Основни понятия в Haskell

Трифон Трифонов

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

1 декември 2016 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> (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`
  - `z = "Hello"`

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`
  - `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 **фиксиран** 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 **фиксиран** 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>]

Стандартен модул `Prelude`

- програмите в Haskell се разделят на модули
- `module` <име> `where`
- дефинира модул с <име>
- `import` <модул>[`.`<име>]
- внася дефиницията <име> от <модул>

Стандартен модул `Prelude`

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

Стандартен модул `Prelude`

- програмите в Haskell се разделят на модули
- `module` <име> `where`
- дефинира модул с <име>
- `import` <модул>[`.`<име>]
- внася дефиницията <име> от <модул>
- ако <име> не е указана, 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>]
- внася дефиницията <име> от <модул>
- ако <име> не е указана, 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

- `t1 -> t2` — тип на функция, която получава параметър от тип `t1` и връща резултат от тип `t2`

Функции в Haskell

- `t1 -> t2` — тип на функция, която получава параметър от тип `t1` и връща резултат от тип `t2`
- <име> <параметър> = <тяло>

Функции в Haskell

- `t1 -> t2` — тип на функция, която получава параметър от тип `t1` и връща резултат от тип `t2`
- <име> <параметър> = <тяло>
- дефиниция на функция с <име>, един <параметър> и <тяло>

Функции в Haskell

- `t1 -> t2` — тип на функция, която получава параметър от тип `t1` и връща резултат от тип `t2`
- <име> <параметър> = <тяло>
- дефиниция на функция с <име>, един <параметър> и <тяло>
- <функция> <израз>

Функции в Haskell

- `t1 -> t2` — тип на функция, която получава параметър от тип `t1` и връща резултат от тип `t2`
- <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` $\longrightarrow$ `4`
  - `square 2.7` $\longrightarrow$ Грешка!

Функции в Haskell

- `t1 -> t2` — тип на функция, която получава параметър от тип `t1` и връща резултат от тип `t2`
- <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` $\longrightarrow$ `4`
  - `square 2.7` $\longrightarrow$ Грешка!
- Прилагането е с по-висок приоритет от другите операции!

Функции в Haskell

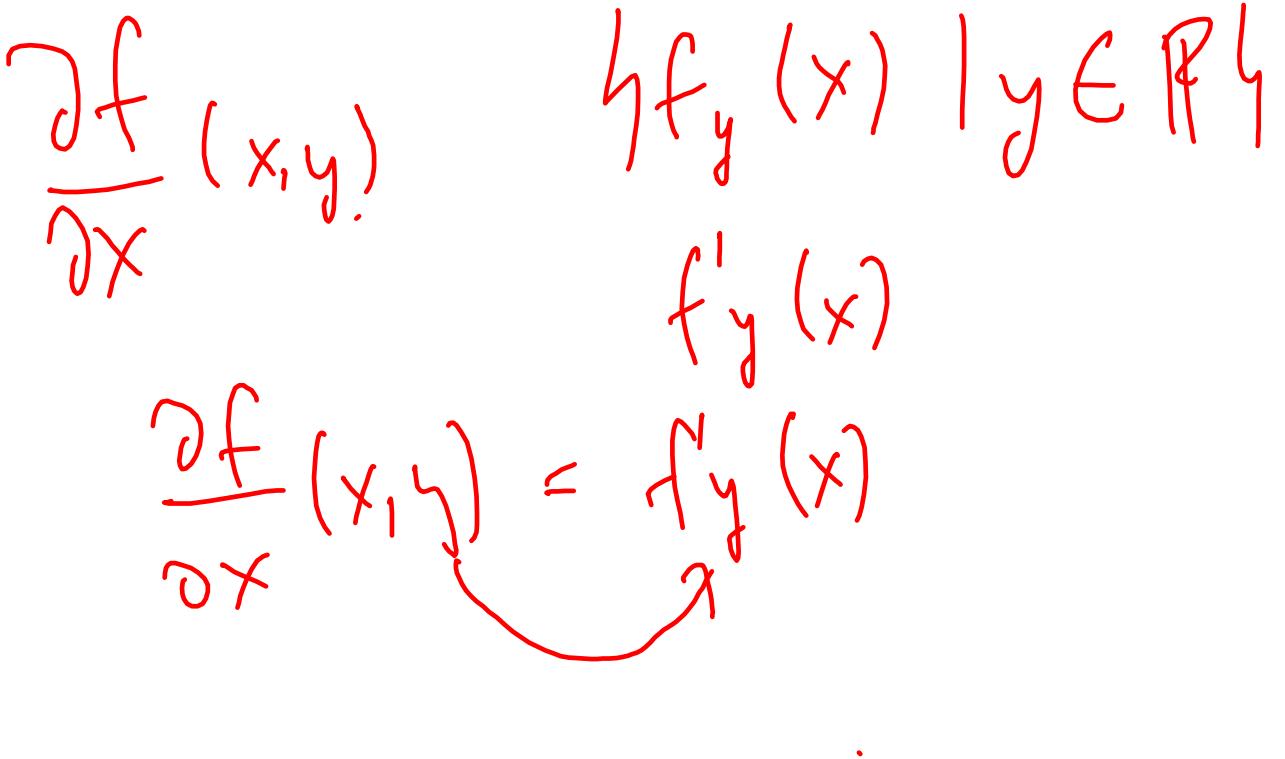
- `t1 -> t2` — тип на функция, която получава параметър от тип `t1` и връща резултат от тип `t2`
- <име> <параметър> = <тяло>
- дефиниция на функция с <име>, един <параметър> и <тяло>
- <функция> <израз>
- прилагане на <функция> над <израз>
  - `square :: Int -> Int`
  - `square x = x * x`
  - `square x` $\longrightarrow$ `4`
  - `square 2.7` $\longrightarrow$ Грешка!
- Прилагането е с по-висок приоритет от другите операции!
- `square 2 + 3` $\longrightarrow$ `7`

