

# Кортежи и списъци

Трифон Трифонов

Функционално програмиране, спец. Информатика, 2016/17 г.

15 декември 2016 г.

# Кортежи (tuples)

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

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

# Кортежи (tuples)

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

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

# Кортежи (tuples)

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

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

# Кортежи (tuples)

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

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

# Кортежи (tuples)

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

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

# Кортежи (tuples)

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

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

# Кортежи (tuples)

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

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



# Кортежи (tuples)

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

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

## Потребителски типове

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

## Потребителски типове

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

# Потребителски типове

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

# Потребителски типове

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

## Потребителски типове

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

# Потребителски типове

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

# Потребителски типове

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



# Потребителски типове

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

# Потребителски типове

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

# Потребителски типове

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

# Потребителски типове

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

# Потребителски типове

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

# Потребителски типове

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

# Потребителски типове

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

# Особености на кортежите

- `fst (1,2,3) → ?`



# Особености на кортежите

- `fst (1,2,3)` → Грешка!

# Особености на кортежите

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

# Особености на кортежите

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

# Особености на кортежите

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

# Особености на кортежите

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

# Особености на кортежите

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

## Образци на кортежи

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



## Образци на кортежи

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

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

## Образци на кортежи

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

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

- `addVectors (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`
- `getFN :: Student -> Int`
- `getFN (_, fn, _) = fn`

## Образци на кортежи

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

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

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

## Образци на кортежи

Образец на кортеж е конструкция от вида  $(p_1, p_2, \dots, p_n)$ .  
 Пасва на всеки кортеж от точно  $n$  елемента  $(x_1, x_2, \dots, x_n)$ , за който  
 образецът  $p_i$  пасва на елемента  $x_i$ .

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

## Образци на кортежи

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

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

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

# Именувани образци

- намиране на студент с по-висока оценка

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



# Именувани образци

- намиране на студент с по-висока оценка

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

- ами ако имахме 10 полета?

# Именувани образци

- намиране на студент с по-висока оценка

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

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

# Именувани образци

- намиране на студент с по-висока оценка

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

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

## Именувани образци

- намиране на студент с по-висока оценка

```

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

```

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

```

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

```

# Списъци

## Дефиниция

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

# Списъци

## Дефиниция

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

# Списъци

## Дефиниция

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

# Списъци

## Дефиниция

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



# Списъци

## Дефиниция

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

# 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** бинарна операция
  - `(1:(2:(3:(4:[])))) = 1:2:3:4:[] ≠ (((1:2):3):4):[]`
  - `[a1, a2, ..., an]` е по-удобен запис за `a1:(a2:... (an:[])...)`
  - `[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

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

$a = Int$

# 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]]`

$X : Y$   
 $::$       $::$   
 $a$       $[a]$

# 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]:[[]] :: [[Int]]`
- `[]:[1] :: ?`

$[ ] :: [a]$   
 $[ ] :: [ [a] ]$   
 $1 :: [a]$

# Примери

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



# Примери

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

$[1,?,3] :: a = [Int]$   
 $[] :: a \in [Int]$

## 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]]`

# 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] :: [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

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

# 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

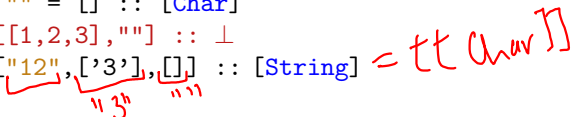
- В Haskell низовете са представени като списъци от символи
- `type String = [Char]`
- Всички операции над списъци важат и над низове
- Примери
  - `['H', 'e', 'l', 'l', 'o'] = "Hello"`
  - `'H':'e':'l':'l':'o':[] = "Hello"`
  - `'H':'e':"llo" = "Hello"`
  - `"" = [] :: [Char]`
  - `[[1,2,3], ""] ::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



# 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] = [[Char]]`  


# Основни функции за списъци

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

# Основни функции за списъци

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

# Основни функции за списъци

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

# Основни функции за списъци

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

# Основни функции за списъци

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

# Основни функции за списъци

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



# Основни функции за списъци

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

# Основни функции за списъци

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

# Основни функции за списъци

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

# Основни функции за списъци

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

# Генератори на списъци

Можем да генерираме списъци от последователни елементи

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

# Генератори на списъци

Можем да генерираме списъци от последователни елементи

- $[a..b] \rightarrow [a, a+1, a+2, \dots, b]$
- Пример:  $[1..5] \rightarrow [1, 2, 3, 4, 5]$
- Пример:  $['a'..'e'] \rightarrow \text{"abcde"}$
- Синтактична захар за `enumFromTo` `from` `to`
- $[a, a + \Delta x .. b] \rightarrow [a, a + \Delta x, a + 2\Delta x, \dots, b']$ , 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
  - $[1..3] ++ [5..7] \rightarrow [1,2,3,5,6,7]$
- $a ++ b = \text{if null } a \text{ then } b \text{ else head } a : \text{tail } a ++ b$



# Рекурсивни функции над списъци

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

# Рекурсивни функции над списъци

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

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

# Рекурсивни функции над списъци

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

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

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

# Рекурсивни функции над списъци

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

`reverse a`

| `null a` = `a`

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

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

## Образци и списъци

Много удобно е да използваме образци на списъци:

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

## Образци и списъци

Много удобно е да използваме образци на списъци:

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

# Образци и списъци

Много удобно е да използваме образци на списъци:

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

## Образци и списъци

Много удобно е да използваме образци на списъци:

- $p_h:p_t$  — пасва на всеки непразен списък  $l$ , 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> }<sup>+</sup>

## 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

# 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`