (декартово произведение)
 - `[x++(' ':y) | x<-["green", "blue"], y<-["sky", "grass"]]`
→ `["green sky", "green grass", "blue sky", "blue grass"]`
 - `[(x,y) | x <- [1,2,3], y <- [5,6,7], x + y <= 8]`
→ `[(1,5), (1,6), (1,7), (2,5), (2,6), (3,5)]`

Примери за отделяне на списъци

- $[2 * x \mid x <- [1..5]] \rightarrow [2, 4, 6, 8, 10]$
- $[x^2 \mid x <- [1..10], \text{odd } x] \rightarrow [1, 9, 25, 49, 81]$
- $[fn \mid (_, fn, grade) <- \text{students}, grade \geq 2]$
- $[x^2 + y^2 \mid (x, y) <- \text{vectors}, x \geq 0, y \geq 0]$
- Ако имаме повече от един генератор, се генерират всички възможни комбинации от елементи (декартово произведение)
 - $[x++(, :y) \mid x <- ["green", "blue"], y <- ["sky", "grass"]] \rightarrow ["green sky", "green grass", "blue sky", "blue grass"]$
 - $[(x, y) \mid x <- [1, 2, 3], y <- [5, 6, 7], x + y \leq 8] \rightarrow [(1, 5), (1, 6), (1, 7), (2, 5), (2, 6), (3, 5)]$
- **Задача.** Да се генерират всички Питагорови тройки в даден интервал.

Отрязване на списъци

- `init :: [a] -> [a]` — списъка без последния му елемент
 - `init [1..5] → [1,2,3,4]`

Отрязване на списъци

- `init :: [a] -> [a]` — списъка без последния му елемент
 - `init [1..5] -> [1,2,3,4]`
- `last :: [a] -> a` — последния елемент на списъка
 - `last "Haskell" -> l`

Отрязване на списъци

- `init :: [a] -> [a]` — списъка без последния му елемент
 - `init [1..5] -> [1,2,3,4]`
- `last :: [a] -> a` — последния елемент на списъка
 - `last "Haskell" -> l`
- `take :: Int -> [a] -> [a]` — първите n елемента на списък
 - `take 4 "Hello, world!" -> "Hell"`

Отрязване на списъци

- `init :: [a] -> [a]` — списъка без последния му елемент
 - `init [1..5] -> [1,2,3,4]`
- `last :: [a] -> a` — последния елемент на списъка
 - `last "Haskell" -> l`
- `take :: Int -> [a] -> [a]` — първите n елемента на списък
 - `take 4 "Hello, world!" -> "Hell"`
- `drop :: Int -> [a] -> [a]` — списъка без първите n елемента
 - `drop 2 [1,3..10] -> [5,7,9]`

Отрязване на списъци

- `init :: [a] -> [a]` — списъка без последния му елемент
 - `init [1..5] -> [1,2,3,4]`
- `last :: [a] -> a` — последния елемент на списъка
 - `last "Haskell" -> l`
- `take :: Int -> [a] -> [a]` — първите n елемента на списък
 - `take 4 "Hello, world!" -> "Hell"`
- `drop :: Int -> [a] -> [a]` — списъка без първите n елемента
 - `drop 2 [1,3..10] -> [5,7,9]`
- `splitAt :: Int -> [a] -> ([a], [a])`
 - `splitAt n l = (take n l, drop n l)`

Агрегиращи функции

- **maximum** :: `Ord a => [a] -> a` — максимален елемент
- **minimum** :: `Ord a => [a] -> a` — минимален елемент

Агрегиращи функции

- `maximum :: Ord a => [a] -> a` — максимален елемент
- `minimum :: Ord a => [a] -> a` — минимален елемент
- `sum :: Num a => [a] -> a` — сума на списък от числа
- `product :: Num a => [a] -> a` — произведение на списък от числа

Агрегиращи функции

- `maximum :: Ord a => [a]` -> а — максимален елемент
- `minimum :: Ord a => [a]` -> а — минимален елемент
- `sum :: Num a => [a]` -> а — сума на списък от числа
- `product :: Num a => [a]` -> а — произведение на списък от числа
- `and :: [Bool]` -> `Bool` — конюнкция на булеви стойности
- `or :: [Bool]` -> `Bool` — дизюнкция на булеви стойности

Агрегиращи функции

- `maximum :: Ord a => [a]` -> а — максимален елемент
- `minimum :: Ord a => [a]` -> а — минимален елемент
- `sum :: Num a => [a]` -> а — сума на списък от числа
- `product :: Num a => [a]` -> а — произведение на списък от числа
- `and :: [Bool]` -> `Bool` — конюнкция на булеви стойности
- `or :: [Bool]` -> `Bool` — дизюнкция на булеви стойности
- `concat :: [[a]]` -> `[a]` — конкатенация на списък от списъци

Агрегиращи функции

- `maximum :: Ord a => [a]` -> а — максимален елемент
- `minimum :: Ord a => [a]` -> а — минимален елемент
- `sum :: Num a => [a]` -> а — сума на списък от числа
- `product :: Num a => [a]` -> а — произведение на списък от числа
- `and :: [Bool]` -> `Bool` — конюнкция на булеви стойности
- `or :: [Bool]` -> `Bool` — дизюнкция на булеви стойности
- `concat :: [[a]]` -> `[a]` — конкатенация на списък от списъци
- **Примери:**

Агрегиращи функции

- `maximum :: Ord a => [a]` -> а — максимален елемент
- `minimum :: Ord a => [a]` -> а — минимален елемент
- `sum :: Num a => [a]` -> а — сума на списък от числа
- `product :: Num a => [a]` -> а — произведение на списък от числа
- `and :: [Bool]` -> `Bool` — конюнкция на булеви стойности
- `or :: [Bool]` -> `Bool` — дизюнкция на булеви стойности
- `concat :: [[a]]` -> `[a]` — конкатенация на списък от списъци
- **Примери:**
 - `[(sum 1, product 1) | 1 <- 1..11, maximum 1 == 2*minimum 1]`

Агрегиращи функции

- `maximum :: Ord a => [a]` -> а — максимален елемент
- `minimum :: Ord a => [a]` -> а — минимален елемент
- `sum :: Num a => [a]` -> а — сума на списък от числа
- `product :: Num a => [a]` -> а — произведение на списък от числа
- `and :: [Bool] -> Bool` — конюнкция на булеви стойности
- `or :: [Bool] -> Bool` — дизюнкция на булеви стойности
- `concat :: [[a]] -> [a]` — конкатенация на списък от списъци
- **Примери:**
 - `[(sum 1, product 1) | 1 <- 11, maximum 1 == 2*minimum 1]`
 - `and [or [mod x k == 0 | x <- row] | row <- matrix]`

for row in matrix k/x

λ -функции

- $\lambda \{ <\text{параметър} > \}^+ \rightarrow <\text{тяло}>$

λ-функции

- $\lambda \{ <\text{параметър} > \}^+ \rightarrow <\text{тяло}>$
- $\lambda <\text{параметър}_1> \dots <\text{параметър}_n> \rightarrow <\text{тяло}>$

λ-функции

- $\lambda \{ <\text{параметър} > \}^+ \rightarrow <\text{тяло}>$
- $\lambda <\text{параметър}_1> \dots <\text{параметър}_n> \rightarrow <\text{тяло}>$
- анонимна функция с n параметъра

λ-функции

- $\lambda \{ <\text{параметър} > \}^+ \rightarrow <\text{тяло}>$
- $\lambda <\text{параметър}_1> \dots <\text{параметър}_n> \rightarrow <\text{тяло}>$
- анонимна функция с n параметъра
- всеки $<\text{параметър}_i>$ всъщност е образец

λ-функции

- $\lambda \{ <\text{параметър} > \}^+ \rightarrow <\text{тяло}>$
- $\lambda <\text{параметър}_1> \dots <\text{параметър}_n> \rightarrow <\text{тяло}>$
- анонимна функция с n параметъра
- всеки $<\text{параметър}_i>$ всъщност е образец
- параметрите са видими само в рамките на $<\text{тяло}>$

λ-функции

- $\lambda \{ <\text{параметър} > \}^+ \rightarrow <\text{тяло}>$
- $\lambda <\text{параметър}_1> \dots <\text{параметър}_n> \rightarrow <\text{тяло}>$
- анонимна функция с n параметъра
- всеки $<\text{параметър}_i>$ всъщност е образец
- параметрите са видими само в рамките на $<\text{тяло}>$
- Примери:

λ -функции

- $\lambda \{ <\text{параметър} > \}^+ \rightarrow <\text{тяло}>$
- $\lambda <\text{параметър}_1> \dots <\text{параметър}_n> \rightarrow <\text{тяло}>$
- анонимна функция с n параметъра
- всеки $<\text{параметър}_i>$ всъщност е образец
- параметрите са видими само в рамките на $<\text{тяло}>$
- Примери:
 - `id = \x -> x`