Функции в Haskell

- `t1 -> t2` — тип на функция, която получава параметър от тип `t1` и връща резултат от тип `t2`
- <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` $\longrightarrow$ `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?



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,...

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,...
- ...която връща като резултат едноаргументната f_x,...

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,...
- ...която връща като резултат едноаргументната f_x,...
- ...така че $f_x(y) = f(x, y)$.

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,...
- ...която връща като резултат едноаргументната f_x,...
- ...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,...
- ...която връща като резултат едноаргументната f_x,...
- ...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 функции 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,...
- ...която връща като резултат едноаргументната f_x,...
- ...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 функции с n аргумента.

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

Currying в Haskell

- `t1 -> (t2 -> t3)`

Currying в Haskell

- `t1 -> (t2 -> 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 -> (t2 -> 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

- `t1 -> (t2 -> 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`
- В общия случай: <функция> `:: t1 -> (t2 -> ...(tn -> t)...)`

Currying в Haskell

- `t1 -> (t2 -> 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`
- В общия случай: <функция> `:: t1 -> (t2 -> ...(tn -> t)...)`
- <функция> ще очаква n параметъра от типове `t1`, `t2`, ..., `tn` и ще връща резултат от тип `t`

Currying в Haskell

- `t1 -> (t2 -> 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`
- В общия случай: <функция> `::` `t1 -> (t2 -> ...(tn -> t)...)`
- <функция> ще очаква n параметъра от типове `t1`, `t2`, ..., `tn` и ще връща резултат от тип `t`
- <функция> <параметър$_1$> ... <параметър$_n$> `=` <тяло>

Currying в Haskell

- `t1 -> (t2 -> 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`
- В общия случай: <функция> `::` `t1 -> (t2 -> ...(tn -> t)...)`
- <функция> ще очаква n параметъра от типове `t1`, `t2`, ..., `tn` и ще връща резултат от тип `t`
- <функция> <параметър$_1$> ... <параметър$_n$> `=` <тяло>
- дефинира <функция> с n параметъра и <тяло>

Currying в Haskell

- `t1 -> (t2 -> 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`
- В общия случай: <функция> `::` `t1 -> (t2 -> ...(tn -> t)...)`
- <функция> ще очаква n параметъра от типове `t1`, `t2`, ..., `tn` и ще връща резултат от тип `t`
- <функция> <параметър$_1$> ... <параметър$_n$> `=` <тяло>
- дефинира <функция> с n параметъра и <тяло>
  - `hypothenuse :: Double -> (Double -> Double)`

Currying в Haskell

- `t1 -> (t2 -> 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`
- В общия случай: <функция> `::` `t1 -> (t2 -> ...(tn -> t)...)`
- <функция> ще очаква n параметъра от типове `t1`, `t2`, ..., `tn` и ще връща резултат от тип `t`
- <функция> <параметър$_1$> ... <параметър$_n$> `=` <тяло>
- дефинира <функция> с n параметъра и <тяло>
  - `hypothenuse :: Double -> Double -> Double`

