

# Дефиниране на функции в Haskell

Трифон Трифонов

Функционално програмиране, 2025/26 г.

13–20 ноември 2025 г.

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

## Разглеждане на случаи

Можем да дефинираме функции с разглеждане на случаи по параметрите.  
Условието на всеки случай се нарича пазач.

- $\langle \text{име} \rangle \{ \langle \text{параметър} \rangle \}$   
 $\{ \mid \langle \text{пазач} \rangle = \langle \text{израз} \rangle \}^+$
- $\langle \text{име} \rangle \langle \text{параметър}_1 \rangle \langle \text{параметър}_2 \rangle \dots \langle \text{параметър}_k \rangle$   
 $\mid \langle \text{пазач}_1 \rangle = \langle \text{израз}_1 \rangle$   
 $\dots$   
 $\mid \langle \text{пазач}_n \rangle = \langle \text{израз}_n \rangle$
- ако  $\langle \text{пазач}_1 \rangle$  е `True` връща  $\langle \text{израз}_1 \rangle$ , а ако е `False`:
- $\dots$
- ако  $\langle \text{пазач}_n \rangle$  е `True` връща  $\langle \text{израз}_n \rangle$ , а ако е `False`:
- **грешка!**
- За удобство `Prelude` дефинира `otherwise = True`

## Разглеждане на случаи — примери

fact n

```
| n == 0    = 1
| n > 0    = n * fact (n - 1)
| n < 0    = error "подадено отрицателно число"
```

- fact (-5) → Грешка!
- добра практика е да имаме изчерпателни случаи
- можем да използваме стандартната функция `error`

grade x

```
| x >= 5.5      = "Отличен"
| x >= 4.5      = "Много добър"
| x >= 3.5      = "Добър"
| x >= 3         = "Среден"
| otherwise       = "Слаб"
```

# Локални дефиниции с let

- `let { <дефиниция> }+`  
`in <тяло>`
- `let <дефиниция1>`  
`<дефиниция2>`  
...  
`<дефиницияn>`  
`in <тяло>`
- `<дефиницияi>` се въвеждат **едновременно**
- областта на действие на дефинициите е само в рамките на `let` конструкцията
- може да са взаимно рекурсивни

## Примери за let

- `let x = 5 in x + 3` → 8
- `let f x = y + x` → 63  
    `y = 7`  
    `in f 2 * y`
- `fact2 n = let fact n = if n == 0 then 1  
                              else n * fact (n-1)  
                              in (fact n)^2`
- В интерактивен режим (GHCi) `let` може да се използва без `in` за въвеждане на нови дефиниции

## Локални дефиниции с where

- <дефиниция-на-функция>  
`where { <дефиниция> }+`
- <дефиниция-на-функция>  
`where <дефиниция1>`  
`<дефиниция2>`  
...  
`<дефиницияn>`
- <дефиниция<sub>i</sub>> се въвеждат **едновременно**
- областта на действие на дефинициите е само в рамките на дефиницията на <функция>
- може да са взаимно рекурсивни

## Примери за where

```
sumLastDigits n = lastDigit n + lastDigit (stripDigit n)
where lastDigit  = ('mod' 10)
      stripDigit = ('div' 10)
```

```
quadratic a b c
| a == 0      = "линейно уравнение"
| d > 0       = "две реални решения"
| d == 0       = "едно реално решение"
| otherwise    = "няма реални решения"
where d = b^2 - 4*a*c
```

## Пример за комбиниране на let и where

```
area x1 y1 x2 y2 x3 y3 =  
  let a = dist x1 y1 x2 y2  
      b = dist x2 y2 x3 y3  
      c = dist x3 y3 x1 y1  
      p = (a + b + c) / 2  
  in sqrt (p * (p - a) * (p - b) * (p - c))  
  where dist u1 v1 u2 v2 = sqrt (du^2 + dv^2)  
      where du = u2 - u1  
            dv = v2 - v1
```

# Сравнение на let и where

- `let` е израз, който може да участва във всеки израз
- `where` може да се използва само в рамките на дефиниция
- `where` дефинициите са видими при всички случаи с пазачи
- `let` са удобни когато има само едно <тяло>
- стилистична разлика:
  - с `let` помощните дефиниции се дават първи
  - с `where` акцентът пада върху основната дефиниция

# Подравняване на дефинициите

```
let h = f + g
    b x = 2
in  b h
```

а защо не:

```
let h = f + g b
    x = 2
in  b h
```

- Подравняването в Haskell има значение!
- Обхватът на блок от дефиниции се определя от това как са подравнени.
- Дефинициите **точно подравнени** по първата са в същия блок
- Дефинициите **вдясно** от първата са в нов вътрешен блок
- Дефинициите **вляво** от първата са във външния блок

## Двумерен синтаксис — пример

```
area x1 y1 x2 y2 x3 y3 =
```

```
let    a = dist x1 y1 x2 y2  
      b = dist x2 y2 x3 y3  
      c = dist x3 y3 x1 y1  
      p = (a + b + c) / 2
```

```
in sqrt (p * (p - a) * (p - b) * (p - c))
```

```
where dist u1 v1 u2 v2 = sqrt (du^2 + dv^2)
```

```
where du = u2 - u1  
      dv = v2 - v1
```

# Алтернативен синтаксис за блокове

- Въщност подравняването е синтактична захар за блок в Haskell
- { { <дефиниция> ; } }
- { <дефиниция<sub>1</sub>> ; ... <дефиниция<sub>n</sub>> [;] }
- Интуитивни правила:
  - при първия символ на дефиниция — запомни позицията и сложи {
  - новият ред е подравнен по първия — сложи ;
  - новият ред е по-наляво — сложи }
  - новият ред е по-надясно — не слагай нищо, счита се за продължение на предния ред
- Пазачите не използват синтаксис за блокове, можем безопасно да ги пишем и на един ред:
- `fact n | n == 0 = 1 | otherwise = n * fact (n-1)`

## Поредица от равенства

Можем да дефинираме функция с поредица от равенства:

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

Можем да имаме произволен брой равенства...

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

... или варианти за различните параметри

```
gcd 0 0 = error "няма най-голям общ делител"
gcd x 0 = x
gcd 0 y = y
gcd x y
| x > y      = gcd (x-y) y
| otherwise   = gcd x (y-x)
```

# Образци

- Как се разбира кое равенство да се използва?
- Видът на формалните параметри наричаме **образец**
- Търси се на кой образец пасва фактическия параметър
- Избира се първият образец **отгоре надолу**
- Видове образци:
  - **литерали** — пасват при точно съвпадение
  - **променливи** — пасват винаги
  - **анонимен образец \_** — пасва винаги без да свързва фактическата стойност с име
- Пример:

```
False && _ = False  
_     && b = b
```

```
(&&) False _ = False  
(&&) _      b = b
```

## Повторение на променливи

- Можем ли да напишем

```
gcd 0 0 = error "няма най-голям общ делител"
```

```
gcd x 0 = x
```

```
gcd 0 y = y
```

```
gcd x x = x
```

```
gcd x y
```

```
  | x > y      = gcd (x-y) y
```

```
  | otherwise   = gcd x (y-x)
```

- Не!
- Всички променливи в образците трябва да са уникални
- Няма унификация, както в Пролог
  - Има езици за функционално и логическо програмиране, в които това е позволено (напр. Curry)