Синтаксис за дефиниране на функции

Трифон Трифонов

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

1–15 декември 2016 г.

Разглеждане на случаи

Можем да дефинниране на функции с разглеждане на случаи по параметрите.

Условието на всеки случай се нарича **пазач**.

- <име> {<параметър>}
  { | <пазач> = <израз> }$^{+}$

Разглеждане на случаи

Можем да дефинниране на функции с разглеждане на случаи по параметрите.

Условието на всеки случай се нарича **пазач**.

- <име> {<параметър>}
  { | <пазач> = <израз> }$^+$
- <име> <параметър$_1$> <параметър$_2$> ... <параметър$_k$>
  | <пазач$_1$> = <израз$_1$>
  ...
  | <пазач$_n$> = <израз$_n$>

Разглеждане на случаи

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

- <име> {<параметър>}
  { | <пазач> = <израз> }$^+$
- <име> <параметър$_1$> <параметър$_2$> ... <параметър$_k$>
  | <пазач$_1$> = <израз$_1$>
  ...
  | <пазач$_n$> = <израз$_n$>
- ако <пазач$_1$> е `True` връща <израз$_1$>, а ако е `False`:
- ...
- ако <пазач$_n$> е `True` връща <израз$_n$>, а ако е `False`:
- грешка!

Разглеждане на случаи

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

- <име> {<параметър>}
  { | <пазач> = <израз> }$^+$
- <име> <параметър$_1$> <параметър$_2$> ... <параметър$_k$>
  | <пазач$_1$> = <израз$_1$>
  ...
  | <пазач$_n$> = <израз$_n$>
- ако <пазач$_1$> е `True` връща <израз$_1$>, а ако е `False`:
- ...
- ако <пазач$_n$> е `True` връща <израз$_n$>, 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)
```

Разглеждане на случаи — примери

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

- `fact (-5)` $\longrightarrow$?

Разглеждане на случаи — примери

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

- `fact (-5)` $\longrightarrow$ Грешка!

Разглеждане на случаи — примери

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

- `fact (-5)` $\longrightarrow$ Грешка!
- добра практика е да имаме изчерпателни случаи

Разглеждане на случаи — примери

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

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

Разглеждане на случаи — примери

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

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

Разглеждане на случаи — примери

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

- `fact (-5)` $\longrightarrow$ 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>

Локални дефиниции с `let`

- `let` { <дефиниция> }$^+$
  `in` <тяло>
- `let` <дефиниция$_1$>
  <дефиниция$_2$>
  ...
  <дефиниция$_n$>
  `in` <тяло>

Локални дефиниции с `let`

- `let` { <дефиниция> }$^+$
  `in` <тяло>
- `let` <дефиниция$_1$>
  <дефиниция$_2$>
  ...
  <дефиниция$_n$>
  `in` <тяло>
- <дефиниция$_i$> 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` $\longrightarrow$ 8

Примери за let

- `let x = 5 in x + 3` $\longrightarrow$ 8
- `let f x = y + x` $\longrightarrow$?

```
    y = 7
in  f 2 * y
```

Примери за let

- `let x = 5 in x + 3` $\longrightarrow$ 8
-
```
let f x = y + x
    y = 7
in  f 2 * y
```
$\longrightarrow$ 63

Примери за let

- `let x = 5 in x + 3` $\longrightarrow$ 8
-
```
let f x = y + x
    y = 7
in  f 2 * y
```
$\longrightarrow$ 63
-
```
fact2 n = let fact n =  if n == 0 then 1
                        else n * fact (n-1)
          in (fact n)^2
```

Примери за let

- `let x = 5 in x + 3` $\longrightarrow$ 8
-
```
let f x = y + x
    y = 7
in  f 2 * y
```
$\longrightarrow$ 63
-
```
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> }$^{+}$

Локални дефиниции с `where`

- <дефиниция-на-функция>
  `where` { <дефиниция> }$^+$
- <дефиниция-на-функция>
  `where` <дефиниция$_1$>
  <дефиниция$_2$>
  ...
  <дефиниция$_n$>

Локални дефиниции с `where`

- <дефиниция-на-функция>
  `where` { <дефиниция> }$^+$
- <дефиниция-на-функция>
  `where` <дефиниция$_1$>
  <дефиниция$_2$>
  ...
  <дефиниция$_n$>
- <дефиниция$_i$> 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)
```

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

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

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

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>

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:

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

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

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

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
```

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!