λ-функции

- $\lambda \{ <\text{параметър} > \}^+ \rightarrow <\text{тяло}>$
- $\lambda <\text{параметър}_1> \dots <\text{параметър}_n> \rightarrow <\text{тяло}>$
- анонимна функция с n параметъра
- всеки $<\text{параметър}_i>$ всъщност е образец
- параметрите са видими само в рамките на $<\text{тяло}>$
- Примери:
 - `id = \x -> x`
 - `const = \x y -> x`

λ-функции

- $\lambda \{ <\text{параметър} > \}^+ \rightarrow <\text{тяло}>$
- $\lambda <\text{параметър}_1> \dots <\text{параметър}_n> \rightarrow <\text{тяло}>$
- анонимна функция с n параметъра
- всеки $<\text{параметър}_i>$ всъщност е образец
- параметрите са видими само в рамките на $<\text{тяло}>$
- Примери:
 - `id = \x -> x`
 - `const = \x y -> x`
 - $(\lambda x -> 2 * x + 1) 3 \longrightarrow 7$

λ-функции

- $\lambda \{ <\text{параметър} > \}^+ \rightarrow <\text{тяло}>$
- $\lambda <\text{параметър}_1> \dots <\text{параметър}_n> \rightarrow <\text{тяло}>$
- анонимна функция с n параметъра
- всеки $<\text{параметър}_i>$ всъщност е образец
- параметрите са видими само в рамките на $<\text{тяло}>$
- Примери:
 - `id = \x -> x`
 - `const = \x y -> x`
 - $(\lambda x -> 2 * x + 1) 3 \longrightarrow 7$
 - $(\lambda x l -> l ++ [x]) 4 [1..3] \longrightarrow [1,2,3,4]$

λ -функции

- $\lambda \{ <\text{параметър} > \}^+ \rightarrow <\text{тяло}>$
- $\lambda <\text{параметър}_1> \dots <\text{параметър}_n> \rightarrow <\text{тяло}>$
- анонимна функция с n параметъра
- всеки $<\text{параметър}_i>$ всъщност е образец
- параметрите са видими само в рамките на $<\text{тяло}>$
- Примери:
 - `id = \x -> x`
 - `const = \x y -> x`
 - $(\lambda x -> 2 * x + 1) 3 \longrightarrow 7$
 - $(\lambda x 1 -> 1 ++ [x]) 4 [1..3] \longrightarrow [1,2,3,4]$
 - $(\lambda(x,y) -> x^2 + y) (3,5) \longrightarrow 14$

λ -функции

- $\lambda \{ <\text{параметър} > \}^+ \rightarrow <\text{тяло}>$
- $\lambda <\text{параметър}_1> \dots <\text{параметър}_n> \rightarrow <\text{тяло}>$
- анонимна функция с n параметъра
- всеки $<\text{параметър}_i>$ всъщност е образец
- параметрите са видими само в рамките на $<\text{тяло}>$
- Примери:
 - `id = \x -> x`
 - `const = \x y -> x`
 - $(\lambda x -> 2 * x + 1) 3 \longrightarrow 7$
 - $(\lambda x 1 -> 1 ++ [x]) 4 [1..3] \longrightarrow [1,2,3,4]$
 - $(\lambda(x,y) -> x^2 + y) (3,5) \longrightarrow 14$
 - $(\lambda f x -> f (f x)) (*3) 4 \longrightarrow 36$

λ-функции

- $\lambda \{ <\text{параметър} > \}^+ \rightarrow <\text{тяло}>$
- $\lambda <\text{параметър}_1> \dots <\text{параметър}_n> \rightarrow <\text{тяло}>$
- анонимна функция с n параметъра
- всеки $<\text{параметър}_i>$ всъщност е образец
- параметрите са видими само в рамките на $<\text{тяло}>$
- Примери:
 - `id = \x -> x`
 - `const = \x y -> x`
 - $(\lambda x -> 2 * x + 1) 3 \longrightarrow 7$
 - $(\lambda x 1 -> 1 ++ [x]) 4 [1..3] \longrightarrow [1,2,3,4]$
 - $(\lambda (x,y) -> x^2 + y) (3,5) \longrightarrow 14$
 - $(\lambda f x -> f (f x)) (*3) 4 \longrightarrow 36$
- отсичането на операции може да се изрази чрез λ-функции:

λ -функции

- $\lambda \{ <\text{параметър} > \}^+ \rightarrow <\text{тяло}>$
- $\lambda <\text{параметър}_1> \dots <\text{параметър}_n> \rightarrow <\text{тяло}>$
- анонимна функция с n параметъра
- всеки $<\text{параметър}_i>$ всъщност е образец
- параметрите са видими само в рамките на $<\text{тяло}>$
- Примери:
 - `id = \x -> x`
 - `const = \x y -> x`
 - $(\lambda x -> 2 * x + 1) 3 \longrightarrow 7$
 - $(\lambda x 1 -> 1 ++ [x]) 4 [1..3] \longrightarrow [1,2,3,4]$
 - $(\lambda (x,y) -> x^2 + y) (3,5) \longrightarrow 14$
 - $(\lambda f x -> f (f x)) (*3) 4 \longrightarrow 36$
- отсичането на операции може да се изрази чрез λ -функции:
 - $(<\text{операция}> <\text{израз}>) = \lambda x -> x <\text{операция}> <\text{израз}>$

λ -функции

- $\lambda \{ <\text{параметър} > \}^+ \rightarrow <\text{тяло}>$
- $\lambda <\text{параметър}_1> \dots <\text{параметър}_n> \rightarrow <\text{тяло}>$
- анонимна функция с n параметъра
- всеки $<\text{параметър}_i>$ всъщност е образец
- параметрите са видими само в рамките на $<\text{тяло}>$
- Примери:
 - `id = \x -> x`
 - `const = \x y -> x`
 - $(\lambda x -> 2 * x + 1) 3 \longrightarrow 7$
 - $(\lambda x 1 -> 1 ++ [x]) 4 [1..3] \longrightarrow [1,2,3,4]$
 - $(\lambda (x,y) -> x^2 + y) (3,5) \longrightarrow 14$
 - $(\lambda f x -> f (f x)) (*3) 4 \longrightarrow 36$
- отсичането на операции може да се изрази чрез λ -функции:
 - $(<\text{операция}> <\text{израз}>) = \lambda x -> x <\text{операция}> <\text{израз}>$
 - $(<\text{израз}> <\text{операция}>) = \lambda x -> <\text{израз}> <\text{операция}> x$

Свойства на λ -функциите

- $\lambda x_1 x_2 \dots x_n \rightarrow <\text{тяло}>$
- $\iff \lambda x_1 \rightarrow \lambda x_2 \rightarrow \dots \lambda x_n \rightarrow <\text{тяло}>$

Свойства на λ -функциите

- $\lambda x_1 x_2 \dots x_n \rightarrow <\text{тяло}>$
- $\iff \lambda x_1 \rightarrow (\lambda x_2 \rightarrow \dots (\lambda x_n \rightarrow <\text{тяло}>))$
- $f\ x = <\text{тяло}>$
- $\iff f = \lambda x \rightarrow <\text{тяло}>$

$$\begin{aligned} f(x) &= q \\ f &= \lambda x \{ \end{aligned}$$

Свойства на λ -функциите

- $\lambda x_1 x_2 \dots x_n \rightarrow <\text{тяло}>$
 $\iff \lambda x_1 \rightarrow \lambda x_2 \rightarrow \dots \lambda x_n \rightarrow <\text{тяло}>$
- $f x = <\text{тяло}>$
 $\iff f = \lambda x \rightarrow <\text{тяло}>$
- $f x y = <\text{тяло}>$
 $\iff f x = \lambda y \rightarrow <\text{тяло}>$
 $\iff f = \lambda x y \rightarrow <\text{тяло}>$

Свойства на λ -функциите

- $\lambda x_1 x_2 \dots x_n \rightarrow <\text{тяло}>$
 $\iff \lambda x_1 \rightarrow \lambda x_2 \rightarrow \dots \lambda x_n \rightarrow <\text{тяло}>$
- $f x = <\text{тяло}>$
 $\iff f = \lambda x \rightarrow <\text{тяло}>$
- $f x y = <\text{тяло}>$
 $\iff f x = \lambda y \rightarrow <\text{тяло}>$
 $\iff f = \lambda x y \rightarrow <\text{тяло}>$
- $f x_1 \dots x_n = <\text{тяло}>$
 $\iff f x_1 \dots x_{n-1} = \lambda x_n \rightarrow <\text{тяло}>$
 $\iff \dots$
 $\iff f = \lambda x_1 \dots x_n \rightarrow <\text{тяло}>$

