

Основни понятия в Haskell

Трифон Трифонов

Функционално програмиране, 2017/18 г.

7 декември 2017 г.

Какво е Haskell?

Какво е Haskell?



Haskell Brooks Curry
(1900–1982)

Какво е Haskell?

Какво е Haskell?

```
fact 0 = 1
fact n = n * fact (n-1)
```

Какво е Haskell?

```
fact 0 = 1
fact n = n * fact (n-1)

quickSort []      = []
quickSort (x:xs) = quickSort less ++ [x] ++ quickSort more
  where less = filter (≤x) xs
        more = filter (>x) xs
```

Какво е Haskell?

```
fact 0 = 1
fact n = n * fact (n-1)

quickSort []      = []
quickSort (x:xs) = quickSort less ++ [x] ++ quickSort more
  where less = filter (≤ x) xs
        more = filter (> x) xs
```

```
студенти = [("Иван", 40000, 3.5), ("Мария", 60001, 5.5),
            ("Петър", 40002, 5.0), ("Галя", 40003, 4.75)]
избрани = foldr1 (++) [ '':име | (име, фн, оценка) <- студенти,
                           оценка > 4.5, фн < 60000 ]
```

Какво е Haskell?

Какво е Haskell?

- Чист функционален език (без странични ефекти)
- Статично типизиран с автоматичен извод на типовете
- Използва нестриктно (лениво) оценяване
- Стандартизиран (Haskell 2010 Language Report)

Помощни материали

- ① S. Thompson. Haskell: The Craft of Functional Programming (2nd ed.). Addison-Wesley, 1999.
- ② P. Hudak, Peterson J., Fasel J. A Gentle Introduction to Haskell 98, 1999 (Internet, 2008).
- ③ Haskell Wiki: <https://wiki.haskell.org/Haskell>
- ④ Haskell Platform: <https://www.haskell.org/platform/>

Синтактични елементи

- Идентификатори: fact, _myvar, студенти

Синтактични елементи

- Идентификатори: fact, _myvar, студенти
 - имена на обекти, започват с малка буква или _

Синтактични елементи

- Идентификатори: `fact`, `_myvar`, `студенти`
 - имена на обекти, започват с малка буква или `_`
- Запазени идентификатори: `case`, `if`, `let`, `where`, ...

Синтактични елементи

- Идентификатори: `fact`, `_myvar`, `студенти`
 - имена на обекти, започват с малка буква или `_`
- Запазени идентификатори: `case`, `if`, `let`, `where`, ...
- Конструктори: `Integer`, `Maybe`, `Just`, ...

Синтактични елементи

- Идентификатори: `fact`, `_myvar`, `студенти`
 - имена на обекти, започват с малка буква или `_`
- Запазени идентификатори: `case`, `if`, `let`, `where`, ...
- Конструктори: `Integer`, `Maybe`, `Just`, ...
 - имена на конструкции, започват с главна буква

Синтактични елементи

- Идентификатори: `fact`, `_myvar`, `студенти`
 - имена на обекти, започват с малка буква или `_`
- Запазени идентификатори: `case`, `if`, `let`, `where`, ...
- Конструктори: `Integer`, `Maybe`, `Just`, ...
 - имена на конструкции, започват с главна буква
- Числа: `10`, `-5.12`, `3.2e+2`, `1.2E-2`, `0x2f`, `0o35`

Синтактични елементи

- Идентификатори: `fact`, `_myvar`, `студенти`
 - имена на обекти, започват с малка буква или `_`
- Запазени идентификатори: `case`, `if`, `let`, `where`, ...
- Конструктори: `Integer`, `Maybe`, `Just`, ...
 - имена на конструкции, започват с главна буква
- Числа: `10`, `-5.12`, `3.2e+2`, `1.2E-2`, `0x2f`, `0o35`
- Операции: `+`, `*`, `&%`, `<==>`, `♠`

Синтактични елементи

- Идентификатори: `fact`, `_myvar`, `студенти`
 - имена на обекти, започват с малка буква или `_`
- Запазени идентификатори: `case`, `if`, `let`, `where`, ...
- Конструктори: `Integer`, `Maybe`, `Just`, ...
 - имена на конструкции, започват с главна буква
- Числа: `10`, `-5.12`, `3.2e+2`, `1.2E-2`, `0x2f`, `0o35`
- Операции: `+`, `*`, `&%`, `<==>`, 
 - поредица от символи (без букви и цифри)

Синтактични елементи

- Идентификатори: `fact`, `_myvar`, `студенти`
 - имена на обекти, започват с малка буква или `_`
- Запазени идентификатори: `case`, `if`, `let`, `where`, ...
- Конструктори: `Integer`, `Maybe`, `Just`, ...
 - имена на конструкции, започват с главна буква
- Числа: `10`, `-5.12`, `3.2e+2`, `1.2E-2`, `0x2f`, `0o35`
- Операции: `+`, `*`, `&%`, `<==>`, 
 - поредица от символи (без букви и цифри)
 - всички операции с изключение на унарния – са инфиксни

Синтактични елементи

- Идентификатори: `fact`, `_myvar`, `студенти`
 - имена на обекти, започват с малка буква или `_`
- Запазени идентификатори: `case`, `if`, `let`, `where`, ...
- Конструктори: `Integer`, `Maybe`, `Just`, ...
 - имена на конструкции, започват с главна буква
- Числа: `10`, `-5.12`, `3.2e+2`, `1.2E-2`, `0x2f`, `0o35`
- Операции: `+`, `*`, `&%`, `<==>`, 
 - поредица от символи (без букви и цифри)
 - всички операции с изключение на унарния - са инфиксни
- Запазени операции: `..` : `::` = `\|` `<-` `->` `@` `~` `=>`

Синтактични елементи

- Идентификатори: `fact`, `_myvar`, `студенти`
 - имена на обекти, започват с малка буква или `_`
- Запазени идентификатори: `case`, `if`, `let`, `where`, ...
- Конструктори: `Integer`, `Maybe`, `Just`, ...
 - имена на конструкции, започват с главна буква
- Числа: `10`, `-5.12`, `3.2e+2`, `1.2E-2`, `0x2f`, `0o35`
- Операции: `+`, `*`, `&%`, `<==>`, 
 - поредица от символи (без букви и цифри)
 - всички операции с изключение на унарния - са инфиксни
- Запазени операции: `..` : `::` = `\|` `<-` `->` `@` `~` `=>`
- Специални символи: `()`, `,` `[]`, `'`, `{ }`

Синтактични елементи

- Идентификатори: `fact`, `_myvar`, `студенти`
 - имена на обекти, започват с малка буква или `_`
- Запазени идентификатори: `case`, `if`, `let`, `where`, ...
- Конструктори: `Integer`, `Maybe`, `Just`, ...
 - имена на конструкции, започват с главна буква
- Числа: `10`, `-5.12`, `3.2e+2`, `1.2E-2`, `0x2f`, `0o35`
- Операции: `+`, `*`, `&%`, `<==>`, 
 - поредица от символи (без букви и цифри)
 - всички операции с изключение на унарния - са инфиксни
- Запазени операции: `..` : `::` = `\|` `<-` `->` `@` `~` `=>`
- Специални символи: `()`, `,` `[]`, `' { }`
- Знаци: `'a'`, `'\\n'`, `'+'`

