

# Кортежи и списъци

Трифон Трифонов

Функционално програмиране, 2024/25 г.

20 декември 2023 г. – 3 януари 2024 г.

Тази презентация е достъпна под лиценза Creative Commons Признание-Некомерсиално-Споделяне на споделеното 4.0 Международен 

# 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:**  $(1, 2)$ ,  $(3.5, 'A', \text{False})$ ,  $((\text{"square"}, (^2)), 1.0)$
- 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:**  $(1, 2)$ ,  $(3.5, 'A', \text{False})$ ,  $((\text{"square"}, (^2)), 1.0)$
- 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:**  $(1, 2)$ ,  $(3.5, 'A', \text{False})$ ,  $((\text{"square"}, (^2)), 1.0)$
- 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:**  $(1, 2)$ ,  $(3.5, 'A', \text{False})$ ,  $((\text{"square"}, (^2)), 1.0)$
- 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:**  $(1, 2)$ ,  $(3.5, 'A', \text{False})$ ,  $((\text{"square"}, (^2)), 1.0)$
- 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

# Кортежи (tuples)

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

- **Примери:**  $(1, 2)$ ,  $(3.5, 'A', \text{False})$ ,  $((\text{"square"}, (^2)), 1.0)$
- 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
  - $\text{fst} :: (a, b) \rightarrow a$  — първа компонента на наредена двойка



# Кортежи (tuples)

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

- **Примери:**  $(1, 2)$ ,  $(3.5, 'A', \text{False})$ ,  $((\text{"square"}, (^2)), 1.0)$
- 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
  - $\text{fst} :: (a, b) \rightarrow a$  — първа компонента на наредена двойка
  - $\text{snd} :: (a, b) \rightarrow b$  — 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 Transformation = 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 Transformation = 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 Transformation = Point -> Point`
  - `type Vector = Point`
  - `addVectors :: Vector -> Vector -> Vector`
  - `addVectors v1 v2 = (fst v1 + fst v2, snd v1 + snd v2)`

# Особености на кортежите

- `fst`  $(1, 2, 3) \longrightarrow ?$



# Особености на кортежите

- `fst`  $(1, 2, 3) \rightarrow$  Грешка!

# Особености на кортежите

- `fst`  $(1, 2, 3) \longrightarrow$  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`  $(1, 2, 3) \longrightarrow$  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`  $(1, 2, 3) \longrightarrow$  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 (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)`

# 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`
- `getYear :: Student -> Int`
- `getYear (_, year, _) = year`

# 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`
- `getYear :: Student -> Int`
- `getYear (_, year, _) = year`
- 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`
- `getYear :: Student -> Int`
- `getYear (_, year, _) = year`
- 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`
- `getYear :: Student -> Int`
- `getYear (_, year, _) = year`
- образците на кортежи могат да се използват за “разглобяване” на кортежи при дефиниция
- `(x, y) = (3.5, 7.8)`
- `let (name, _, grade) = student in (name, min (grade + 1) 6)`

# Именувани образци

- намиране на студент с по-висока оценка

```
betterStudent (name1, year1, grade1) (name2, year2, grade2)  
  | grade1 > grade2 = (name1, year1, grade1)  
  | otherwise      = (name2, year2, grade2)
```



# Именувани образци

- намиране на студент с по-висока оценка

```
betterStudent (name1, year1, grade1) (name2, year2, grade2)  
  | grade1 > grade2 = (name1, year1, grade1)  
  | otherwise      = (name2, year2, grade2)
```

- ами ако имаме 10 полета?

# Именувани образци

- намиране на студент с по-висока оценка

```
betterStudent (name1, year1, grade1) (name2, year2, grade2)  
  | grade1 > grade2 = (name1, year1, grade1)  
  | otherwise      = (name2, year2, grade2)
```

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

# Именувани образци

- намиране на студент с по-висока оценка

```
betterStudent (name1, year1, grade1) (name2, year2, grade2)  
  | grade1 > grade2 = (name1, year1, grade1)  
  | otherwise      = (name2, year2, grade2)
```

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

# Именувани образци

- намиране на студент с по-висока оценка

```
betterStudent (name1, year1, grade1) (name2, year2, grade2)  
  | grade1 > grade2 = (name1, year1, grade1)  
  | otherwise      = (name2, year2, grade2)
```