Свойства на λ -функциите

- $\lambda x_1 x_2 \dots x_n \rightarrow <\text{тяло}>$
- $\iff \lambda x_1 \rightarrow \lambda x_2 \rightarrow \dots \lambda x_n \rightarrow <\text{тяло}>$
- $f x = <\text{тяло}>$
- $\iff f = \lambda x \rightarrow <\text{тяло}>$
- $f x y = <\text{тяло}>$
- $\iff f x = \lambda y \rightarrow <\text{тяло}>$
- $\iff f = \lambda x y \rightarrow <\text{тяло}>$
- $f x_1 \dots x_n = <\text{тяло}>$
- $\iff f x_1 \dots x_{n-1} = \lambda x_n \rightarrow <\text{тяло}>$
- $\iff \dots$
- $\iff f = \lambda x_1 \dots x_n \rightarrow <\text{тяло}>$
- $\lambda x y \rightarrow f x y$
- $\iff \cancel{\lambda x} \rightarrow f \cancel{x y}$
- $\iff f$

$(\lambda x \lambda y f) x y$

$(\lambda x (\lambda y (f x y)))$

Трансформация (map)

- `map` :: $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$

Трансформация (map)

- `map` :: $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$
- `map f l = [f x | x <- l]`

Трансформация (map)

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

Трансформация (map)

- `map :: (a -> b) -> [a] -> [b]`
- `map f l = [f x | x <- l]`
- `map _ [] = []`
- `map f (x:xs) = f x : map f xs`
- Примери:

Трансформация (map)

- `map` :: $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$
- `map f l = [f x | x <- l]`
- `map _ [] = []`
- `map f (x:xs) = f x : map f xs`
- Примери:
 - `map (^2) [1,2,3] → ?`

Трансформация (map)

- `map :: (a -> b) -> [a] -> [b]`
- `map f l = [f x | x <- l]`
- `map _ [] = []`
- `map f (x:xs) = f x : map f xs`
- Примери:
 - `map (^2) [1,2,3] —> [1,4,9]`

Трансформация (map)

- `map :: (a -> b) -> [a] -> [b]`
- `map f l = [f x | x <- l]`
- `map _ [] = []`
- `map f (x:xs) = f x : map f xs`
- Примери:
 - `map (^2) [1,2,3] —> [1,4,9]`
 - `map (!!1) [[1,2,3],[4,5,6],[7,8,9]] —> ?`

Трансформация (map)

- `map :: (a -> b) -> [a] -> [b]`
- `map f l = [f x | x <- l]`
- `map _ [] = []`
- `map f (x:xs) = f x : map f xs`
- Примери:
 - `map (^2) [1,2,3] → [1,4,9]`
 - `map (!!1) [[1,2,3],[4,5,6],[7,8,9]] → [2,5,8]`

Трансформация (map)

- `map :: (a -> b) -> [a] -> [b]`
- `map f l = [f x | x <- l]`
- `map _ [] = []`
- `map f (x:xs) = f x : map f xs`
- Примери:
 - `map (^2) [1,2,3] → [1,4,9]`
 - `map (!!1) [[1,2,3],[4,5,6],[7,8,9]] → [2,5,8]`
 - `map sum [[1,2,3],[4,5,6],[7,8,9]] → ?`

Трансформация (map)

- `map :: (a -> b) -> [a] -> [b]`
- `map f l = [f x | x <- l]`
- `map _ [] = []`
- `map f (x:xs) = f x : map f xs`
- Примери:
 - `map (^2) [1,2,3] → [1,4,9]`
 - `map (!!1) [[1,2,3],[4,5,6],[7,8,9]] → [2,5,8]`
 - `map sum [[1,2,3],[4,5,6],[7,8,9]] → [6,15,24]`

Трансформация (map)

- `map :: (a -> b) -> [a] -> [b]`
- `map f l = [f x | x <- l]`
- `map _ [] = []`
- `map f (x:xs) = f x : map f xs`
- Примери:
 - `map (^2) [1,2,3] → [1,4,9]`
 - `map (!!1) [[1,2,3],[4,5,6],[7,8,9]] → [2,5,8]`
 - `map sum [[1,2,3],[4,5,6],[7,8,9]] → [6,15,24]`
 - `map ("a"++) ["cat","dog","pig"] → ?`

Трансформация (map)

- `map :: (a -> b) -> [a] -> [b]`
- `map f l = [f x | x <- l]`
- `map _ [] = []`
- `map f (x:xs) = f x : map f xs`
- Примери:

- `map (^2) [1,2,3] → [1,4,9]`
- `map (!!1) [[1,2,3],[4,5,6],[7,8,9]] → [2,5,8]`
- `map sum [[1,2,3],[4,5,6],[7,8,9]] → [6,15,24]`
- `map ("a"++) ["cat","dog","pig"]
→ ["a cat","a dog","a pig"]`

Трансформация (map)

- `map :: (a -> b) -> [a] -> [b]`
- `map f l = [f x | x <- l]`
- `map _ [] = []`
- `map f (x:xs) = f x : map f xs`
- Примери:

- `map (^2) [1,2,3] → [1,4,9]`
- `map (!!1) [[1,2,3],[4,5,6],[7,8,9]] → [2,5,8]`
- `map sum [[1,2,3],[4,5,6],[7,8,9]] → [6,15,24]`
- `map ("a "++) ["cat","dog","pig"]
→ ["a cat","a dog","a pig"]`
- `map (\f -> f 2) [(^2),(1+),(*3)] → ?`

Трансформация (map)

- `map :: (a -> b) -> [a] -> [b]`
- `map f l = [f x | x <- l]`
- `map _ [] = []`
- `map f (x:xs) = f x : map f xs`
- Примери:

- `map (^2) [1,2,3] → [1,4,9]`
- `map (!!1) [[1,2,3],[4,5,6],[7,8,9]] → [2,5,8]`
- `map sum [[1,2,3],[4,5,6],[7,8,9]] → [6,15,24]`
- `map ("a "++) ["cat","dog","pig"]
→ ["a cat","a dog","a pig"]`
- `map (\f -> f 2) [(^2),(1+),(*3)] → [4,3,6]`

Филтриране (filter)

- `filter :: (a -> Bool) -> [a] -> [a]`

Филтриране (filter)

- `filter :: (a -> Bool) -> [a] -> [a]`
- `filter p l = [x | x <- l, p x]`

Филтриране (filter)

- `filter :: (a -> Bool) -> [a] -> [a]`
- `filter p l = [x | x <- l, p x]`
- `filter _ [] = []`
`filter p (x:xs)`
| `p x` = `x : rest`
| `otherwise` = `rest`
`where rest = filter p xs`

Филтриране (filter)

- `filter :: (a -> Bool) -> [a] -> [a]`
- `filter p l = [x | x <- l, p x]`
- `filter _ [] = []`
`filter p (x:xs)`
 | `p x` = `x : rest`
 | `otherwise` = `rest`
`where rest = filter p xs`

- Примери:

Филтриране (filter)

- `filter :: (a -> Bool) -> [a] -> [a]`
- `filter p l = [x | x <- l, p x]`
- `filter _ [] = []`
`filter p (x:xs)`
| `p x` = `x : rest`
| `otherwise` = `rest`
`where rest = filter p xs`
- Примери:
 - `filter odd [1..5] → ?`

Филтриране (filter)

- `filter :: (a -> Bool) -> [a] -> [a]`
- `filter p l = [x | x <- l, p x]`
- `filter _ [] = []`
`filter p (x:xs)`
| `p x` = `x : rest`
| `otherwise` = `rest`
`where rest = filter p xs`
- Примери:
 - `filter odd [1..5] → [1,3,5]`

Филтриране (filter)

- `filter :: (a -> Bool) -> [a] -> [a]`
- `filter p l = [x | x <- l, p x]`
- `filter _ [] = []`
`filter p (x:xs)`
| `p x` = `x : rest`
| `otherwise` = `rest`
`where rest = filter p xs`

- Примери:

- `filter odd [1..5] → [1,3,5]`
- `filter (\f -> f 2 > 3) [(^2),(+1),(*3)] → ?`

Филтриране (filter)

- `filter :: (a -> Bool) -> [a] -> [a]`
- `filter p l = [x | x <- l, p x]`
- `filter _ [] = []`
`filter p (x:xs)`
| `p x` = `x : rest`
| `otherwise` = `rest`
`where rest = filter p xs`

• Примери:

- `filter odd [1..5] → [1,3,5]`
- `filter (\f -> f 2 > 3) [(^2),(+1),(*3)] → [(^2),(*3)]`

Филтриране (filter)

- `filter :: (a -> Bool) -> [a] -> [a]`
- `filter p l = [x | x <- l, p x]`
- `filter _ [] = []`
`filter p (x:xs)`
| `p x` = `x : rest`
| `otherwise` = `rest`
`where rest = filter p xs`

- Примери:

- `filter odd [1..5] → [1,3,5]`
- `filter (\f -> f 2 > 3) [(^2),(+1),(*3)] → [(^2),(*3)]`
- `map (filter even) [[1,2,3],[4,5,6],[7,8,9]]`
→ ?