Синтактични елементи

- Идентификатори: `fact`, `_myvar`, `студенти`
 - имена на обекти, започват с малка буква или `_`
- Запазени идентификатори: `case`, `if`, `let`, `where`, ...
- Конструктори: `Integer`, `Maybe`, `Just`, ...
 - имена на конструкции, започват с главна буква
- Числа: `10`, `-5.12`, `3.2e+2`, `1.2E-2`, `0x2f`, `0o35`
- Операции: `+`, `*`, `&%`, `<==>`, 
 - поредица от символи (без букви и цифри)
 - всички операции с изключение на унарния - са инфиксни
- Запазени операции: `..` : `::` = `\|` `<-` `->` `@` `~` `=>`
- Специални символи: `()`, `,` `[]`, `' { }`
- Знаци: `'a'`, `'\\n'`, `'+'`
- Низове: `"Hello, world!"`, `"произволен низ"`

Декларации и дефиниции

- <име> :: <тип> (типова декларация)

Декларации и дефиниции

- <име> :: <тип> (типова декларация)
- декларира се, че <име> ще се свързва със стойности от <тип>

Декларации и дефиниции

- <име> :: <тип> (типова декларация)
- декларира се, че <име> ще се свързва със стойности от <тип>
- **типовите декларации са незадължителни: в повечето случаи Haskell може сам да се ориентира за правилния тип**

Декларации и дефиниции

- <име> :: <тип> (типова декларация)
- декларира се, че <име> ще се свързва със стойности от <тип>
- **типовите декларации са незадължителни: в повечето случаи Haskell може сам да се ориентира за правилния тип**
 - x :: Int
 - y :: Double
 - z :: String

Декларации и дефиниции

- <име> :: <тип> (типова декларация)
- декларира се, че <име> ще се свързва със стойности от <тип>
- **типовите декларации са незадължителни: в повечето случаи Haskell може сам да се ориентира за правилния тип**
 - x :: Int
 - y :: Double
 - z :: String
- <име> = <израз> (дефиниция)

Декларации и дефиниции

- `<име> :: <тип>` (типова декларация)
- декларира се, че `<име>` ще се свързва със стойности от `<тип>`
- **типовите декларации са незадължителни: в повечето случаи Haskell може сам да се ориентира за правилния тип**
 - `x :: Int`
 - `y :: Double`
 - `z :: String`
- `<име> = <израз>` (дефиниция)
- `<име>` се свърза с `<израз>`

Декларации и дефиниции

- `<име> :: <тип>` (типова декларация)
- декларира се, че `<име>` ще се свързва със стойности от `<тип>`
- **типовите декларации са незадължителни: в повечето случаи Haskell може сам да се ориентира за правилния тип**
 - `x :: Int`
 - `y :: Double`
 - `z :: String`
- `<име> = <израз>` (дефиниция)
- `<име>` се свърза с `<израз>`
 - `x = 2`

Декларации и дефиниции

- <име> :: <тип> (типова декларация)
- декларира се, че <име> ще се свързва със стойности от <тип>
- **типовите декларации са незадължителни: в повечето случаи Haskell може сам да се ориентира за правилния тип**
 - x :: Int
 - y :: Double
 - z :: String
- <име> = <израз> (дефиниция)
- <име> се свърза с <израз>
 - x = 2
 - y = x^2 + 7.5

Декларации и дефиниции

- <име> :: <тип> (типова декларация)
- декларира се, че <име> ще се свързва със стойности от <тип>
- **типовите декларации са незадължителни: в повечето случаи Haskell може сам да се ориентира за правилния тип**
 - x :: Int
 - y :: Double
 - z :: String
- <име> = <израз> (дефиниция)
- <име> се свърза с <израз>
 - x = 2
 - y = fromIntegral x^2 + 7.5

Декларации и дефиниции

- `<име> :: <тип>` (типовна декларация)
- декларира се, че `<име>` ще се свързва със стойности от `<тип>`
- **типовите декларации са незадължителни: в повечето случаи Haskell може сам да се ориентира за правилния тип**
 - `x :: Int`
 - `y :: Double`
 - `z :: String`
- `<име> = <израз>` (дефиниция)
- `<име>` се свърза с `<израз>`
 - `x = 2`
 - `y = fromIntegral x^2 + 7.5`
 - `z = "Hello"`

Декларации и дефиниции

- <име> :: <тип> (типовна декларация)
- декларира се, че <име> ще се свързва със стойности от <тип>
- **типовите декларации са незадължителни: в повечето случаи Haskell може сам да се ориентира за правилния тип**
 - x :: Int
 - y :: Double
 - z :: String
- <име> = <израз> (дефиниция)
- <име> се свърза с <израз>
 - x = 2
 - y = fromIntegral x^2 + 7.5
 - z = "Hello"
 - ~~z = x + y~~

Типове

Типовете в Haskell обикновено се задават с конструктори.

Типове

Типовете в Haskell обикновено се задават с конструктори.

- `Bool` — булев тип с константи `True` и `False`

Типове

Типовете в Haskell обикновено се задават с конструктори.

- `Bool` — булев тип с константи `True` и `False`
- `Char` — Unicode знаци

Типове

Типовете в Haskell обикновено се задават с конструктори.

- `Bool` — булев тип с константи `True` и `False`
- `Char` — Unicode знаци
- Целочислени

Типове

Типовете в Haskell обикновено се задават с конструктори.

- `Bool` — булев тип с константи `True` и `False`
- `Char` — Unicode знаци
- Целочислени
 - `Int` — цели числа с фиксирана големина $[-2^{63}; 2^{63} - 1]$

Типове

Типовете в Haskell обикновено се задават с конструктори.

- `Bool` — булев тип с константи `True` и `False`
- `Char` — Unicode знаци
- Целочислени
 - `Int` — цели числа с фиксирана големина $[-2^{63}; 2^{63} - 1]$
 - `Integer` — цели числа с произволна големина

Типове

Типовете в Haskell обикновено се задават с конструктори.

- `Bool` — булев тип с константи `True` и `False`
- `Char` — Unicode знаци
- Целочислени
 - `Int` — цели числа с фиксирана големина $[-2^{63}; 2^{63} - 1]$
 - `Integer` — цели числа с произволна големина
- С плаваща запетая

Типове

Типовете в Haskell обикновено се задават с конструктори.

- `Bool` — булев тип с константи `True` и `False`
- `Char` — Unicode знаци
- Целочислени
 - `Int` — цели числа с фиксирана големина $[-2^{63}; 2^{63} - 1]$
 - `Integer` — цели числа с произволна големина
- С плаваща запетая
 - `Float` — дробни числа с единична точност

Типове

Типовете в Haskell обикновено се задават с конструктори.

- `Bool` — булев тип с константи `True` и `False`
- `Char` — Unicode знаци
- Целочислени
 - `Int` — цели числа с фиксирана големина $[-2^{63}; 2^{63} - 1]$
 - `Integer` — цели числа с произволна големина
- С плаваща запетая
 - `Float` — дробни числа с единична точност
 - `Double` — дробни числа с двойна точност

Типове

Типовете в Haskell обикновено се задават с конструктори.

- `Bool` — булев тип с константи `True` и `False`
- `Char` — Unicode знаци
- Целочислени
 - `Int` — цели числа с фиксирана големина $[-2^{63}; 2^{63} - 1]$
 - `Integer` — цели числа с произволна големина
- С плаваща запетая
 - `Float` — дробни числа с единична точност
 - `Double` — дробни числа с двойна точност
- Съставни

Типове

Типовете в Haskell обикновено се задават с конструктори.