Currying в Haskell

- `t1 -> (t2 -> 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`
- В общия случай: <функция> `::` `t1 -> (t2 -> ...(tn -> t)...)`
- <функция> ще очаква n параметъра от типове `t1`, `t2`, ..., `tn` и ще връща резултат от тип `t`
- <функция> <параметър$_1$> ... <параметър$_n$> `=` <тяло>
- дефинира <функция> с n параметъра и <тяло>
  - `hypothenuse :: Double -> Double -> Double`
  - `hypothenuse a b = sqrt (a**2 + b**2)`

Currying в Haskell

- `t1 -> (t2 -> 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`
- В общия случай: <функция> `::` `t1 -> (t2 -> ...(tn -> t)...)`
- <функция> ще очаква n параметъра от типове `t1`, `t2`, ..., `tn` и ще връща резултат от тип `t`
- <функция> <параметър$_1$> ... <параметър$_n$> `=` <тяло>
- дефинира <функция> с n параметъра и <тяло>
  - `hypothenuse :: Double -> Double -> Double`
  - `hypothenuse a b = sqrt (a**2 + b**2)`
  - `(hypothenuse 3) 4` $\longrightarrow$ `5`

Currying в Haskell

- `t1 -> (t2 -> 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`
- В общия случай: <функция> `::` `t1 -> (t2 -> ...(tn -> t)...)`
- <функция> ще очаква n параметъра от типове `t1`, `t2`, ..., `tn` и ще връща резултат от тип `t`
- <функция> <параметър$_1$> ... <параметър$_n$> `=` <тяло>
- дефинира <функция> с n параметъра и <тяло>
  - `hypothenuse :: Double -> Double -> Double`
  - `hypothenuse a b = sqrt (a**2 + b**2)`
  - `hypothenuse 3 4` $\longrightarrow$ `5`

Частично прилагане на функции

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

- `div50 :: Int -> Int`

$$div50(x) := \left\lfloor \frac{50}{x} \right\rfloor$$

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

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

- `div50 :: Int -> Int`
- `div50 x = div 50 x`
- `div50 4` $\longrightarrow$ `12`

Частично прилагане на функции

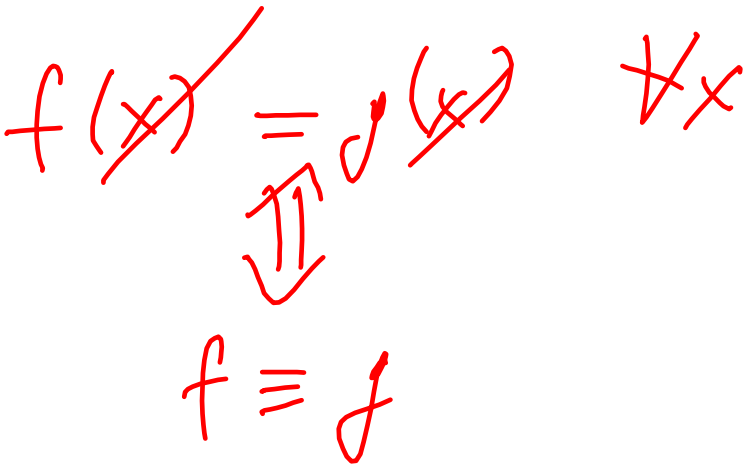
Кърингът позволява удобно прилагане на функция към само част от параметрите.

- `div50 :: Int -> Int`
- `div50 x = div 50 x`
- `div50 4` $\longrightarrow$ `12`

Частично прилагане на функции

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

- `div50 :: Int -> Int`
- `div50 = div 50`
- `div50 4` $\longrightarrow$ `12`



Функции от по-висок ред

Внимание: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

Функции от по-висок ред

Внимание: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**

Функции от по-висок ред

Внимание: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**
- `t1 -> (t2 -> t3)` $\equiv$ `t1 -> t2 -> t3`

Функции от по-висок ред

Внимание: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**
- `t1 -> (t2 -> t3)` $\equiv$ `t1 -> t2 -> t3`
- `(t1 -> t2) -> 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 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**
- `t1 -> (t2 -> t3)` $\equiv$ `t1 -> t2 -> t3`
- `(t1 -> t2) -> 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: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**
- `t1 -> (t2 -> t3)` $\equiv$ `t1 -> t2 -> t3`
- `(t1 -> t2) -> 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
  - `twice f x = f (f x)`