- ами ако имаме 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 -> [a] -> [a]` е **дясноасоциативна** двуместна 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 -> [a] -> [a]` е **дясноасоциативна** двуместна 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 -> [a] -> [a]` е **дясноасоциативна** двуместна 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

- ❶ Празният списък `[]` е списък от тип `[a]`
  - ❷ 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 -> [a] -> [a]` е дясноасоциативна двуместна 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] :: ⊥`

# Примери

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

# Примери

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



# Примери

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

# Примери

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

# 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]]`
- `[]:[] :: ?`

# 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]: [[]] :: ?`

# 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] :: ?`

# 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],[[]]] :: ?`

# 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], ""] :: ?`

# 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'], []] :: ?`

# 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]`

# 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]]`

# 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

# 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  $l$ , 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



# 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 елемента  $x_i$
- **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  $l$ , 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`
  - `null [] = True`
  - `null _ = False`

# 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`
  - `null [] = True`
  - `null _ = False`
  - `length [] = 0`
  - `length (_,t) = 1 + length t`

## Случаи по образци (case)

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

## Случаи по образци (case)

- `case` <израз> `of` { <образец> `->` <израз> }<sup>+</sup>
- `case` <израз> `of` <образец<sub>1</sub>> `->` <израз<sub>1</sub>>  
...  
<образец<sub>n</sub>> `->` <израз<sub>n</sub>>



## Случаи по образци (case)

- `case` <израз> `of` { <образец> `->` <израз> }<sup>+</sup>
- `case` <израз> `of` <образец<sub>1</sub>> `->` <израз<sub>1</sub>>  
...  
    <образец<sub>n</sub>> `->` <израз<sub>n</sub>>
- ако <израз> пасва на <образец<sub>1</sub>>, връща <израз<sub>1</sub>>, иначе:
- ...
- ако <израз> пасва на <образец<sub>n</sub>>, връща <израз<sub>n</sub>>, иначе:
- Грешка!

## Случаи по образци (case)

- `case` <израз> `of` { <образец> `->` <израз> }<sup>+</sup>
- `case` <израз> `of` <образец<sub>1</sub>> `->` <израз<sub>1</sub>>  
...  
    <образец<sub>n</sub>> `->` <израз<sub>n</sub>>
- ако <израз> пасва на <образец<sub>1</sub>>, връща <израз<sub>1</sub>>, иначе:
- ...
- ако <израз> пасва на <образец<sub>n</sub>>, връща <израз<sub>n</sub>>, иначе:
- **Грешка!**
- Използването на образци в дефиниции всъщност е синтактична захар за конструкцията `case`!

## Случаи по образци (case)

- `case` <израз> `of` { <образец> `->` <израз> }<sup>+</sup>
- `case` <израз> `of` <образец<sub>1</sub>> `->` <израз<sub>1</sub>>  
...  
    <образец<sub>n</sub>> `->` <израз<sub>n</sub>>
- ако <израз> пасва на <образец<sub>1</sub>>, връща <израз<sub>1</sub>>, иначе:
- ...
- ако <израз> пасва на <образец<sub>n</sub>>, връща <израз<sub>n</sub>>, иначе:
- **Грешка!**
- Използването на образци в дефиниции всъщност е синтактична захар за конструкцията `case`!
- `case` може да се използва навсякъде, където се очаква израз

# 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`

# 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`
  
- $[a, a + \Delta x .. b] \rightarrow [a, a + \Delta x, a + 2\Delta x, \dots, b']$ , 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
  - $[1..3] ++ [5..7] \rightarrow [1,2,3,5,6,7]$

# Рекурсивни функции над списъци

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

$[] \quad ++ \text{ys} = \text{ys}$

$(x:\text{xs}) ++ \text{ys} = x:\text{xs} ++ \text{ys}$

# Рекурсивни функции над списъци

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

$[] ++ ys = ys$

$(x:xs) ++ ys = x:xs ++ ys$

- **reverse**  $:: [a] \rightarrow [a]$  — обръща списък
  - **reverse**  $[1..5] \rightarrow [5,4,3,2,1]$



# Рекурсивни функции над списъци

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

$[] ++ ys = ys$

$(x:xs) ++ ys = x:xs ++ ys$

- $reverse :: [a] \rightarrow [a]$  — обръща списък
  - $reverse [1..5] \rightarrow [5,4,3,2,1]$