- `Bool` — булев тип с константи `True` и `False`
- `Char` — Unicode знаци
- Целочислени
 - `Int` — цели числа с фиксирана големина $[-2^{63}; 2^{63} - 1]$
 - `Integer` — цели числа с произволна големина
- С плаваща запетая
 - `Float` — дробни числа с единична точност
 - `Double` — дробни числа с двойна точност
- Съставни
 - `[a]` — тип списък с **произволна** дължина и елементи от **фиксиран** тип `a`

Типове

Типовете в Haskell обикновено се задават с конструктори.

- `Bool` — булев тип с константи `True` и `False`
- `Char` — Unicode знаци
- Целочислени
 - `Int` — цели числа с фиксирана големина $[-2^{63}; 2^{63} - 1]$
 - `Integer` — цели числа с произволна големина
- С плаваща запетая
 - `Float` — дробни числа с единична точност
 - `Double` — дробни числа с двойна точност
- Съставни
 - `[a]` — тип списък с **произволна** дължина и елементи от **фиксирани** тип `a`
 - `String = [Char]` — низ (списък от знаци)

Типове

Типовете в Haskell обикновено се задават с конструктори.

- `Bool` — булев тип с константи `True` и `False`
- `Char` — Unicode знаци
- Целочислени
 - `Int` — цели числа с фиксирана големина $[-2^{63}; 2^{63} - 1]$
 - `Integer` — цели числа с произволна големина
- С плаваща запетая
 - `Float` — дробни числа с единична точност
 - `Double` — дробни числа с двойна точност
- Съставни
 - `[a]` — тип списък с **произволна** дължина и елементи от **фиксирани** тип `a`
 - `String = [Char]` — низ (списък от знаци)
 - `(a, b, c)` — тип кортеж (наредена n -торка) с **фиксирана** дължина и **произволни** типове на компонентите

Стандартен модул Prelude

- програмите в Haskell се разделят на модули

Стандартен модул Prelude

- програмите в Haskell се разделят на модули
- `module <име> where`

Стандартен модул Prelude

- програмите в Haskell се разделят на модули
- `module <име> where`
- дефинира модул с `<име>`

Стандартен модул Prelude

- програмите в Haskell се разделят на модули
- `module <име> where`
- дефинира модул с `<име>`
- `import <модул> [<име>{,<име>}]`

Стандартен модул Prelude

- програмите в Haskell се разделят на модули
- `module <име> where`
- дефинира модул с `<име>`
- `import <модул> [<име>{,<име>}]`
- внася дефинициите `<име>` от `<модул>`

Стандартен модул Prelude

- програмите в Haskell се разделят на модули
- `module <име> where`
- дефинира модул с `<име>`
- `import <модул> [<име>{,<име>}]`
- внася дефинициите `<име>` от `<модул>`
- ако `<име>` не е указано, внася всички дефиниции

Стандартен модул Prelude

- програмите в Haskell се разделят на модули
- `module <име> where`
- дефинира модул с `<име>`
- `import <модул> [<име>{,<име>}]`
- внася дефинициите `<име>` от `<модул>`
- ако `<име>` не е указано, внася всички дефиниции
- стандартната библиотека в Haskell се съдържа в модула `Prelude`

Стандартен модул Prelude

- програмите в Haskell се разделят на модули
- `module <име> where`
- дефинира модул с `<име>`
- `import <модул> [<име>{,<име>}]`
- внася дефинициите `<име>` от `<модул>`
- ако `<име>` не е указано, внася всички дефиниции
- стандартната библиотека в Haskell се съдържа в модула `Prelude`
- всички дефиниции от `Prelude` се внасят автоматично във всяко програма

Стандартни числови функции

Аритметични операции

`+, -, *, /, ^, ^^`

Други числови функции

`div, mod, max, min, gcd, lcm`

Функции за преобразуване

`fromIntegral, fromInteger, toInteger, realToFrac, fromRational,
toRational, round, ceiling, floor`

Функции над дробни числа

`exp, log, sin, cos, tan, asin, acos, atan, sqrt, **`

Стандартни предикати

Числови предикати

`<, >, ==, /=, <=, >=, odd, even`

Булеви операции

`&&, ||, not`

Функции в Haskell

- $t_1 \rightarrow t_2$ — тип на функция, която получава параметър от тип t_1 и връща резултат от тип t_2

Функции в Haskell

- $t_1 \rightarrow t_2$ — тип на функция, която получава параметър от тип t_1 и връща резултат от тип t_2
- <име> <параметър> = <тяло>

Функции в Haskell

- $t_1 \rightarrow t_2$ — тип на функция, която получава параметър от тип t_1 и връща резултат от тип t_2
- $\langle \text{име} \rangle \langle \text{параметър} \rangle = \langle \text{тяло} \rangle$
- дефиниция на функция с $\langle \text{име} \rangle$, един $\langle \text{параметър} \rangle$ и $\langle \text{тяло} \rangle$

Функции в Haskell

- $t_1 \rightarrow t_2$ — тип на функция, която получава параметър от тип t_1 и връща резултат от тип t_2
- $\langle \text{име} \rangle \langle \text{параметър} \rangle = \langle \text{тяло} \rangle$
- дефиниция на функция с $\langle \text{име} \rangle$, един $\langle \text{параметър} \rangle$ и $\langle \text{тяло} \rangle$
- $\langle \text{функция} \rangle \langle \text{израз} \rangle$

Функции в Haskell

- $t_1 \rightarrow t_2$ — тип на функция, която получава параметър от тип t_1 и връща резултат от тип t_2
- <име> <параметър> = <тяло>
- дефиниция на функция с <име>, един <параметър> и <тяло>
- <функция> <израз>
- прилагане на <функция> над <израз>
 - `square :: Int -> Int`
 - `square x = x * x`
 - `square x → 4`
 - `square 2.7 → Грешка!`

Функции в Haskell

- $t_1 \rightarrow t_2$ — тип на функция, която получава параметър от тип t_1 и връща резултат от тип t_2
- <име> <параметър> = <тяло>
- дефиниция на функция с <име>, един <параметър> и <тяло>
- <функция> <израз>
- прилагане на <функция> над <израз>
 - square :: Int \rightarrow Int
 - square x = x * x
 - square x \longrightarrow 4
 - square 2.7 \longrightarrow Грешка!
- Прилагането е с по-висок приоритет от другите операции!

Функции в Haskell

- $t_1 \rightarrow t_2$ — тип на функция, която получава параметър от тип t_1 и връща резултат от тип t_2
- <име> <параметър> = <тяло>
- дефиниция на функция с <име>, един <параметър> и <тяло>
- <функция> <израз>
- прилагане на <функция> над <израз>
 - square :: Int \rightarrow Int
 - square x = x * x
 - square x \longrightarrow 4
 - square 2.7 \longrightarrow Грешка!
- Прилагането е с по-висок приоритет от другите операции!
- $(\text{square } 2) + 3 \longrightarrow 7$

Функции в Haskell

- $t_1 \rightarrow t_2$ — тип на функция, която получава параметър от тип t_1 и връща резултат от тип t_2
- <име> <параметър> = <тяло>
- дефиниция на функция с <име>, един <параметър> и <тяло>
- <функция> <израз>
- прилагане на <функция> над <израз>
 - square :: Int \rightarrow Int
 - square x = x * x
 - square x \longrightarrow 4
 - square 2.7 \longrightarrow Грешка!
- Прилагането е с по-висок приоритет от другите операции!
- square 2 + 3 \longrightarrow 7
- square (2 + 3) \longrightarrow 25

Функции на повече параметри

- Как можем да изразим двуаргументна функция $f(x, y)$, ако разполагаме само с едноаргументни функции?