Филтриране (filter)

- `filter :: (a -> Bool) -> [a] -> [a]`
- `filter p l = [x | x <- l, p x]`
- `filter _ [] = []`
`filter p (x:xs)`
| `p x` = `x : rest`
| `otherwise` = `rest`
`where rest = filter p xs`

- Примери:

- `filter odd [1..5] → [1,3,5]`
- `filter (\f -> f 2 > 3) [(^2),(+1),(*3)] → [(^2),(*3)]`
- `map (filter even) [[1,2,3],[4,5,6],[7,8,9]]`
→ `[[2],[4,6],[8]]`

Филтриране (filter)

- `filter :: (a -> Bool) -> [a] -> [a]`
- `filter p l = [x | x <- l, p x]`
- `filter _ [] = []`
`filter p (x:xs)`
`| p x = x : rest`
`| otherwise = rest`
`where rest = filter p xs`

- Примери:

- `filter odd [1..5] → [1,3,5]`
- `filter (\f -> f 2 > 3) [(^2),(+1),(*3)] → [(^2),(*3)]`
- `map (filter even) [[1,2,3],[4,5,6],[7,8,9]]`
`→ [[2],[4,6],[8]]`
- `map (\x -> map (\f -> filter f x) [(<0),(==0),(>0)])`
`[[{-2,1,0},[1,4,-1],[0,0,1]]`
`→ ?`

Филтриране (filter)

- `filter :: (a -> Bool) -> [a] -> [a]`
- `filter p l = [x | x <- l, p x]`
- `filter _ [] = []`
`filter p (x:xs)`
`| p x = x : rest`
`| otherwise = rest`
`where rest = filter p xs`

• Примери:

- `filter odd [1..5] → [1,3,5]`
- `filter (\f -> f 2 > 3) [(^2),(+1),(*3)] → [(^2),(*3)]`
- `map (filter even) [[1,2,3],[4,5,6],[7,8,9]]`
`→ [[2],[4,6],[8]]`
- `map (\x -> map (\f -> filter f x) [(<0),(==0),(>0)])`
`[[[-2,1,0],[1,4,-1],[0,0,1]]`
`→ [[[[-2],[0],[1]],[[-1],[],[1,4]]],[],[0,0],[1]]]`

Отделяне на списъци с `map` и `filter`

Отделянето на списъци е синтактична захар за `map` и `filter`

Отделяне на списъци с `map` и `filter`

Отделянето на списъци е синтактична захар за `map` и `filter`

- [`<израз> | <образец> <- <списък>, <условие>`]
↔
`map (\<образец> -> <израз>)`
`(filter (\<образец> -> <условие>) <списък>)`

Отделяне на списъци с `map` и `filter`

Отделянето на списъци е синтактична захар за `map` и `filter`

- [*<израз> | <образец> <- <списък>, <условие>*]
↔
`map (\<образец> -> <израз>)`
`(filter (\<образец> -> <условие>) <списък>)`
- [*<образец> | <образец> <- <списък>, <условие₁>, <условие₂>*]
↔
`filter (\<образец> -> <условие2>)`
`(filter (\<образец> -> <условие1>) <списък>)`

Отделяне на списъци с `map` и `filter`

Отделянето на списъци е синтактична захар за `map` и `filter`

- [`<израз>` | `<образец>` \leftarrow `<списък>`, `<условие>`]
 \leftrightarrow

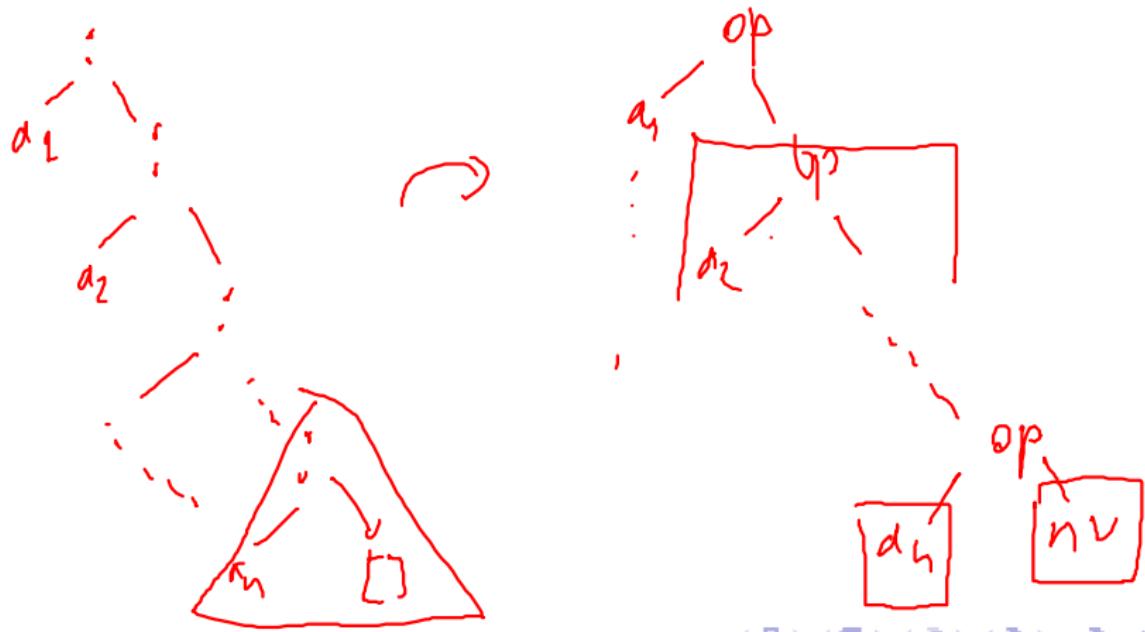
```
map (\<образец> -> <израз>)
     (filter (\<образец> -> <условие>) <списък>)
```
- [`<образец>` | `<образец>` \leftarrow `<списък>`, `<условие1>`, `<условие2>`]
 \leftrightarrow

```
filter (\<образец> -> <условие2>)
       (filter (\<образец> -> <условие1>) <списък>)
```
- [`<израз>` | `<образец1>` \leftarrow `<списък1>`, `<образец2>` \leftarrow `<списък2>`]
 \leftrightarrow

```
concat (map (\<образец1> ->
              map (\<образец2> -> <израз>) <списък2>)
                   <списък1>)
```

Дясно свиване (foldr)

- **foldr** :: $(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$



Дясно свиване (foldr)

- `foldr` :: $(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
- $\text{foldr } op \text{ nv } [x_1, x_2, \dots, x_n] =$
 $x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_n \text{ 'op' } nv) \dots)$

Дясно свиване (`foldr`)

- `foldr` :: $(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
- $\text{foldr } op \text{ nv } [x_1, x_2, \dots, x_n] = x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_n \text{ 'op' } nv) \dots)$
- $\text{foldr } _- \text{ nv } [] = nv$
 $\text{foldr } op \text{ nv } (x:xs) = x \text{ 'op' } \text{foldr } op \text{ nv } xs$

Дясно свиване (`foldr`)

- `foldr` :: $(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
- $\text{foldr } op \text{ nv } [x_1, x_2, \dots, x_n] = x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_n \text{ 'op' } nv) \dots)$
- $\text{foldr } _- \text{ nv } [] = nv$
 $\text{foldr } op \text{ nv } (x:xs) = x \text{ 'op' } \text{foldr } op \text{ nv } xs$
- Примери:

Дясно свиване (foldr)

- `foldr` :: $(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
- $\text{foldr } op \text{ nv } [x_1, x_2, \dots, x_n] = x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_n \text{ 'op' } nv) \dots)$
- $\text{foldr } _- \text{ nv } [] = nv$
 $\text{foldr } op \text{ nv } (x:xs) = x \text{ 'op' } \text{foldr } op \text{ nv } xs$
- Примери:
 - `sum = foldr (+) 0`

(if px then f x else fx)
 ↓
 (if px then f else g) x

Дясно свиване (`foldr`)

- `foldr` :: $(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
- $\text{foldr } op \text{ nv } [x_1, x_2, \dots, x_n] = x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_n \text{ 'op' } nv) \dots)$
- $\text{foldr } _- \text{ nv } [] = nv$
 $\text{foldr } op \text{ nv } (x:xs) = x \text{ 'op' } \text{foldr } op \text{ nv } xs$
- Примери:
 - `sum = foldr (+) 0`
 - `product = foldr (*) 1`

Дясно свиване (`foldr`)