$reverse [] = []$

$reverse (x:xs) = reverse xs ++ [x]$

# Рекурсивни функции над списъци

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

$[] \quad ++ \quad ys = ys$

$(x:xs) ++ ys = x:xs ++ ys$

- $reverse :: [a] \rightarrow [a]$  — обръща списък
  - $reverse [1..5] \rightarrow [5,4,3,2,1]$

$reverse [] = []$

$reverse (x:xs) = reverse xs ++ [x]$

- $(!!) :: [a] \rightarrow Int \rightarrow a$  — елемент с пореден номер (от 0)
  - $"Haskell" !! 2 \rightarrow 's'$

# Рекурсивни функции над списъци

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

```
[]      ++ ys = ys  
(x:xs) ++ ys = x:xs ++ ys
```

- `reverse`  $:: [a] \rightarrow [a]$  — обръща списък
  - `reverse`  $[1..5] \rightarrow [5,4,3,2,1]$

```
reverse []      = []  
reverse (x:xs) = reverse xs ++ [x]
```

- $(!!) :: [a] \rightarrow \text{Int} \rightarrow a$  — елемент с пореден номер (от 0)
  - `"Haskell" !! 2`  $\rightarrow$  `'s'`
- `elem`  $:: \text{Eq } a \Rightarrow a \rightarrow [a] \rightarrow \text{Bool}$  — проверка за принадлежност на елемент към списък
  - `3 'elem' [1..5]`  $\rightarrow$  `True`

# 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

## 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> } ]

# 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

- $[ 2 * x \mid x \leftarrow [1..5] ] \longrightarrow ?$

## Примери за отделяне на списъци

- $[ 2 * x \mid x \leftarrow [1..5] ] \longrightarrow [2,4,6,8,10]$

## Примери за отделяне на списъци

- `[ 2 * x | x <- [1..5] ]`  $\longrightarrow$  `[2,4,6,8,10]`
- `[ x^2 | x <- [1..10], odd x]`  $\longrightarrow$  ?

## Примери за отделяне на списъци

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

## Примери за отделяне на списъци

- `[ 2 * x | x <- [1..5] ]`  $\longrightarrow$  `[2,4,6,8,10]`
- `[ x^2 | x <- [1..10], odd x]`  $\longrightarrow$  `[1,9,25,49,81]`
- `[ name | (name, _, grade) <- students, grade >= 3 ]`

## Примери за отделяне на списъци

- `[ 2 * x | x <- [1..5] ]`  $\longrightarrow$  `[2,4,6,8,10]`
- `[ x^2 | x <- [1..10], odd x]`  $\longrightarrow$  `[1,9,25,49,81]`
- `[ name | (name, _, grade) <- students, grade >= 3 ]`
- `[ x^2 + y^2 | (x, y) <- vectors, x >= 0, y >= 0 ]`

## Примери за отделяне на списъци

- `[ 2 * x | x <- [1..5] ]`  $\longrightarrow$  `[2,4,6,8,10]`
- `[ x^2 | x <- [1..10], odd x]`  $\longrightarrow$  `[1,9,25,49,81]`
- `[ name | (name, _, grade) <- students, grade >= 3 ]`
- `[ 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)

## Примери за отделяне на списъци

- `[ 2 * x | x <- [1..5] ]`  $\longrightarrow$  `[2,4,6,8,10]`
- `[ x^2 | x <- [1..10], odd x ]`  $\longrightarrow$  `[1,9,25,49,81]`
- `[ name | (name, _, grade) <- students, grade >= 3 ]`
- `[ 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 <- ["grass", "sky"] ]`  
 $\longrightarrow$  ?



## Примери за отделяне на списъци

- `[ 2 * x | x <- [1..5] ]`  $\longrightarrow$  `[2,4,6,8,10]`
- `[ x^2 | x <- [1..10], odd x]`  $\longrightarrow$  `[1,9,25,49,81]`
- `[ name | (name, _, grade) <- students, grade >= 3 ]`
- `[ 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 <- ["grass", "sky"]]`  
 $\longrightarrow$  `["green grass", "green sky", "blue grass", "blue sky"]`

## Примери за отделяне на списъци

- `[ 2 * x | x <- [1..5] ]`  $\longrightarrow$  `[2,4,6,8,10]`
- `[ x^2 | x <- [1..10], odd x ]`  $\longrightarrow$  `[1,9,25,49,81]`
- `[ name | (name, _, grade) <- students, grade >= 3 ]`
- `[ 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 <- ["grass", "sky"] ]`  
 $\longrightarrow$  `["green grass", "green sky", "blue grass", "blue sky"]`
- `[ (x,y) | x <- [1,2,3], y <- [5,6,7], x + y <= 8 ]`  
 $\longrightarrow$  ?