$\exists \ldots = x \mid \exists_1(\exists_2) \mid \lambda_x \in$

Функции на повече параметри

- Как можем да изразим двуаргументна функция $f(x, y)$, ако разполагаме само с едноаргументни функции?
- Разглеждаме функция F с един аргумент x, \dots

Функции на повече параметри

- Как можем да изразим двуаргументна функция $f(x, y)$, ако разполагаме само с едноаргументни функции?
- Разглеждаме функция F с един аргумент x, \dots
- ...която връща като резултат едноаргументната f_x, \dots

Функции на повече параметри

- Как можем да изразим двуаргументна функция $f(x, y)$, ако разполагаме само с едноаргументни функции?
- Разглеждаме функция F с един аргумент x, \dots
- ...която връща като резултат едноаргументната f_x, \dots
- ...така че $f_x(y) = f(x, y)$.

$$\frac{\partial f}{\partial y}(x, y) = f_x'(y)$$

Функции на повече параметри

- Как можем да изразим двуаргументна функция $f(x, y)$, ако разполагаме само с едноаргументни функции?
- Разглеждаме функция F с един аргумент x, \dots
- ...която връща като резултат едноаргументната f_x, \dots
- ...така че $f_x(y) = f(x, y)$.
- Така имаме $f(x, y) = F(x)(y)$.

Функции на повече параметри

- Как можем да изразим двуаргументна функция $f(x, y)$, ако разполагаме само с едноаргументни функции?
- Разглеждаме функция F с един аргумент x, \dots
- ...която връща като резултат едноаргументната f_x, \dots
- ...така че $f_x(y) = f(x, y)$.
- Така имаме $f(x, y) = F(x)(y)$.

Основна идея

Можем да разглеждаме функция с $n + 1$ аргумента, като функция с един аргумент, която връща функция с n аргумента.

Функции на повече параметри

- Как можем да изразим двуаргументна функция $f(x, y)$, ако разполагаме само с едноаргументни функции?
- Разглеждаме функция F с един аргумент x, \dots
- ...която връща като резултат едноаргументната f_x, \dots
- ...така че $f_x(y) = f(x, y)$.
- Така имаме $f(x, y) = F(x)(y)$.

Основна идея

Можем да разглеждаме функция с $n + 1$ аргумента, като функция с един аргумент, която връща функция с n аргумента.

Това представяне на функциите с повече аргументи се нарича “къринг” (“currying”).

Currying в Haskell

- $t1 \rightarrow (t2 \rightarrow t3)$

Currying в Haskell

- $t1 \rightarrow (t2 \rightarrow t3)$
 - функция с параметър от тип $t1$, която връща функция, която приема параметър от тип $t2$ и връща резултат от тип $t3$; или

Currying в Haskell

- $t1 \rightarrow (t2 \rightarrow t3)$
 - функция с параметър от тип $t1$, която връща функция, която приема параметър от тип $t2$ и връща резултат от тип $t3$; или
 - функция на два параметъра от типове $t1$ и $t2$, която връща резултат от тип $t3$

Currying в Haskell

- $t_1 \rightarrow (t_2 \rightarrow t_3)$
 - функция с параметър от тип t_1 , която връща функция, която приема параметър от тип t_2 и връща резултат от тип t_3 ; или
 - функция на два параметъра от типове t_1 и t_2 , която връща резултат от тип t_3
- В общия случай: <функция> :: $t_1 \rightarrow (t_2 \rightarrow \dots (t_n \rightarrow t) \dots)$

Currying в Haskell

- $t_1 \rightarrow (t_2 \rightarrow t_3)$
 - функция с параметър от тип t_1 , която връща функция, която приема параметър от тип t_2 и връща резултат от тип t_3 ; или
 - функция на два параметъра от типове t_1 и t_2 , която връща резултат от тип t_3
- В общия случай: <функция> :: $t_1 \rightarrow (t_2 \rightarrow \dots (t_n \rightarrow t) \dots)$
- <функция> ще очаква n параметъра от типове t_1, t_2, \dots, t_n и ще връща резултат от тип t

Currying в Haskell

- $t_1 \rightarrow (t_2 \rightarrow t_3)$
 - функция с параметър от тип t_1 , която връща функция, която приема параметър от тип t_2 и връща резултат от тип t_3 ; или
 - функция на два параметъра от типове t_1 и t_2 , която връща резултат от тип t_3
- В общия случай: <функция> :: $t_1 \rightarrow (t_2 \rightarrow \dots (t_n \rightarrow t) \dots)$
- <функция> ще очаква n параметъра от типове t_1, t_2, \dots, t_n и ще връща резултат от тип t
- <функция> <параметър₁> ... <параметър _{n} > = <тяло>

Currying в Haskell

- $t_1 \rightarrow (t_2 \rightarrow t_3)$
 - функция с параметър от тип t_1 , която връща функция, която приема параметър от тип t_2 и връща резултат от тип t_3 ; или
 - функция на два параметъра от типове t_1 и t_2 , която връща резултат от тип t_3
- В общия случай: <функция> :: $t_1 \rightarrow (t_2 \rightarrow \dots (t_n \rightarrow t) \dots)$
- <функция> ще очаква n параметъра от типове t_1, t_2, \dots, t_n и ще връща резултат от тип t
- <функция> <параметър₁> ... <параметър _{n} > = <тяло>
- дефинира <функция> с n параметъра и <тяло>

Currying в Haskell

- $t_1 \rightarrow (t_2 \rightarrow t_3)$
 - функция с параметър от тип t_1 , която връща функция, която приема параметър от тип t_2 и връща резултат от тип t_3 ; или
 - функция на два параметъра от типове t_1 и t_2 , която връща резултат от тип t_3
- В общия случай: <функция> :: $t_1 \rightarrow (t_2 \rightarrow \dots (t_n \rightarrow t) \dots)$
- <функция> ще очаква n параметъра от типове t_1, t_2, \dots, t_n и ще връща резултат от тип t
- <функция> <параметър₁> ... <параметър _{n} > = <тяло>
- дефинира <функция> с n параметъра и <тяло>
 - `hypotenuse :: Double -> (Double -> Double)`

Currying в Haskell

- $t_1 \rightarrow (t_2 \rightarrow t_3)$
 - функция с параметър от тип t_1 , която връща функция, която приема параметър от тип t_2 и връща резултат от тип t_3 ; или
 - функция на два параметъра от типове t_1 и t_2 , която връща резултат от тип t_3
- В общия случай: <функция> :: $t_1 \rightarrow (t_2 \rightarrow \dots (t_n \rightarrow t) \dots)$
- <функция> ще очаква n параметъра от типове t_1, t_2, \dots, t_n и ще връща резултат от тип t
- <функция> <параметър₁> ... <параметър _{n} > = <тяло>
- дефинира <функция> с n параметъра и <тяло>
 - `hypotenuse :: Double -> Double -> Double`