- `foldr` :: $(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$
- $\text{foldr } op \text{ nv } [x_1, x_2, \dots, x_n] = x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_n \text{ 'op' } nv) \dots)$
- $\text{foldr } _- \text{ nv } [] = nv$
 $\text{foldr } op \text{ nv } (x:xs) = x \text{ 'op' } \text{foldr } op \text{ nv } xs$
- Примери:
 - `sum = foldr (+) 0`
 - `product = foldr (*) 1`
 - `concat = foldr (++) []`

Дясно свиване (foldr)

- `foldr :: (a -> b -> b) -> b -> [a] -> b`
- $$\text{foldr } op \text{ nv } [x_1, x_2, \dots, x_n] = x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_n \text{ 'op' } nv) \dots)$$
- $$\text{foldr } _- \text{ nv } [] = \text{nv}$$

$$\text{foldr } op \text{ nv } (x:xs) = x \text{ 'op' } \text{foldr } op \text{ nv } xs$$
- **Примери:**
 - `sum = foldr (+) 0`
 - `product = foldr (*) 1`
 - `concat = foldr (++) []`
 - `and = foldr (&&) True`

Дясно свиване (foldr)

- `foldr :: (a -> b -> b) -> b -> [a] -> b`
- $$\text{foldr } op \text{ nv } [x_1, x_2, \dots, x_n] = x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_n \text{ 'op' } nv) \dots)$$
- $$\text{foldr } _- \text{ nv } [] = \text{nv}$$

$$\text{foldr } op \text{ nv } (x:xs) = x \text{ 'op' } \text{foldr } op \text{ nv } xs$$
- **Примери:**
 - `sum = foldr (+) 0`
 - `product = foldr (*) 1`
 - `concat = foldr (++) []`
 - `and = foldr (&&) True`
 - `or = foldr (||) False`

Дясно свиване (foldr)

- `foldr :: (a -> b -> b) -> b -> [a] -> b`
- `foldr op nv [x1, x2, ..., xn] =`
 $x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_n \text{ 'op' } nv) \dots)$
- `foldr _ nv [] = nv`
`foldr op nv (x:xs) = x 'op' foldr op nv xs`
- **Примери:**
 - `sum = foldr (+) 0`
 - `product = foldr (*) 1`
 - `concat = foldr (++) []`
 - `and = foldr (&&) True`
 - `or = foldr (||) False`
 - `map f = foldr (\x r -> f x : r) []`

Дясно свиване (foldr)

- `foldr :: (a -> b -> b) -> b -> [a] -> b`
- $$\text{foldr } op \text{ nv } [x_1, x_2, \dots, x_n] = x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_n \text{ 'op' } nv) \dots)$$
- $$\text{foldr } _- \text{ nv } [] = \text{nv}$$

$$\text{foldr } op \text{ nv } (x:xs) = x \text{ 'op' } \text{foldr } op \text{ nv } xs$$
- **Примери:**
 - `sum = foldr (+) 0`
 - `product = foldr (*) 1`
 - `concat = foldr (++) []`
 - `and = foldr (&&) True`
 - `or = foldr (||) False`
 - `map f = foldr (\x -> (f x:)) []`

Дясно свиване (foldr)

- `foldr :: (a -> b -> b) -> b -> [a] -> b`
- `foldr op nv [x1, x2, ..., xn] =`
 $x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_n \text{ 'op' } nv) \dots)$
- `foldr _ nv [] = nv`
`foldr op nv (x:xs) = x 'op' foldr op nv xs`
- **Примери:**
 - `sum = foldr (+) 0`
 - `product = foldr (*) 1`
 - `concat = foldr (++) []`
 - `and = foldr (&&) True`
 - `or = foldr (||) False`
 - `map f = foldr (\x -> (f x:)) []`
 - `filter p = foldr (\x r -> if p x then x : r else r) []`

Дясно свиване (`foldr`)

- `foldr :: (a -> b -> b) -> b -> [a] -> b`
- `foldr op nv [x1, x2, ..., xn] =`
 $x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_n \text{ 'op' } nv) \dots)$
- `foldr _ nv [] = nv`
 $\text{foldr op nv (x:xs)} = x \text{ 'op' } \text{foldr op nv xs}$
- **Примери:**
 - `sum = foldr (+) 0`
 - `product = foldr (*) 1`
 - `concat = foldr (++) []`
 - `and = foldr (&&) True`
 - `or = foldr (||) False`
 - `map f = foldr (\x -> (f x:)) []`
 - `filter p = foldr (\x r -> (if p x then (x:) else id) r)`
 $[\dots]$

Дясно свиване (foldr)

- `foldr :: (a -> b -> b) -> b -> [a] -> b`
- `foldr op nv [x1, x2, ..., xn] =`
 $x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_n \text{ 'op' } nv) \dots)$
- `foldr _ nv [] = nv`
`foldr op nv (x:xs) = x 'op' foldr op nv xs`
- **Примери:**
 - `sum = foldr (+) 0`
 - `product = foldr (*) 1`
 - `concat = foldr (++) []`
 - `and = foldr (&&) True`
 - `or = foldr (||) False`
 - `map f = foldr (\x -> (f x:)) []`
 - `filter p = foldr (\x -> if p x then (x:) else id) []`

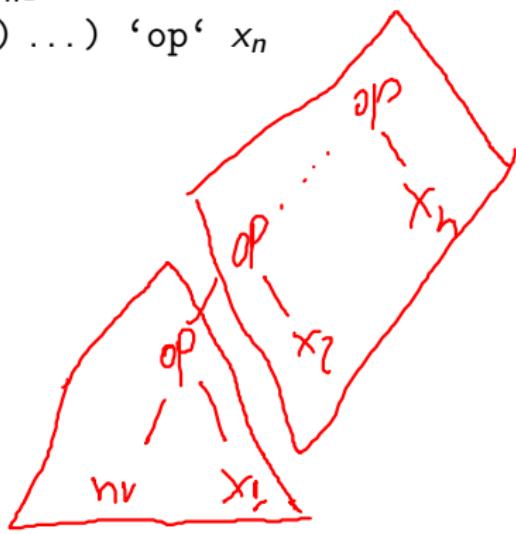
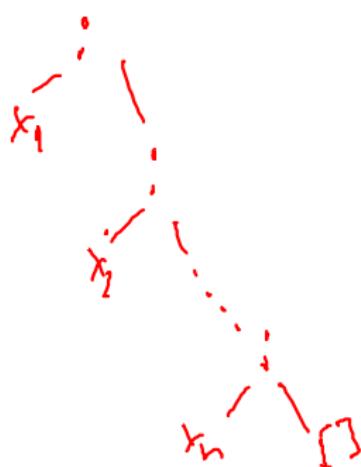
Ляво свиване (foldl)

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

Ляво свиване (foldl)

- `foldl :: (b -> a -> b) -> b -> [a] -> b`
- `foldl op nv [x1, x2, ..., xn] =`

$$(\dots ((nv \text{ `op`} x_1) \text{ `op`} x_2) \dots) \text{ `op`} x_n$$



Ляво свиване (foldl)

- `foldl :: (b -> a -> b) -> b -> [a] -> b`
- `foldl op nv [x1, x2, ..., xn] =`
`(...((nv `op` x1) `op` x2) ...) `op` xn`
- `foldl _ nv [] = nv`
`foldl op nv (x:xs) = foldl op (nv `op` x) xs`

Ляво свиване (foldl)

- `foldl :: (b -> a -> b) -> b -> [a] -> b`
- `foldl op nv [x1, x2, ..., xn] = (...((nv `op` x1) `op` x2) ...) `op` xn`
- `foldl _ nv [] = nv`
`foldl op nv (x:xs) = foldl op (nv `op` x) xs`
- Пример:
 - `flip f x y = f y x`
 - `reverse = foldl (flip (:)) []`

Сиване на непразни списъци (foldr1 и foldl1)

- `foldr1 :: (a -> a -> a) -> [a] -> a`

Сиване на непразни списъци (foldr1 и foldl1)

- **foldr1** :: ($a \rightarrow a \rightarrow a$) $\rightarrow [a] \rightarrow a$
- **foldr1** op nv $[x_1, x_2, \dots, x_n] =$
 $x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_{n-1} \text{ 'op' } x_n) \dots)$

Сиване на непразни списъци (`foldr1` и `foldl1`)

- `foldr1 :: (a -> a -> a) -> [a] -> a`
- `foldr1 op nv [x1, x2, ..., xn] =`
 $x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_{n-1} \text{ 'op' } x_n) \dots)$
- `foldr1 _ [x] = x`
`foldr1 op (x:xs) = x 'op' foldr1 op xs`

Сиване на непразни списъци (`foldr1` и `foldl1`)

- `foldr1 :: (a -> a -> a) -> [a] -> a`
- `foldr1 op nv [x1, x2, ..., xn] =`
 $x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_{n-1} \text{ 'op' } x_n) \dots)$
- `foldr1 _ [x] = x`
`foldr1 op (x:xs) = x 'op' foldr1 op xs`
- `foldl1 :: (a -> a -> a) -> [a] -> a`