Функции от по-висок ред

Внимание: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**
- `t1 -> (t2 -> t3)` $\equiv$ `t1 -> t2 -> t3`
- `(t1 -> t2) -> 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
  - `twice f x = f (f x)`
  - `twice :: (Int -> Int) -> Int -> Int`

Функции от по-висок ред

Внимание: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**
- `t1 -> (t2 -> t3)` $\equiv$ `t1 -> t2 -> t3`
- `(t1 -> t2) -> 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
  - `twice f x = f (f x)`
  - `twice :: (Int -> Int) -> Int -> Int`
  - `twice square 3` $\longrightarrow$?

Функции от по-висок ред

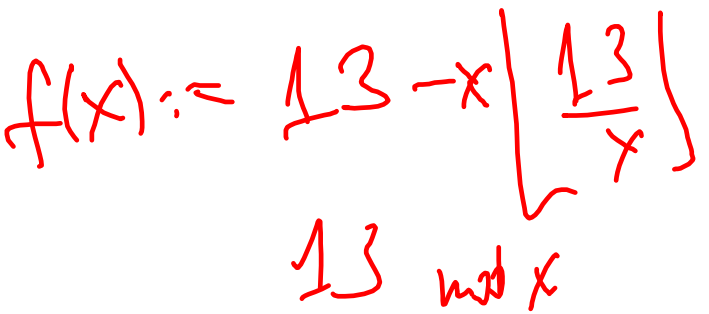
Внимание: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**
- `t1 -> (t2 -> t3)` $\equiv$ `t1 -> t2 -> t3`
- `(t1 -> t2) -> 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
  - `twice f x = f (f x)`
  - `twice :: (Int -> Int) -> Int -> Int`
  - `twice square 3` $\longrightarrow$ `81`

Функции от по-висок ред

Внимание: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**
- `t1 -> (t2 -> t3)` $\equiv$ `t1 -> t2 -> t3`
- `(t1 -> t2) -> 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
  - `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: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**
- `t1 -> (t2 -> t3)` $\equiv$ `t1 -> t2 -> t3`
- `(t1 -> t2) -> 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
  - `twice f x = f (f x)`
  - `twice :: (Int -> Int) -> Int -> Int`
  - `twice square 3` $\longrightarrow$ `81`
  - `twice (mod 13) 5` $\longrightarrow$ `1`

13 mod 5 = 3

13 mod 3 = 1

Функции от по-висок ред

Внимание: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**
- `t1 -> (t2 -> t3)` $\equiv$ `t1 -> t2 -> t3`
- `(t1 -> t2) -> 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
  - `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: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**
- `t1 -> (t2 -> t3)` $\equiv$ `t1 -> t2 -> t3`
- `(t1 -> t2) -> 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
  - `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: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**
- `t1 -> (t2 -> t3)` $\equiv$ `t1 -> t2 -> t3`
- `(t1 -> t2) -> 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
  - `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: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**
- `t1 -> (t2 -> t3)` $\equiv$ `t1 -> t2 -> t3`
- `(t1 -> t2) -> 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
  - `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: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**
- `t1 -> (t2 -> t3)` $\equiv$ `t1 -> t2 -> t3`
- `(t1 -> t2) -> 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
  - `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: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**
- `t1 -> (t2 -> t3)` $\equiv$ `t1 -> t2 -> t3`
- `(t1 -> t2) -> 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
  - `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 hypothenuse 1` $\longrightarrow$?

Функции от по-висок ред

Внимание: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**
- `t1 -> (t2 -> t3)` $\equiv$ `t1 -> t2 -> t3`
- `(t1 -> t2) -> 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
  - `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 hypothenuse 1` $\longrightarrow$ `1.4142135623730951`

Функции от по-висок ред

Внимание: `t1 -> (t2 -> t3)` $\neq$ `(t1 -> t2) -> t3`!

- операцията `->` е **дясноасоциативна**
- `t1 -> (t2 -> t3)` $\equiv$ `t1 -> t2 -> t3`
- `(t1 -> t2) -> 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
  - `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 hypothenuse 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` $\longrightarrow$ 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` $\longrightarrow$ Грешка!
  - `square (-x)` $\longrightarrow$ `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` $\longrightarrow$ Грешка!
  - `square (-x)` $\longrightarrow$ `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` $\longrightarrow$ Грешка!
  - `square (-x)` $\longrightarrow$ `4`
- Преобразуване на двуаргументни функции към бинарни операции: ‘<функция>‘
  - `13 ‘div‘ 5` $\longrightarrow$ `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` $\longrightarrow$ Грешка!
  - `square (-x)` $\longrightarrow$ `4`
- Преобразуване на двуаргументни функции към бинарни операции: ‘<функция>‘
  - `13 ‘div‘ 5` $\longrightarrow$ `3`
  - `2 ‘square‘` $\longrightarrow$ Грешка!