Currying в Haskell

- $t_1 \rightarrow (t_2 \rightarrow t_3)$
 - функция с параметър от тип t_1 , която връща функция, която приема параметър от тип t_2 и връща резултат от тип t_3 ; или
 - функция на два параметъра от типове t_1 и t_2 , която връща резултат от тип t_3
- В общия случай: <функция> :: $t_1 \rightarrow (t_2 \rightarrow \dots (t_n \rightarrow t) \dots)$
- <функция> ще очаква n параметъра от типове t_1, t_2, \dots, t_n и ще връща резултат от тип t
- <функция> <параметър₁> ... <параметър _{n} > = <тяло>
- дефинира <функция> с n параметъра и <тяло>
 - `hypotenuse :: Double -> Double -> Double`
 - `hypotenuse a b = sqrt (a**2 + b**2)`

Currying в Haskell

- $t_1 \rightarrow (t_2 \rightarrow t_3)$
 - функция с параметър от тип t_1 , която връща функция, която приема параметър от тип t_2 и връща резултат от тип t_3 ; или
 - функция на два параметъра от типове t_1 и t_2 , която връща резултат от тип t_3
- В общия случай: <функция> :: $t_1 \rightarrow (t_2 \rightarrow \dots (t_n \rightarrow t) \dots)$
- <функция> ще очаква n параметъра от типове t_1, t_2, \dots, t_n и ще връща резултат от тип t
- <функция> <параметър₁> ... <параметър _{n} > = <тяло>
- дефинира <функция> с n параметъра и <тяло>
 - `hypotenuse :: Double -> Double -> Double`
 - `hypotenuse a b = sqrt (a**2 + b**2)`
 - `(hypotenuse 3) 4` → 5

Currying в Haskell

- $t_1 \rightarrow (t_2 \rightarrow t_3)$
 - функция с параметър от тип t_1 , която връща функция, която приема параметър от тип t_2 и връща резултат от тип t_3 ; или
 - функция на два параметъра от типове t_1 и t_2 , която връща резултат от тип t_3
- В общия случай: <функция> :: $t_1 \rightarrow (t_2 \rightarrow \dots (t_n \rightarrow t) \dots)$
- <функция> ще очаква n параметъра от типове t_1, t_2, \dots, t_n и ще връща резултат от тип t
- <функция> <параметър₁> ... <параметър _{n} > = <тяло>
- дефинира <функция> с n параметъра и <тяло>
 - `hypotenuse :: Double -> Double -> Double`
 - `hypotenuse a b = sqrt (a**2 + b**2)`
 - `hypotenuse 3 4 → 5`

Частично прилагане на функции

Кърингът позволява удобно прилагане на функция към само част от параметрите.

- `div50 :: Int -> Int`

Частично прилагане на функции

Кърингът позволява удобно прилагане на функция към само част от параметрите.

- `div50 :: Int -> Int`
- `div50 x = div 50 x`

Частично прилагане на функции

Кърингт позволява удобно прилагане на функция към само част от параметрите.

- `div50 :: Int -> Int`
- `div50 x = (div 50) x`
- `div50 4 → 12`

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

Частично прилагане на функции

Кърингът позволява удобно прилагане на функция към само част от параметрите.

- `div50 :: Int -> Int`
- `div50 x = div 50 x`
- `div50 4 —> 12`

Частично прилагане на функции

Кърингът позволява удобно прилагане на функция към само част от параметрите.

- `div50 :: Int -> Int`
- `div50 = div 50`
- `div50 4 —> 12`

Функции от по-висок ред

Внимание: $t1 \rightarrow (t2 \rightarrow t3) \neq (t1 \rightarrow t2) \rightarrow t3!$

Функции от по-висок ред

Внимание: $t1 \rightarrow (t2 \rightarrow t3) \neq (t1 \rightarrow t2) \rightarrow t3!$

- операцията \rightarrow е дясноассоциативна

Функции от по-висок ред

Внимание: $t1 \rightarrow (t2 \rightarrow t3) \neq (t1 \rightarrow t2) \rightarrow t3!$

- операцията \rightarrow е **дясноассоциативна**
- $t1 \rightarrow (t2 \rightarrow t3) \equiv t1 \rightarrow t2 \rightarrow t3$

Функции от по-висок ред

Внимание: $t1 \rightarrow (t2 \rightarrow t3) \neq (t1 \rightarrow t2) \rightarrow t3!$

- операцията \rightarrow е **дясноасоциативна**
- $t1 \rightarrow (t2 \rightarrow t3) \equiv t1 \rightarrow t2 \rightarrow t3$
- $(t1 \rightarrow t2) \rightarrow t3$ — функция, която връща резултат от тип $t3$, а приема като единствен параметър функция, която приема един параметър от тип $t1$ и връща резултат от тип $t2$

Функции от по-висок ред

Внимание: $t1 \rightarrow (t2 \rightarrow t3) \neq (t1 \rightarrow t2) \rightarrow t3!$

- операцията \rightarrow е **дясноасоциативна**
- $t1 \rightarrow (t2 \rightarrow t3) \equiv t1 \rightarrow t2 \rightarrow t3$
- $(t1 \rightarrow t2) \rightarrow t3$ — функция, която връща резултат от тип $t3$, а приема като единствен параметър функция, която приема един параметър от тип $t1$ и връща резултат от тип $t2$
- **функция от втори ред**

Функции от по-висок ред

Внимание: $t_1 \rightarrow (t_2 \rightarrow t_3) \neq (t_1 \rightarrow t_2) \rightarrow t_3!$

- операцията \rightarrow е **дясноасоциативна**
- $t_1 \rightarrow (t_2 \rightarrow t_3) \equiv t_1 \rightarrow t_2 \rightarrow t_3$
- $(t_1 \rightarrow t_2) \rightarrow t_3$ — функция, която връща резултат от тип t_3 , а приема като единствен параметър функция, която приема един параметър от тип t_1 и връща резултат от тип t_2
- **функция от втори ред**
 - `twice f x = f (f x)`

Функции от по-висок ред

Внимание: $t_1 \rightarrow (t_2 \rightarrow t_3) \neq (t_1 \rightarrow t_2) \rightarrow t_3!$

- операцията \rightarrow е **дясноасоциативна**
- $t_1 \rightarrow (t_2 \rightarrow t_3) \equiv t_1 \rightarrow t_2 \rightarrow t_3$
- $(t_1 \rightarrow t_2) \rightarrow t_3$ — функция, която връща резултат от тип t_3 , а приема като единствен параметър функция, която приема един параметър от тип t_1 и връща резултат от тип t_2
- **функция от втори ред**
 - `twice f x = f (f x)`
 - `twice :: (Int -> Int) -> Int -> Int`

Функции от по-висок ред

Внимание: $t_1 \rightarrow (t_2 \rightarrow t_3) \neq (t_1 \rightarrow t_2) \rightarrow t_3!$

- операцията \rightarrow е **дясноасоциативна**
- $t_1 \rightarrow (t_2 \rightarrow t_3) \equiv t_1 \rightarrow t_2 \rightarrow t_3$
- $(t_1 \rightarrow t_2) \rightarrow t_3$ — функция, която връща резултат от тип t_3 , а приема като единствен параметър функция, която приема един параметър от тип t_1 и връща резултат от тип t_2
- **функция от втори ред**
 - `twice f x = f (f x)`
 - `twice :: (Int -> Int) -> Int -> Int`
 - `twice square 3` — ?