Сиване на непразни списъци (`foldr1` и `foldl1`)

- `foldr1` :: $(a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$
- $\text{foldr1 } op \text{ nv } [x_1, x_2, \dots, x_n] = x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_{n-1} \text{ 'op' } x_n) \dots)$
- $\text{foldr1 } _- \text{ [x]} = x$
 $\text{foldr1 } op \text{ (x:xs)} = x \text{ 'op' } \text{foldr1 } op \text{ xs}$
- `foldl1` :: $(a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$
- $\text{foldl1 } op \text{ nv } [x_1, x_2, \dots, x_n] = (\dots ((x_1 \text{ 'op' } x_2) \dots) \text{ 'op' } x_n$

Сиване на непразни списъци (`foldr1` и `foldl1`)

- `foldr1` :: $(a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$
- $\text{foldr1 } op \text{ nv } [x_1, x_2, \dots, x_n] = x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_{n-1} \text{ 'op' } x_n) \dots)$
- $\text{foldr1 } _- \text{ [x]} = x$
 $\text{foldr1 } op \text{ (x:xs)} = x \text{ 'op' } \text{foldr1 } op \text{ xs}$
- `foldl1` :: $(a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$
- $\text{foldl1 } op \text{ nv } [x_1, x_2, \dots, x_n] = (\dots((x_1 \text{ 'op' } x_2) \dots) \text{ 'op' } x_n$
- $\text{foldl1 } op \text{ (x:xs)} = \text{foldl } op \text{ x xs}$

Сиване на непразни списъци (`foldr1` и `foldl1`)

- `foldr1` :: $(a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$
- $\text{foldr1 } op \text{ nv } [x_1, x_2, \dots, x_n] = x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_{n-1} \text{ 'op' } x_n) \dots)$
- $\text{foldr1 } _- \text{ [x]} = x$
 $\text{foldr1 } op \text{ (x:xs)} = x \text{ 'op' } \text{foldr1 } op \text{ xs}$
- `foldl1` :: $(a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$
- $\text{foldl1 } op \text{ nv } [x_1, x_2, \dots, x_n] = (\dots((x_1 \text{ 'op' } x_2) \dots) \text{ 'op' } x_n$
- $\text{foldl1 } op \text{ (x:xs)} = \text{foldl } op \text{ x xs}$
- Примери:

Сиване на непразни списъци (`foldr1` и `foldl1`)

- `foldr1` :: $(a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$
- $\text{foldr1 } op \text{ nv } [x_1, x_2, \dots, x_n] = x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_{n-1} \text{ 'op' } x_n) \dots)$
- $\text{foldr1 } _- \text{ [x]} = x$
 $\text{foldr1 } op \text{ (x:xs)} = x \text{ 'op' } \text{foldr1 } op \text{ xs}$
- `foldl1` :: $(a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$
- $\text{foldl1 } op \text{ nv } [x_1, x_2, \dots, x_n] = (\dots((x_1 \text{ 'op' } x_2) \dots) \text{ 'op' } x_n$
- $\text{foldl1 } op \text{ (x:xs)} = \text{foldl } op \text{ x xs}$
- Примери:
 - `maximum = foldr1 max`

Сиване на непразни списъци (`foldr1` и `foldl1`)

- `foldr1` :: $(a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$
- $\text{foldr1 } op \text{ nv } [x_1, x_2, \dots, x_n] = x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_{n-1} \text{ 'op' } x_n) \dots)$
- $\text{foldr1 } _- \text{ [x]} = x$
 $\text{foldr1 } op \text{ (x:xs)} = x \text{ 'op' } \text{foldr1 } op \text{ xs}$
- `foldl1` :: $(a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$
- $\text{foldl1 } op \text{ nv } [x_1, x_2, \dots, x_n] = (\dots((x_1 \text{ 'op' } x_2) \dots) \text{ 'op' } x_n$
- $\text{foldl1 } op \text{ (x:xs)} = \text{foldl } op \text{ x xs}$
- Примери:
 - `maximum` = `foldr1 max`
 - `minimum` = `foldr1 min`

Свиване на непразни списъци (`foldr1` и `foldl1`)

- `foldr1` :: $(a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$
- $\text{foldr1 } op \text{ nv } [x_1, x_2, \dots, x_n] = x_1 \text{ 'op' } (x_2 \text{ 'op' } \dots (x_{n-1} \text{ 'op' } x_n) \dots)$
- $\text{foldr1 } _- [x] = x$
 $\text{foldr1 } op (x:xs) = x \text{ 'op' } \text{foldr1 } op xs$
- `foldl1` :: $(a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$
- $\text{foldl1 } op \text{ nv } [x_1, x_2, \dots, x_n] = (\dots((x_1 \text{ 'op' } x_2) \dots) \text{ 'op' } x_n$
- $\text{foldl1 } op (x:xs) = \text{foldl } op x xs$
- Примери:
 - `maximum` = `foldr1 max`
 - `minimum` = `foldr1 min`
 - `last` = `foldl1 (_ x -> x)`

Сканиране на списъци (`scanl`, `scanr`)

`scanr` и `scanl` връщат историята на пресмятането на `foldr` и `foldl`

Сканиране на списъци (`scanl`, `scanr`)

`scanr` и `scanl` връщат историята на пресмятането на `foldr` и `foldl`

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

Сканиране на списъци (`scanl`, `scanr`)

`scanr` и `scanl` връщат историята на пресмятането на `foldr` и `foldl`

- `scanr :: (a -> b -> b) -> b -> [a] -> [b]`
- `scanr op nv [x1, x2, ..., xn] =`
$$[x_1 \text{ `op' } (x_2 \text{ `op' } \dots (x_n \text{ `op' } nv) \dots),$$
$$x_2 \text{ `op' } (\dots (x_n \text{ `op' } nv) \dots),$$
$$\dots$$
$$x_n \text{ `op' } nv,$$
$$nv]$$

Сканиране на списъци (`scanl`, `scanr`)

`scanr` и `scanl` връщат историята на пресмятането на `foldr` и `foldl`

- `scanr :: (a -> b -> b) -> b -> [a] -> [b]`
- `scanr op nv [x1, x2, ..., xn] =`
$$[x_1 \text{ `op' } (x_2 \text{ `op' } \dots (x_n \text{ `op' } nv) \dots),$$
$$x_2 \text{ `op' } (\dots (x_n \text{ `op' } nv) \dots),$$
$$\dots$$
$$x_n \text{ `op' } nv,$$
$$nv]$$
- `scanl :: (b -> a -> b) -> b -> [a] -> [b]`

Сканиране на списъци (`scanl`, `scanr`)

`scanr` и `scanl` връщат историята на пресмятането на `foldr` и `foldl`

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

- `scanr op nv [x1, x2, ..., xn] =`
 $[x_1 \text{ `op' } (x_2 \text{ `op' } \dots (x_n \text{ `op' } nv) \dots),$
 $x_2 \text{ `op' } (\dots (x_n \text{ `op' } nv) \dots),$
 \dots
 $x_n \text{ `op' } nv,$
 $nv]$

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

- `scanl op nv [x1, x2, ..., xn] =`
 $[nv,$
 $nv \text{ `op' } x_1,$
 $(nv \text{ `op' } x_1) \text{ `op' } x_2,$
 \dots
 $(\dots((nv \text{ `op' } x_1) \text{ `op' } x_2) \dots) \text{ `op' } x_n]$

Съшиване на списъци (`zip`, `zipWith`)

- `zip` :: `[a] -> [b] -> [(a,b)]`

Съшиване на списъци (`zip`, `zipWith`)

- `zip` :: $[a] \rightarrow [b] \rightarrow [(a, b)]$
- `zip` $[x_1, x_2, \dots, x_n]$ $[y_1, y_2, \dots, y_n] \longrightarrow [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$

Съшиване на списъци (`zip`, `zipWith`)

- `zip` :: $[a] \rightarrow [b] \rightarrow [(a, b)]$
 - $[x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \longrightarrow [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$
 - ако единият списък е по-къс, спира когато свърши той

Съшиване на списъци (`zip`, `zipWith`)

- `zip` :: $[a] \rightarrow [b] \rightarrow [(a, b)]$
 - $[x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \longrightarrow [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$
 - ако единият списък е по-къс, спира когато свърши той
- `unzip` :: $[(a, b)] \rightarrow ([a], [b])$

Съшиване на списъци (`zip`, `zipWith`)

- `zip` :: $[a] \rightarrow [b] \rightarrow [(a, b)]$
 - `zip` $[x_1, x_2, \dots, x_n]$ $[y_1, y_2, \dots, y_n] \longrightarrow [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$
 - ако единият списък е по-къс, спира когато свърши той
- `unzip` :: $[(a, b)] \rightarrow ([a], [b])$
 - разделя списък от двойки на два списъка с равна дължина