## Примери за отделяне на списъци

- `[ 2 * x | x <- [1..5] ]`  $\longrightarrow$  `[2,4,6,8,10]`
- `[ x^2 | x <- [1..10], odd x ]`  $\longrightarrow$  `[1,9,25,49,81]`
- `[ name | (name, _, grade) <- students, grade >= 3 ]`
- `[ 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 <- ["grass", "sky"] ]`  
 $\longrightarrow$  `["green grass", "green sky", "blue grass", "blue sky"]`
- `[ (x,y) | x <- [1,2,3], y <- [5,6,7], x + y <= 8 ]`  
 $\longrightarrow$  `[(1,5),(1,6),(1,7),(2,5),(2,6),(3,5)]`

## Примери за отделяне на списъци

- `[ 2 * x | x <- [1..5] ]`  $\longrightarrow$  `[2,4,6,8,10]`
- `[ x^2 | x <- [1..10], odd x ]`  $\longrightarrow$  `[1,9,25,49,81]`
- `[ name | (name, _, grade) <- students, grade >= 3 ]`
- `[ 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 <- ["grass", "sky"] ]`  
 $\longrightarrow$  `["green grass", "green sky", "blue grass", "blue sky"]`
- `[ (x,y) | x <- [1,2,3], y <- [5,6,7], x + y <= 8 ]`  
 $\longrightarrow$  `[(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] -> 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
- `and :: [Bool] -> Bool` — конюнкция на булеви стойности
- `or :: [Bool] -> Bool` — дизюнкция на булеви стойности

# 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
- `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] -> 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
- `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] -> 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
- `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]`

# 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
- `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]`



# $\lambda$ -функции

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

## $\lambda$ -функции

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

## $\lambda$ -функции

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

# $\lambda$ -функции

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

# $\lambda$ -функции

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

# $\lambda$ -функции

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

# $\lambda$ -функции

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

# λ-функции

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



# λ-функции

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

# λ-функции

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

# λ-функции

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

# λ-функции

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

# λ-функции

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

# λ-функции

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

# λ-функции

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

## Свойства на $\lambda$ -функциите

- $$\lambda x_1 x_2 \dots x_n \rightarrow \langle \text{тяло} \rangle$$
$$\iff \lambda x_1 \rightarrow \lambda x_2 \rightarrow \dots \lambda x_n \rightarrow \langle \text{тяло} \rangle$$



# Свойства на $\lambda$ -функциите

- $\lambda x_1 x_2 \dots x_n \rightarrow \langle \text{тяло} \rangle$   
 $\iff \lambda x_1 \rightarrow \lambda x_2 \rightarrow \dots \lambda x_n \rightarrow \langle \text{тяло} \rangle$
- $f x = \langle \text{тяло} \rangle$   
 $\iff f = \lambda x \rightarrow \langle \text{тяло} \rangle$

# Свойства на $\lambda$ -функциите

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

# Свойства на $\lambda$ -функциите

- $\backslash x_1 x_2 \dots x_n \rightarrow \langle \text{тяло} \rangle$   
 $\iff \backslash x_1 \rightarrow \backslash x_2 \rightarrow \dots \backslash x_n \rightarrow \langle \text{тяло} \rangle$
- $f x = \langle \text{тяло} \rangle$   
 $\iff f = \backslash x \rightarrow \langle \text{тяло} \rangle$
- $f x y = \langle \text{тяло} \rangle$   
 $\iff f x = \backslash y \rightarrow \langle \text{тяло} \rangle$   
 $\iff f = \backslash x y \rightarrow \langle \text{тяло} \rangle$
- $f x_1 \dots x_n = \langle \text{тяло} \rangle$   
 $\iff f x_1 \dots x_{n-1} = \backslash x_n \rightarrow \langle \text{тяло} \rangle$   
 $\iff \dots$   
 $\iff f = \backslash x_1 \dots x_n \rightarrow \langle \text{тяло} \rangle$

# Свойства на $\lambda$ -функциите

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

# Трансформация (map)

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

# Трансформация (map)

- `map :: (a -> b) -> [a] -> [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 -> 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] → ?`