Функции от по-висок ред

Внимание: $t_1 \rightarrow (t_2 \rightarrow t_3) \neq (t_1 \rightarrow t_2) \rightarrow t_3!$

- операцията \rightarrow е **дясноасоциативна**
- $t_1 \rightarrow (t_2 \rightarrow t_3) \equiv t_1 \rightarrow t_2 \rightarrow t_3$
- $(t_1 \rightarrow t_2) \rightarrow t_3$ — функция, която връща резултат от тип t_3 , а приема като единствен параметър функция, която приема един параметър от тип t_1 и връща резултат от тип t_2
- **функция от втори ред**
 - `twice f x = f (f x)`
 - `twice :: (Int -> Int) -> Int -> Int`
 - `twice square 3` \longrightarrow 81

Функции от по-висок ред

Внимание: $t_1 \rightarrow (t_2 \rightarrow t_3) \neq (t_1 \rightarrow t_2) \rightarrow t_3!$

- операцията \rightarrow е **дясноасоциативна**
- $t_1 \rightarrow (t_2 \rightarrow t_3) \equiv t_1 \rightarrow t_2 \rightarrow t_3$
- $(t_1 \rightarrow t_2) \rightarrow t_3$ — функция, която връща резултат от тип t_3 , а приема като единствен параметър функция, която приема един параметър от тип t_1 и връща резултат от тип t_2
- **функция от втори ред**
 - `twice f x = f (f x)`
 - `twice :: (Int -> Int) -> Int -> Int`
 - `twice square 3` \longrightarrow 81
 - `twice (mod 13) 5` \longrightarrow ?

Функции от по-висок ред

Внимание: $t_1 \rightarrow (t_2 \rightarrow t_3) \neq (t_1 \rightarrow t_2) \rightarrow t_3!$

- операцията \rightarrow е **дясноасоциативна**
- $t_1 \rightarrow (t_2 \rightarrow t_3) \equiv t_1 \rightarrow t_2 \rightarrow t_3$
- $(t_1 \rightarrow t_2) \rightarrow t_3$ — функция, която връща резултат от тип t_3 , а приема като единствен параметър функция, която приема един параметър от тип t_1 и връща резултат от тип t_2
- **функция от втори ред**
 - `twice f x = f (f x)`
 - `twice :: (Int -> Int) -> Int -> Int`
 - `twice square 3` \longrightarrow 81
 - `twice (mod 13) 5` \longrightarrow 1

Функции от по-висок ред

Внимание: $t_1 \rightarrow (t_2 \rightarrow t_3) \neq (t_1 \rightarrow t_2) \rightarrow t_3!$

- операцията \rightarrow е **дясноасоциативна**
- $t_1 \rightarrow (t_2 \rightarrow t_3) \equiv t_1 \rightarrow t_2 \rightarrow t_3$
- $(t_1 \rightarrow t_2) \rightarrow t_3$ — функция, която връща резултат от тип t_3 , а приема като единствен параметър функция, която приема един параметър от тип t_1 и връща резултат от тип t_2
- **функция от втори ред**
 - `twice f x = f (f x)`
 - `twice :: (t -> t) -> t -> t`
 - `twice square 3` \longrightarrow 81
 - `twice (mod 13) 5` \longrightarrow 1

Функции от по-висок ред

Внимание: $t_1 \rightarrow (t_2 \rightarrow t_3) \neq (t_1 \rightarrow t_2) \rightarrow t_3!$

- операцията \rightarrow е **дясноассоциативна**
- $t_1 \rightarrow (t_2 \rightarrow t_3) \equiv t_1 \rightarrow t_2 \rightarrow t_3$
- $(t_1 \rightarrow t_2) \rightarrow t_3$ — функция, която връща резултат от тип t_3 , а приема като единствен параметър функция, която приема един параметър от тип t_1 и връща резултат от тип t_2
- **функция от втори ред**
 - `twice f x = f (f x)`
 - `twice :: (t -> t) -> t -> t`
 - `twice square 3` \longrightarrow 81
 - `twice (mod 13) 5` \longrightarrow 1
 - `diag f x = f x x`

Функции от по-висок ред

Внимание: $t_1 \rightarrow (t_2 \rightarrow t_3) \neq (t_1 \rightarrow t_2) \rightarrow t_3!$

- операцията \rightarrow е **дясноасоциативна**
- $t_1 \rightarrow (t_2 \rightarrow t_3) \equiv t_1 \rightarrow t_2 \rightarrow t_3$
- $(t_1 \rightarrow t_2) \rightarrow t_3$ — функция, която връща резултат от тип t_3 , а приема като единствен параметър функция, която приема един параметър от тип t_1 и връща резултат от тип t_2
- **функция от втори ред**
 - `twice f x = f (f x)`
 - `twice :: (t -> t) -> t -> t`
 - `twice square 3` \longrightarrow 81
 - `twice (mod 13) 5` \longrightarrow 1
 - `diag f x = f x x`
 - `diag :: (Int -> Int -> Int) -> Int -> Int`

Функции от по-висок ред

Внимание: $t_1 \rightarrow (t_2 \rightarrow t_3) \neq (t_1 \rightarrow t_2) \rightarrow t_3!$

- операцията \rightarrow е **дясноасоциативна**
- $t_1 \rightarrow (t_2 \rightarrow t_3) \equiv t_1 \rightarrow t_2 \rightarrow t_3$
- $(t_1 \rightarrow t_2) \rightarrow t_3$ — функция, която връща резултат от тип t_3 , а приема като единствен параметър функция, която приема един параметър от тип t_1 и връща резултат от тип t_2
- **функция от втори ред**
 - `twice f x = f (f x)`
 - `twice :: (t -> t) -> t -> t`
 - `twice square 3` \longrightarrow 81
 - `twice (mod 13) 5` \longrightarrow 1
 - `diag f x = f x x`
 - `diag :: (Int -> Int -> Int) -> Int -> Int`
 - `diag div 5` \longrightarrow ?

Функции от по-висок ред

Внимание: $t_1 \rightarrow (t_2 \rightarrow t_3) \neq (t_1 \rightarrow t_2) \rightarrow t_3!$

- операцията \rightarrow е **дясноасоциативна**
- $t_1 \rightarrow (t_2 \rightarrow t_3) \equiv t_1 \rightarrow t_2 \rightarrow t_3$
- $(t_1 \rightarrow t_2) \rightarrow t_3$ — функция, която връща резултат от тип t_3 , а приема като единствен параметър функция, която приема един параметър от тип t_1 и връща резултат от тип t_2
- **функция от втори ред**
 - `twice f x = f (f x)`
 - `twice :: (t -> t) -> t -> t`
 - `twice square 3` \longrightarrow 81
 - `twice (mod 13) 5` \longrightarrow 1
 - `diag f x = f x x`
 - `diag :: (Int -> Int -> Int) -> Int -> Int`
 - `diag div 5` \longrightarrow 1

Функции от по-висок ред

Внимание: $t_1 \rightarrow (t_2 \rightarrow t_3) \neq (t_1 \rightarrow t_2) \rightarrow t_3!$

- операцията \rightarrow е **дясноасоциативна**
- $t_1 \rightarrow (t_2 \rightarrow t_3) \equiv t_1 \rightarrow t_2 \rightarrow t_3$
- $(t_1 \rightarrow t_2) \rightarrow t_3$ — функция, която връща резултат от тип t_3 , а приема като единствен параметър функция, която приема един параметър от тип t_1 и връща резултат от тип t_2
- **функция от втори ред**
 - `twice f x = f (f x)`
 - `twice :: (t -> t) -> t -> t`
 - `twice square 3` \longrightarrow 81
 - `twice (mod 13) 5` \longrightarrow 1
 - `diag f x = f x x`
 - `diag :: (Int -> Int -> Int) -> Int -> Int`
 - `diag div 5` \longrightarrow 1
 - `diag hypotenuse 1` \longrightarrow ?