Съшиване на списъци (`zip`, `zipWith`)

- `zip` :: $[a] \rightarrow [b] \rightarrow [(a,b)]$
 - `zip` $[x_1, x_2, \dots, x_n]$ $[y_1, y_2, \dots, y_n] \longrightarrow [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$
 - ако единият списък е по-къс, спира когато свърши той
- `unzip` :: $[(a,b)] \rightarrow ([a], [b])$
 - разделя списък от двойки на два списъка с равна дължина
 - `unzip` $[(x_1,y_1), (x_2,y_2), \dots, (x_n,y_n)] \longrightarrow ([x_1, x_2, \dots, x_n], [y_1, y_2, \dots, y_n])$

Съшиване на списъци (`zip`, `zipWith`)

- `zip` :: $[a] \rightarrow [b] \rightarrow [(a, b)]$
 - `zip` $[x_1, x_2, \dots, x_n]$ $[y_1, y_2, \dots, y_n] \longrightarrow [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$
 - ако единият списък е по-къс, спира когато свърши той
- `unzip` :: $[(a, b)] \rightarrow ([a], [b])$
 - разделя списък от двойки на два списъка с равна дължина
 - `unzip` $[(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)] \longrightarrow ([x_1, x_2, \dots, x_n], [y_1, y_2, \dots, y_n])$
- `zipWith` :: $(a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$

Съшиване на списъци (`zip`, `zipWith`)

- `zip` :: $[a] \rightarrow [b] \rightarrow [(a, b)]$
 - `zip` $[x_1, x_2, \dots, x_n]$ $[y_1, y_2, \dots, y_n] \longrightarrow [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$
 - ако единият списък е по-къс, спира когато свърши той
- `unzip` :: $[(a, b)] \rightarrow ([a], [b])$
 - разделя списък от двойки на два списъка с равна дължина
 - `unzip` $[(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)] \longrightarrow ([x_1, x_2, \dots, x_n], [y_1, y_2, \dots, y_n])$
- `zipWith` :: $(a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$
 - съшива два списъка с дадена бинарна операция

Съшиване на списъци (`zip`, `zipWith`)

- **`zip`** :: $[a] \rightarrow [b] \rightarrow [(a, b)]$
 - $\text{zip } [x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \longrightarrow [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$
 - ако единият списък е по-къс, спира когато свърши той
- **`unzip`** :: $[(a, b)] \rightarrow ([a], [b])$
 - разделя списък от двойки на два списъка с равна дължина
 - $\text{unzip } [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)] \longrightarrow ([x_1, x_2, \dots, x_n], [y_1, y_2, \dots, y_n])$
- **`zipWith`** :: $(a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$
 - съшива два списъка с дадена бинарна операция
 - $\text{zipWith op } [x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \longrightarrow [x_1 \text{ 'op' } y_1, x_2 \text{ 'op' } y_2, \dots, x_n \text{ 'op' } y_n]$

Съшиване на списъци (`zip`, `zipWith`)

- `zip` :: $[a] \rightarrow [b] \rightarrow [(a, b)]$
 - `zip` $[x_1, x_2, \dots, x_n]$ $[y_1, y_2, \dots, y_n] \longrightarrow [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$
 - ако единият списък е по-къс, спира когато свърши той
- `unzip` :: $[(a, b)] \rightarrow ([a], [b])$
 - разделя списък от двойки на два списъка с равна дължина
 - `unzip` $[(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)] \longrightarrow ([x_1, x_2, \dots, x_n], [y_1, y_2, \dots, y_n])$
- `zipWith` :: $(a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$
 - съшива два списъка с дадена бинарна операция
 - `zipWith` op $[x_1, x_2, \dots, x_n]$ $[y_1, y_2, \dots, y_n] \longrightarrow [x_1 \text{ 'op'} y_1, x_2 \text{ 'op'} y_2, \dots, x_n \text{ 'op'} y_n]$
- **Примери:**

Съшиване на списъци (`zip`, `zipWith`)

- **`zip`** :: $[a] \rightarrow [b] \rightarrow [(a, b)]$
 - $\text{zip } [x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \longrightarrow [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$
 - ако единият списък е по-къс, спира когато свърши той
- **`unzip`** :: $[(a, b)] \rightarrow ([a], [b])$
 - разделя списък от двойки на два списъка с равна дължина
 - $\text{unzip } [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)] \longrightarrow ([x_1, x_2, \dots, x_n], [y_1, y_2, \dots, y_n])$
- **`zipWith`** :: $(a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$
 - съшива два списъка с дадена бинарна операция
 - $\text{zipWith op } [x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \longrightarrow [x_1 \text{ 'op' } y_1, x_2 \text{ 'op' } y_2, \dots, x_n \text{ 'op' } y_n]$
- **Примери:**
 - $\text{zip } [1..3] [5..10] \longrightarrow [(1, 5), (2, 6), (3, 7)]$

Съшиване на списъци (`zip`, `zipWith`)

- **`zip`** :: $[a] \rightarrow [b] \rightarrow [(a, b)]$
 - $\text{zip } [x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \longrightarrow [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$
 - ако единият списък е по-къс, спира когато свърши той
- **`unzip`** :: $[(a, b)] \rightarrow ([a], [b])$
 - разделя списък от двойки на два списъка с равна дължина
 - $\text{unzip } [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)] \longrightarrow ([x_1, x_2, \dots, x_n], [y_1, y_2, \dots, y_n])$
- **`zipWith`** :: $(a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$
 - съшива два списъка с дадена бинарна операция
 - $\text{zipWith } \text{op } [x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \longrightarrow [x_1 \text{ 'op' } y_1, x_2 \text{ 'op' } y_2, \dots, x_n \text{ 'op' } y_n]$
- **Примери:**
 - $\text{zip } [1..3] [5..10] \longrightarrow [(1, 5), (2, 6), (3, 7)]$
 - $\text{zipWith } (*) [1..3] [5..10] \longrightarrow [5, 12, 21]$

Съшиване на списъци (`zip`, `zipWith`)

- `zip` :: $[a] \rightarrow [b] \rightarrow [(a,b)]$
 - `zip` $[x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \longrightarrow [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$
 - ако единият списък е по-къс, спира когато свърши той
- `unzip` :: $[(a,b)] \rightarrow ([a], [b])$
 - разделя списък от двойки на два списъка с равна дължина
 - `unzip` $[(x_1,y_1), (x_2,y_2), \dots, (x_n,y_n)] \longrightarrow ([x_1, x_2, \dots, x_n], [y_1, y_2, \dots, y_n])$
- `zipWith` :: $(a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$
 - съшива два списъка с дадена бинарна операция
 - `zipWith` $\text{op} [x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \longrightarrow [x_1 \text{ 'op'} y_1, x_2 \text{ 'op'} y_2, \dots, x_n \text{ 'op'} y_n]$
- **Примери:**
 - `zip` $[1..3] [5..10] \longrightarrow [(1,5), (2,6), (3,7)]$
 - `zipWith` $(*) [1..3] [5..10] \longrightarrow [5, 12, 21]$
 - `zip = zipWith (,,)`

Съшиване на списъци (`zip`, `zipWith`)

- **`zip`** :: $[a] \rightarrow [b] \rightarrow [(a,b)]$
 - `zip` $[x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \rightarrow [(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$
 - ако единият списък е по-къс, спира когато свърши той
- **`unzip`** :: $[(a,b)] \rightarrow ([a], [b])$
 - разделя списък от двойки на два списъка с равна дължина
 - `unzip` $[(x_1,y_1), (x_2,y_2), \dots, (x_n,y_n)] \rightarrow ([x_1, x_2, \dots, x_n], [y_1, y_2, \dots, y_n])$
- **`zipWith`** :: $(a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$
 - съшива два списъка с дадена бинарна операция
 - `zipWith` $\text{op} [x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \rightarrow [x_1 \text{ 'op'} y_1, x_2 \text{ 'op'} y_2, \dots, x_n \text{ 'op'} y_n]$
- **Примери:**
 - `zip` $[1..3] [5..10] \rightarrow [(1,5), (2,6), (3,7)]$
 - `zipWith` $(*) [1..3] [5..10] \rightarrow [5, 12, 21]$
 - `zip = zipWith (,,)`
 - `unzip = foldr (\(x,y) (xs,ys) -> (x:xs,y:ys)) ([], [])`

Разбивания на списъци

- `takeWhile :: (a -> Bool) -> [a] -> [a]`

Разбивания на списъци

- `takeWhile :: (a -> Bool) -> [a] -> [a]`
 - връща първите елементи на списъка, за които е вярно условието

Разбивания на списъци

- `takeWhile :: (a -> Bool) -> [a] -> [a]`
 - връща първите елементи на списъка, за които е вярно условието
 - `takeWhile p = foldr (\x r -> if p x then x:r else []) []`