# 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>`]  
     $\longleftrightarrow$   
    `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>`]  
     $\longleftrightarrow$   
    `map` (`\<образец>` `->` `<израз>`)  
        (`filter` (`\<образец>` `->` `<условие>`) `<списък>`)
- [`<образец>` | `<образец>` `<-` `<списък>`, `<условие1>`, `<условие2>`]  
     $\longleftrightarrow$   
    `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>` `<- <списък>`, `<условие>`]  
     $\longleftrightarrow$   
    `map` (`\<образец> -> <израз>`)  
        (`filter` (`\<образец> -> <условие>`) `<списък>`)
- [`<образец>` | `<образец>` `<- <списък>`, `<условие1>`, `<условие2>`]  
     $\longleftrightarrow$   
    `filter` (`\<образец> -> <условие2>`)  
        (`filter` (`\<образец> -> <условие1>`) `<списък>`)
- [`<израз>` | `<образец1>` `<- <списък1>`, `<образец2>` `<- <списък2>`]  
     $\longleftrightarrow$   
    `concat` (`map` (`\<образец1> ->`  
        `map` (`\<образец2> -> <израз>`) `<списък2>`)  
        `<списък1>`)

# 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>, <условие<sub>1</sub>>, <условие<sub>2</sub>>]

↔

```
filter (\<образец> -> <условие2>)
  (filter (\<образец> -> <условие1>) <списък>)
```
- [<израз> | <образец<sub>1</sub>> <- <списък<sub>1</sub>>, <образец<sub>2</sub>> <- <списък<sub>2</sub>>]

↔

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

# 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>`)
- [`<израз>` | `<образец1>` `<-` `<списък1>`, `<образец2>` `<-` `<списък2>`]  
↔  
`concatMap` (`\<образец1> ->`  
    `map` (`\<образец2> -> <израз>`) `<списък2>`)  
    `<списък1>`

## Дясно свиване (foldr)

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

## Дясно свиване (foldr)

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

## Дясно свиване (foldr)

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



## Дясно свиване (foldr)

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

## Дясно свиване (foldr)

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

## Дясно свиване (foldr)

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

## Дясно свиване (foldr)

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

## Дясно свиване (foldr)

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

## Дясно свиване (foldr)

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

## Дясно свиване (foldr)

- `foldr :: (a -> b -> b) -> b -> [a] -> b`
- `foldr op nv [x1, x2, ..., xn] =`  
`x1 'op' (x2 'op' ... (xn 'op' nv) ...)`
- `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] =`  
`x1 'op' (x2 'op' ... (xn 'op' nv) ...)`
- `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:) else id) r) []`

## Дясно свиване (foldr)

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

## 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 (:)) []`

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

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

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

- `foldr1` :: (a -> a -> a) -> [a] -> a
- `foldr1` op  $[x_1, x_2, \dots, x_n]$  =  
 $x_1$  'op'  $(x_2$  'op'  $\dots (x_{n-1}$  'op'  $x_n) \dots)$



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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- `foldr1` :: (a -> a -> a) -> [a] -> a
- `foldr1` op  $[x_1, x_2, \dots, x_n]$  =  
 $x_1$  'op' ( $x_2$  'op' ... ( $x_{n-1}$  'op'  $x_n$ ) ...)
- `foldr1` \_ [x] = x  
`foldr1` op (x:xs) = x 'op' `foldr1` op xs
- `foldl1` :: (a -> a -> a) -> [a] -> a
- `foldl1` op  $[x_1, x_2, \dots, x_n]$  =  
 $(\dots ((x_1$  'op'  $x_2)$  ...) 'op'  $x_n$
- `foldl1` op (x:xs) = `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 [x<sub>1</sub>, x<sub>2</sub>, ..., x<sub>n</sub>] =  
[x<sub>1</sub> 'op' (x<sub>2</sub> 'op' ... (x<sub>n</sub> 'op' nv) ...),  
x<sub>2</sub> 'op' (... (x<sub>n</sub> 'op' nv) ...),  
...,  
x<sub>n</sub> '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 [x<sub>1</sub>, x<sub>2</sub>, ..., x<sub>n</sub>] =  
 [x<sub>1</sub> 'op' (x<sub>2</sub> 'op' ... (x<sub>n</sub> 'op' nv) ...),  
 x<sub>2</sub> 'op' (... (x<sub>n</sub> 'op' nv) ...),  
 ...  
 x<sub>n</sub> '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 [x<sub>1</sub>, x<sub>2</sub>, ..., x<sub>n</sub>] =  
 [x<sub>1</sub> 'op' (x<sub>2</sub> 'op' ... (x<sub>n</sub> 'op' nv) ...),  
 x<sub>2</sub> 'op' (... (x<sub>n</sub> 'op' nv) ...),  
 ...  
 x<sub>n</sub> 'op' nv,  
 nv]
- `scanl` :: (b -> a -> b) -> b -> [a] -> [b]
- `scanl` op nv [x<sub>1</sub>, x<sub>2</sub>, ..., x<sub>n</sub>] =  
 [nv,  
 nv 'op' x<sub>1</sub>,  
 (nv 'op' x<sub>1</sub>) 'op' x<sub>2</sub>,  
 ...  
 (... ((nv 'op' x<sub>1</sub>) 'op' x<sub>2</sub>) ...) 'op' x<sub>n</sub>]