Операции и функции

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

Операции и функции

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

Операции и функции

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

Операции и функции

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

Операции и функции

- Преобразуване на операции към двуаргументни функции: (<операция>)
  - `(+) 2 3` $\longrightarrow$ `5`
  - `plus1 = (+) 1`
  - `square = 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`
  - `square = (^2)`
  - `(-5) 8` $\longrightarrow$?

Операции и функции

- Преобразуване на операции към двуаргументни функции: (<операция>)
  - `(+) 2 3` $\longrightarrow$ `5`
  - `plus1 = (+) 1`
  - `square = 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`
  - `square = (^2)`
  - `(-5) 8` $\longrightarrow$ Грешка!

Операции и функции

- Преобразуване на операции към двуаргументни функции: (<операция>)
  - `(+) 2 3` $\longrightarrow$ `5`
  - `plus1 = (+) 1`
  - `square = 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`
  - `square = (^2)`
  - `(-5) 8` $\longrightarrow$ Грешка!
  - `twice (*2) 5` $\longrightarrow$?

Операции и функции

- Преобразуване на операции към двуаргументни функции: (<операция>)
  - `(+) 2 3` $\longrightarrow$ `5`
  - `plus1 = (+) 1`
  - `square = 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`
  - `square = (^2)`
  - `(-5) 8` $\longrightarrow$ Грешка!
  - `twice (*2) 5` $\longrightarrow$ `20`

Операции и функции

- Преобразуване на операции към двуаргументни функции: (<операция>)
  - `(+) 2 3` $\longrightarrow$ `5`
  - `plus1 = (+) 1`
  - `square = 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`
  - `square = (^2)`
  - `(-5) 8` $\longrightarrow$ Грешка!
  - `twice (*2) 5` $\longrightarrow$ `20`
  - `positive = (>0)`

Операции и функции

- Преобразуване на операции към двуаргументни функции: (<операция>)
  - `(+) 2 3` $\longrightarrow$ `5`
  - `plus1 = (+) 1`
  - `square = 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`
  - `square = (^2)`
  - `(-5) 8` $\longrightarrow$ Грешка!
  - `twice (*2) 5` $\longrightarrow$ `20`
  - `positive = (>0)`
  - `lastDigit = (`mod` 10)`

`if ... then ... else`

- `if` <условие> `then` <израз$_1$> `else` <израз$_2$>
  - Ако <условие> е `True`, връща <израз$_1$>
  - Ако <условие> е `False`, връща <израз$_2$>

if ... then ... else

- `if` <условие> `then` <израз$_1$> `else` <израз$_2$>
  - Ако <условие> е `True`, връща <израз$_1$>
  - Ако <условие> е `False`, връща <израз$_2$>
- `abs x = if x < 0 then -x else x`

if ... then ... else

- `if` <условие> `then` <израз$_1$> `else` <израз$_2$>
  - Ако <условие> е `True`, връща <израз$_1$>
  - Ако <условие> е `False`, връща <израз$_2$>
- `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` <израз$_1$> `else` <израз$_2$>
  - Ако <условие> е `True`, връща <израз$_1$>
  - Ако <условие> е `False`, връща <израз$_2$>
- `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"` $\longrightarrow$ Грешка!

if ... then ... else

- `if` <условие> `then` <израз$_1$> `else` <израз$_2$>
  - Ако <условие> е `True`, връща <израз$_1$>
  - Ако <условие> е `False`, връща <израз$_2$>
- `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"` $\longrightarrow$ Грешка!
- <израз$_1$> и <израз$_2$> трябва да са от един и същи тип!

if ... then ... else

- if <условие> then <израз$_1$> else <израз$_2$>
  - Ако <условие> е `True`, връща <израз$_1$>
  - Ако <условие> е `False`, връща <израз$_2$>
- `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"` $\longrightarrow$ Грешка!
- <израз$_1$> и <израз$_2$> трябва да са от един и същи тип!
- <условие> трябва да е от тип `Bool`!