Функции от по-висок ред

Внимание: $t_1 \rightarrow (t_2 \rightarrow t_3) \neq (t_1 \rightarrow t_2) \rightarrow t_3!$

- операцията \rightarrow е **дясноасоциативна**
- $t_1 \rightarrow (t_2 \rightarrow t_3) \equiv t_1 \rightarrow t_2 \rightarrow t_3$
- $(t_1 \rightarrow t_2) \rightarrow t_3$ — функция, която връща резултат от тип t_3 , а приема като единствен параметър функция, която приема един параметър от тип t_1 и връща резултат от тип t_2
- **функция от втори ред**
 - `twice f x = f (f x)`
 - `twice :: (t -> t) -> t -> t`
 - `twice square 3` \longrightarrow 81
 - `twice (mod 13) 5` \longrightarrow 1
 - `diag f x = f x x`
 - `diag :: (Int -> Int -> Int) -> Int -> Int`
 - `diag div 5` \longrightarrow 1
 - `diag hypotenuse 1` \longrightarrow 1.4142135623730951

Функции от по-висок ред

Внимание: $t_1 \rightarrow (t_2 \rightarrow t_3) \neq (t_1 \rightarrow t_2) \rightarrow t_3!$

- операцията \rightarrow е **дясноасоциативна**
- $t_1 \rightarrow (t_2 \rightarrow t_3) \equiv t_1 \rightarrow t_2 \rightarrow t_3$
- $(t_1 \rightarrow t_2) \rightarrow t_3$ — функция, която връща резултат от тип t_3 , а приема като единствен параметър функция, която приема един параметър от тип t_1 и връща резултат от тип t_2
- **функция от втори ред**
 - `twice f x = f (f x)`
 - `twice :: (t -> t) -> t -> t`
 - `twice square 3` \longrightarrow 81
 - `twice (mod 13) 5` \longrightarrow 1
 - `diag f x = f x x`
 - `diag :: (t1 -> t1 -> t) -> t1 -> t`
 - `diag div 5` \longrightarrow 1
 - `diag hypotenuse 1` \longrightarrow 1.4142135623730951

Функции и операции

- Функциите в Haskell са винаги с префиксен запис

Функции и операции

- Функциите в Haskell са винаги с префиксен запис
- Операциите в Haskell са винаги бинарни с инфиксен запис.

Функции и операции

- Функциите в Haskell са винаги с префиксен запис
- Операциите в Haskell са винаги бинарни с инфиксен запис.
 - **Изключение:** унарен минус: -a

Функции и операции

- Функциите в Haskell са винаги с префиксен запис
- Операциите в Haskell са винаги бинарни с инфиксен запис.
 - **Изключение:** унарен минус: `-a`
 - `square -x` → **Грешка!**

Функции и операции

- Функциите в Haskell са винаги с префиксен запис
- Операциите в Haskell са винаги бинарни с инфиксен запис.
 - **Изключение:** унарен минус: `-a`
 - `square -x` → **Грешка!**
 - `square (-x)` → 4

Функции и операции

- Функциите в Haskell са винаги с префиксен запис
- Операциите в Haskell са винаги бинарни с инфиксен запис.
 - **Изключение:** унарен минус: `-a`
 - `square -x` → **Грешка!**
 - `square (-x)` → 4
- Преобразуване на двуаргументни функции към бинарни операции:
`'<функция>'`

Функции и операции

- Функциите в Haskell са винаги с префиксен запис
- Операциите в Haskell са винаги бинарни с инфиксен запис.
 - **Изключение:** унарен минус: `-a`
 - `square -x` → **Грешка!**
 - `square (-x)` → 4
- Преобразуване на двуаргументни функции към бинарни операции:
`'<функция>'`
 - `13 `div` 5` → 3

Функции и операции

- Функциите в Haskell са винаги с префиксен запис
- Операциите в Haskell са винаги бинарни с инфиксен запис.
 - **Изключение:** унарен минус: `-a`
 - `square -x` → **Грешка!**
 - `square (-x)` → 4
- Преобразуване на двуаргументни функции към бинарни операции:
`'<функция>'`
 - `13 `div` 5` → 3
 - `2 `square`` → **Грешка!**

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(<операция>)

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(*<операция>*)
 - (+) 2 3 → 5

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(*<операция>*)
 - $(+) \ 2 \ 3 \longrightarrow 5$
 - plus1 = $(+) \ 1$

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(*<операция>*)
 - $(+) \ 2 \ 3 \longrightarrow 5$
 - `plus1 = (+) 1`
 - `square = diag (*)`

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(*<операция>*)
 - $(+) \ 2 \ 3 \longrightarrow 5$
 - $\text{plus1} = (+) \ 1$
 - $\text{square} = \text{diag} \ (*)$
- Преобразуване на операции към едноаргументни функции
(отсичане на операции)

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(*<операция>*)
 - $(+) \ 2 \ 3 \longrightarrow 5$
 - $\text{plus1} = (+) \ 1$
 - $\text{square} = \text{diag} \ (*)$
- Преобразуване на операции към едноаргументни функции
(отсичане на операции)
 - (*<израз> <операция>*) — ляво отсичане

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(*<операция>*)
 - $(+) \ 2 \ 3 \longrightarrow 5$
 - $\text{plus1} = (+) \ 1$
 - $\text{square} = \text{diag} \ (*)$
- Преобразуване на операции към едноаргументни функции
(отсичане на операции)
 - (*<израз> <операция>*) — ляво отсичане
 - (*<операция> <израз>*) — дясно отсичане

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(<операция>)
 - $(+) \ 2 \ 3 \longrightarrow 5$
 - $\text{plus1} = (+) \ 1$
 - $\text{square} = \text{diag} \ (*)$
- Преобразуване на операции към едноаргументни функции
(отсичане на операции)
 - (*<израз> <операция>*) — ляво отсичане
 - (*<операция> <израз>*) — дясно отсичане
 - $(2^) \ 3 \longrightarrow ?$

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(<операция>)
 - $(+) \ 2 \ 3 \longrightarrow 5$
 - $\text{plus1} = (+) \ 1$
 - $\text{square} = \text{diag} \ (*)$
- Преобразуване на операции към едноаргументни функции
(отсичане на операции)
 - (*<израз> <операция>*) — ляво отсичане
 - (*<операция> <израз>*) — дясно отсичане
 - $(2^) \ 3 \longrightarrow 8$