## Съшиване на списъци (zip, zipWith)

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

## Съшиване на списъци (zip, zipWith)

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



## Съшиване на списъци (zip, zipWith)

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

## Съшиване на списъци (zip, zipWith)

- `zip` :: `[a] -> [b] -> [(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)] -> ([a],[b])`
  - разделя списък от двойки на два списъка с равна дължина

## Съшиване на списъци (zip, zipWith)

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

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

- `zip` :: `[a] -> [b] -> [(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)] -> ([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 -> b -> c) -> [a] -> [b] -> [c]`
  - “съшива” два списъка с дадена двуместна операция

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

- `zip` :: `[a] -> [b] -> [(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)] -> ([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 -> b -> c) -> [a] -> [b] -> [c]`
  - “съшива” два списъка с дадена двуместна операция
  - `zipWith`  $\text{op } [x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \longrightarrow [\text{op } x_1 y_1, \text{op } x_2 y_2, \dots, \text{op } x_n y_n]$

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

- `zip` :: `[a] -> [b] -> [(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)] -> ([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 -> b -> c) -> [a] -> [b] -> [c]`
  - “съшива” два списъка с дадена двуместна операция
  - `zipWith`  $\text{op } [x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \longrightarrow [\text{op } x_1 y_1, \text{op } x_2 y_2, \dots, \text{op } x_n y_n]$
- Примери:

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

- `zip` :: `[a] -> [b] -> [(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)] -> ([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 -> b -> c) -> [a] -> [b] -> [c]`
  - “съшива” два списъка с дадена двуместна операция
  - `zipWith`  $\text{op } [x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \longrightarrow [\text{op } x_1 y_1, \text{op } x_2 y_2, \dots, \text{op } x_n y_n]$
- Примери:
  - `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 двуместна операция
  - $\text{zipWith } \text{op } [x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \longrightarrow [\text{op } x_1 y_1, \text{op } x_2 y_2, \dots, \text{op } x_n y_n]$
- **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] -> [b] -> [(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)] -> ([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 -> b -> c) -> [a] -> [b] -> [c]`
  - “съшива” два списъка с дадена двуместна операция
  - `zipWith`  $\text{op } [x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \longrightarrow [\text{op } x_1 y_1, \text{op } x_2 y_2, \dots, \text{op } x_n y_n]$
- 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] -> [b] -> [(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)] -> ([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 -> b -> c) -> [a] -> [b] -> [c]`
  - “съшива” два списъка с дадена двуместна операция
  - `zipWith`  $\text{op } [x_1, x_2, \dots, x_n] [y_1, y_2, \dots, y_n] \longrightarrow [\text{op } x_1 y_1, \text{op } x_2 y_2, \dots, \text{op } x_n y_n]$
- **Примери:**
  - `zip`  $[1..3] [5..10] \longrightarrow [(1,5), (2,6), (3,7)]$
  - `zipWith`  $(*) [1..3] [5..10] \longrightarrow [5,12,21]$
  - `zip` = `zipWith` `(,)`
  - `unzip` = `foldr`  $\backslash(x,y) (xs,ys) -> (x:xs,y:ys)) ([], [])$

## 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 -> 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)`
  - `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 -> Bool) -> [a] -> Bool
  - проверява дали предикатът е изпълнен за **някой елемент** от 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)`

# 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 -> 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`

# 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

# 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 -> 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)`
  - `sorted l = all (\(x,y) -> x <= y) (zip l (tail l))`