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.

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 **вдясно и надолу** от първата са в същия блок

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 **вдясно и надолу** от първата са в същия блок
- Дефинициите **вляво** 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

- Всъщност подравняването е синтактична захар за блок в Haskell

Алтернативен синтаксис за блокове

- Всъщност подравняването е синтактична захар за блок в Haskell
- { { <дефиниция> ; } }

Алтернативен синтаксис за блокове

- Всъщност подравняването е синтактична захар за блок в Haskell
- { { <дефиниция> ; } }
- { <дефиниция$_1$> ; ... <дефиниция$_n$> [;] }

Алтернативен синтаксис за блокове

- Всъщност подравняването е синтактична захар за блок в Haskell
- { { <дефиниция> ; } }
- { <дефиниция$_1$> ; ... <дефиниция$_n$> [;] }
- Интуитивни правила:

Алтернативен синтаксис за блокове

- Всъщност подравняването е синтактична захар за блок в Haskell
- { { <дефиниция> ; } }
- { <дефиниция$_1$> ; ... <дефиниция$_n$> [;] }
- Интуитивни правила:
  - при първия символ на дефиниция — запомни позицията и сложи {

Алтернативен синтаксис за блокове

- Всъщност подравняването е синтактична захар за блок в Haskell
- { { <дефиниция> ; } }
- { <дефиниция$_1$> ; ... <дефиниция$_n$> [;] }
- Интуитивни правила:
  - при първия символ на дефиниция — запомни позицията и сложи {
  - начало на дефиниция по-надясно или по-надолу — сложи ;

Алтернативен синтаксис за блокове

- Всъщност подравняването е синтактична захар за блок в Haskell
- { { <дефиниция> ; } }
- { <дефиниция$_1$> ; ... <дефиниция$_n$> [;] }
- Интуитивни правила:
  - при първия символ на дефиниция — запомни позицията и сложи {
  - начало на дефиниция по-надясно или по-надолу — сложи ;
  - начало на дефиниция по-наляво — сложи }

Алтернативен синтаксис за блокове

- Всъщност подравняването е синтактична захар за блок в Haskell
- { { <дефиниция> ; } }
- { <дефиниция$_1$> ; ... <дефиниция$_n$> [;] }
- Интуитивни правила:
  - при първия символ на дефиниция — запомни позицията и сложи {
  - начало на дефиниция по-надясно или по-надолу — сложи ;
  - начало на дефиниция по-наляво — сложи }
- Пазачите не използват синтаксис за блокове, можем безопасно да ги пишем и на един ред:

Алтернативен синтаксис за блокове

- Всъщност подравняването е синтактична захар за блок в Haskell
- { { <дефиниция> ; } }
- { <дефиниция$_1$> ; ... <дефиниция$_n$> [;] }
- Интуитивни правила:
  - при първия символ на дефиниция — запомни позицията и сложи {
  - начало на дефиниция по-надясно или по-надолу — сложи ;
  - начало на дефиниция по-наляво — сложи }
- 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)
```

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)
```

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?

Образци

- Как се разбира кое равенство да се използва?
- Видът на формалните параметри наричаме образец

Образци

- Как се разбира кое равенство да се използва?
- Видът на формалните параметри наричаме образец
- Търси се на кой образец пасва фактическия параметър

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

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:

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

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

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

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
```

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
```

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)
```

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!

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

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

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)