Разбивания на списъци

- `takeWhile :: (a -> Bool) -> [a] -> [a]`
 - връща първите елементи на списъка, за които е вярно условието
 - `takeWhile p = foldr (\x r -> if p x then x:r else []) []`
 - `takeWhile (<0) [-3..3] —> [-3,-2,-1]`

Разбивания на списъци

- `takeWhile :: (a -> Bool) -> [a] -> [a]`
 - връща първите елементи на списъка, за които е вярно условието
 - `takeWhile p = foldr (\x r -> if p x then x:r else []) []`
 - `takeWhile (<0) [-3..3] —> [-3,-2,-1]`
- `dropWhile :: (a -> Bool) -> [a] -> [a]`

Разбивания на списъци

- `takeWhile :: (a -> Bool) -> [a] -> [a]`
 - връща първите елементи на списъка, за които е вярно условието
 - `takeWhile p = foldr (\x r -> if p x then x:r else []) []`
 - `takeWhile (<0) [-3..3] —> [-3,-2,-1]`
- `dropWhile :: (a -> Bool) -> [a] -> [a]`
 - премахва първите елементи на списъка, за които е вярно условието

Разбивания на списъци

- `takeWhile :: (a -> Bool) -> [a] -> [a]`
 - връща първите елементи на списъка, за които е вярно условието
 - `takeWhile p = foldr (\x r -> if p x then x:r else []) []`
 - `takeWhile (<0) [-3..3] —> [-3,-2,-1]`
- `dropWhile :: (a -> Bool) -> [a] -> [a]`
 - премахва първите елементи на списъка, за които е вярно условието
 - `dropWhile (<0) [-3..3] —> [0,1,2,3]`

Разбивания на списъци

- `takeWhile :: (a -> Bool) -> [a] -> [a]`
 - връща първите елементи на списъка, за които е вярно условието
 - `takeWhile p = foldr (\x r -> if p x then x:r else []) []`
 - `takeWhile (<0) [-3..3] —> [-3,-2,-1]`
- `dropWhile :: (a -> Bool) -> [a] -> [a]`
 - премахва първите елементи на списъка, за които е вярно условието
 - `dropWhile (<0) [-3..3] —> [0,1,2,3]`
- `span :: (a -> Bool) -> [a] -> ([a], [a])`

Разбивания на списъци

- `takeWhile :: (a -> Bool) -> [a] -> [a]`
 - връща първите елементи на списъка, за които е вярно условието
 - `takeWhile p = foldr (\x r -> if p x then x:r else []) []`
 - `takeWhile (<0) [-3..3] -> [-3,-2,-1]`
- `dropWhile :: (a -> Bool) -> [a] -> [a]`
 - премахва първите елементи на списъка, за които е вярно условието
 - `dropWhile (<0) [-3..3] -> [0,1,2,3]`
- `span :: (a -> Bool) -> [a] -> ([a], [a])`
 - `span p l = (takeWhile p l, dropWhile p l)`

Разбивания на списъци

- `takeWhile :: (a -> Bool) -> [a] -> [a]`
 - връща първите елементи на списъка, за които е вярно условието
 - `takeWhile p = foldr (\x r -> if p x then x:r else []) []`
 - `takeWhile (<0) [-3..3] -> [-3,-2,-1]`
- `dropWhile :: (a -> Bool) -> [a] -> [a]`
 - премахва първите елементи на списъка, за които е вярно условието
 - `dropWhile (<0) [-3..3] -> [0,1,2,3]`
- `span :: (a -> Bool) -> [a] -> ([a], [a])`
 - `span p l = (takeWhile p l, dropWhile p l)`
 - `span (<0) [-3..3] -> ([-3,-2,-1], [0,1,2,3])`

Разбивания на списъци

- `takeWhile :: (a -> Bool) -> [a] -> [a]`
 - връща първите елементи на списъка, за които е вярно условието
 - `takeWhile p = foldr (\x r -> if p x then x:r else []) []`
 - `takeWhile (<0) [-3..3] -> [-3,-2,-1]`
- `dropWhile :: (a -> Bool) -> [a] -> [a]`
 - премахва първите елементи на списъка, за които е вярно условието
 - `dropWhile (<0) [-3..3] -> [0,1,2,3]`
- `span :: (a -> Bool) -> [a] -> ([a], [a])`
 - `span p l = (takeWhile p l, dropWhile p l)`
 - `span (<0) [-3..3] -> ([-3,-2,-1], [0,1,2,3])`
- `break :: (a -> Bool) -> [a] -> ([a], [a])`

Разбивания на списъци

- `takeWhile :: (a -> Bool) -> [a] -> [a]`
 - връща първите елементи на списъка, за които е вярно условието
 - `takeWhile p = foldr (\x r -> if p x then x:r else []) []`
 - `takeWhile (<0) [-3..3] → [-3,-2,-1]`
- `dropWhile :: (a -> Bool) -> [a] -> [a]`
 - премахва първите елементи на списъка, за които е вярно условието
 - `dropWhile (<0) [-3..3] → [0,1,2,3]`
- `span :: (a -> Bool) -> [a] -> ([a], [a])`
 - `span p l = (takeWhile p l, dropWhile p l)`
 - `span (<0) [-3..3] → ([-3,-2,-1], [0,1,2,3])`
- `break :: (a -> Bool) -> [a] -> ([a], [a])`
 - `break p l = (takeWhile q l, dropWhile q l)`
`where q x = not (p x)`

Разбивания на списъци

- **takeWhile** :: ($a \rightarrow \text{Bool}$) $\rightarrow [a] \rightarrow [a]$
 - връща първите елементи на списъка, за които е вярно условието
 - `takeWhile p = foldr (\x r -> if p x then x:r else []) []`
 - `takeWhile (<0) [-3..3] → [-3,-2,-1]`
- **dropWhile** :: ($a \rightarrow \text{Bool}$) $\rightarrow [a] \rightarrow [a]$
 - премахва първите елементи на списъка, за които е вярно условието
 - `dropWhile (<0) [-3..3] → [0,1,2,3]`
- **span** :: ($a \rightarrow \text{Bool}$) $\rightarrow [a] \rightarrow ([a], [a])$
 - `span p l = (takeWhile p l, dropWhile p l)`
 - `span (<0) [-3..3] → ([-3,-2,-1], [0,1,2,3])`
- **break** :: ($a \rightarrow \text{Bool}$) $\rightarrow [a] \rightarrow ([a], [a])$
 - `break p l = (takeWhile q l, dropWhile q l)`
`where q x = not (p x)`
 - `break (>0) [-3..3] → ([-3,-2,-1,0], [1,2,3])`

Логически квантори (any, all)

- `any` :: (`a` → `Bool`) → [`a`] → `Bool`

Логически квантори (any, all)

- `any` :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool}$
 - проверява дали предикатът е изпълнен за **някой елемент** от списъка

Логически квантори (any, all)

- `any` :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool}$
 - проверява дали предикатът е изпълнен за **някой елемент** от списъка
 - `any p l = or (map p l)`

Логически квантори (any, all)

- `any :: (a -> Bool) -> [a] -> Bool`
 - проверява дали предикатът е изпълнен за **някой елемент** от списъка
 - `any p l = or (map p l)`
 - `elem x = any (==x)`

Логически квантори (any, all)

- `any` :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool}$
 - проверява дали предикатът е изпълнен за **някой елемент** от списъка
 - `any p l = or (map p l)`
 - `elem x = any (==x)`
- `all` :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool}$

Логически квантори (any, all)

- `any` :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool}$
 - проверява дали предикатът е изпълнен за **някой елемент** от списъка
 - `any p l = or (map p l)`
 - `elem x = any (==x)`
- `all` :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool}$
 - проверява дали предикатът е изпълнен за **всички елементи** на списъка

Логически квантори (any, all)

- `any :: (a -> Bool) -> [a] -> Bool`
 - проверява дали предикатът е изпълнен за **някой елемент** от списъка
 - `any p l = or (map p l)`
 - `elem x = any (==x)`
- `all :: (a -> Bool) -> [a] -> Bool`
 - проверява дали предикатът е изпълнен за **всички елементи** на списъка
 - `all p l = and (map p l)`

Логически квантори (any, all)

- **any** :: ($a \rightarrow \text{Bool}$) $\rightarrow [a] \rightarrow \text{Bool}$
 - проверява дали предикатът е изпълнен за **някой елемент** от списъка
 - `any p l = or (map p l)`
 - `elem x = any (==x)`
- **all** :: ($a \rightarrow \text{Bool}$) $\rightarrow [a] \rightarrow \text{Bool}$
 - проверява дали предикатът е изпълнен за **всички елементи** на списъка
 - `all p l = and (map p l)`
 - `sorted l = all (\(x,y) \rightarrow x \leq y) (zip l (tail l))`