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(<операция>)
 - $(+) \ 2 \ 3 \longrightarrow 5$
 - $\text{plus1} = (+) \ 1$
 - $\text{square} = \text{diag} \ (*)$
- Преобразуване на операции към едноаргументни функции
(отсичане на операции)
 - (*<израз> <операция>*) — ляво отсичане
 - (*<операция> <израз>*) — дясно отсичане
 - $(2^) \ 3 \longrightarrow 8$
 - $(^2) \ 3 \longrightarrow ?$

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(<операция>)
 - $(+) 2 3 \rightarrow 5$
 - $\text{plus1} = (+) 1$
 - $\text{square} = \text{diag} (*)$
- Преобразуване на операции към едноаргументни функции
(отсичане на операции)
 - (*<израз> <операция>*) — ляво отсичане
 - (*<операция> <израз>*) — дясно отсичане
 - $(2^) 3 \rightarrow 8$ 
 - $(^2) 3 \rightarrow 9$  

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(<операция>)
 - $(+) 2 3 \rightarrow 5$
 - $\text{plus1} = (+) 1$
 - $\text{square} = \text{diag} (*)$
- Преобразуване на операции към едноаргументни функции
(отсичане на операции)
 - (*<израз> <операция>*) — ляво отсичане
 - (*<операция> <израз>*) — дясно отсичане
 - $(2^) 3 \rightarrow 8$
 - $(^2) 3 \rightarrow 9$
 - $\text{square} = (^2)$

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(<операция>)
 - $(+) \ 2 \ 3 \longrightarrow 5$
 - $\text{plus1} = (+) \ 1$
 - $\text{square} = \text{diag} \ (*)$
- Преобразуване на операции към едноаргументни функции
(отсичане на операции)
 - (*<израз> <операция>*) — ляво отсичане
 - (*<операция> <израз>*) — дясно отсичане
 - $(2^) \ 3 \longrightarrow 8$
 - $(^2) \ 3 \longrightarrow 9$
 - $\text{square} = (^2)$
 - $(-5) \ 8 \longrightarrow ?$

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(<операция>)
 - $(+) 2 3 \rightarrow 5$
 - $\text{plus1} = (+) 1$
 - $\text{square} = \text{diag} (*)$
- Преобразуване на операции към едноаргументни функции
(отсичане на операции)
 - (*<израз> <операция>*) — ляво отсичане
 - (*<операция> <израз>*) — дясно отсичане
 - $(2^) 3 \rightarrow 8$
 - $(^2) 3 \rightarrow 9$
 - $\text{square} = (^2)$
 - $(-5) 8 \rightarrow \text{Грешка!}$

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(<операция>)
 - $(+) \ 2 \ 3 \longrightarrow 5$
 - $\text{plus1} = (+) \ 1$
 - $\text{square} = \text{diag} \ (*)$
- Преобразуване на операции към едноаргументни функции
(отсичане на операции)
 - (*<израз> <операция>*) — ляво отсичане
 - (*<операция> <израз>*) — дясно отсичане
 - $(2^) \ 3 \longrightarrow 8$
 - $(^2) \ 3 \longrightarrow 9$
 - $\text{square} = (^2)$
 - $(-5) \ 8 \longrightarrow \text{Грешка!}$
 - $\text{twice} \ (*2) \ 5 \longrightarrow ?$

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(<операция>)
 - $(+) 2 3 \rightarrow 5$
 - $\text{plus1} = (+) 1$
 - $\text{square} = \text{diag} (*)$
- Преобразуване на операции към едноаргументни функции
(отсичане на операции)
 - (<израз> <операция>) — ляво отсичане
 - (<операция> <израз>) — дясно отсичане
 - $(2^) 3 \rightarrow 8$
 - $(^2) 3 \rightarrow 9$
 - $\text{square} = (^2)$
 - $(-5) 8 \rightarrow \text{Грешка!}$
 - $\text{twice} (*2) 5 \rightarrow 20$

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(<операция>)
 - (+) 2 3 → 5
 - plus1 = (+) 1
 - square = diag (*)
- Преобразуване на операции към едноаргументни функции
(отсичане на операции)
 - (<израз> <операция>) — ляво отсичане
 - (<операция> <израз>) — дясно отсичане
 - ($2^$) 3 → 8
 - ($\wedge 2$) 3 → 9
 - square = ($\wedge 2$)
 - (-5) 8 → Грешка!
 - twice (*2) 5 → 20
 - positive = (>0)

Операции и функции

- Преобразуване на операции към двуаргументни функции:
(<операция>)
 - $(+) \ 2 \ 3 \longrightarrow 5$
 - $\text{plus1} = (+) \ 1$
 - $\text{square} = \text{diag} \ (*)$
- Преобразуване на операции към едноаргументни функции
(отсичане на операции)
 - (*<израз> <операция>*) — ляво отсичане
 - (*<операция> <израз>*) — дясно отсичане
 - $(2^) \ 3 \longrightarrow 8$
 - $(^2) \ 3 \longrightarrow 9$
 - $\text{square} = (^2)$
 - $(-5) \ 8 \longrightarrow \text{Грешка!}$
 - $\text{twice} \ (*2) \ 5 \longrightarrow 20$
 - $\text{positive} = (>0)$
 - $\text{lastDigit} = (\text{'mod'} \ 10)$

if ... then ... else

- **if** <условие> **then** <израз1> **else** <израз2>
 - Ако <условие> е **True**, връща <израз1>
 - Ако <условие> е **False**, връща <израз2>

if ... then ... else

- **if** <условие> **then** <израз1> **else** <израз2>
 - Ако <условие> е **True**, връща <израз1>
 - Ако <условие> е **False**, връща <израз2>
- **abs** x = **if** x < 0 **then** -x **else** x

if ... then ... else

- **if** <условие> **then** <израз1> **else** <израз2>
 - Ако <условие> е **True**, връща <израз1>
 - Ако <условие> е **False**, връща <израз2>
- **abs** x = **if** x < 0 **then** -x **else** x
- **fact** n = **if** n == 0 **then** 1 **else** n * **fact** (n - 1)

if ... then ... else

- **if** <условие> **then** <израз1> **else** <израз2>
 - Ако <условие> е **True**, връща <израз1>
 - Ако <условие> е **False**, връща <израз2>
- **abs** x = **if** x < 0 **then** -x **else** x
- **fact** n = **if** n == 0 **then** 1 **else** n * **fact** (n - 1)
- **if** x > 5 **then** x + 2 **else** "Error" → Грешка!

if ... then ... else

- **if** <условие> **then** <израз1> **else** <израз2>
 - Ако <условие> е **True**, връща <израз1>
 - Ако <условие> е **False**, връща <израз2>
- **abs** x = **if** x < 0 **then** -x **else** x
- **fact** n = **if** n == 0 **then** 1 **else** n * **fact** (n - 1)
- **if** x > 5 **then** x + 2 **else** "Error" → Грешка!
- <израз1> и <израз2> трябва да са от един и същи тип!

if ... then ... else

- **if <условие> then <израз1> else <израз2>**
 - Ако <условие> е **True**, връща <израз1>
 - Ако <условие> е **False**, връща <израз2>
- **abs x = if x < 0 then -x else x**
- **fact n = if n == 0 then 1 else n * fact (n - 1)**
- **if x > 5 then x + 2 else "Error" → Грешка!**
- **<израз1> и <израз2> трябва да са от един и същи тип!**
- **<условие> трябва да е от тип Bool!**