Тема 3

Оператори. Дефиниране на оператори в Haskell

Дефиниция и свойства на операторите

Операторите в Haskell са ***инфиксни функции***, т.е. такива (двуаргументни) функции, означенията на които се записват между аргументите им, а не преди тях.

По принцип е възможно поредици от прилагания на група оператори да бъдат записани с използване на скоби като например

((((4+8)+(7+2))*3)+5)

Този запис обаче не е много удобен, затова по принцип употребата на излишни скоби на практика се избягва.

Това е възможно заради наличието на две важни свойства на операторите – техните ***приоритет*** (сила на свързването, binding power) и ***асоциативност***.

Приоритет

Приоритетът е свойство на операторите, което определя реда на изпълнение на поредица от различни оператори.

Например в аритметиката операциите (операторите) умножение и деление имат по-висок приоритет от събирането и изваждането, а степенуването има по-висок приоритет от умножението и делението.

Това означава, че 2+3*4 е еквивалентно на (2+(3*4)), а 2^3*4 е еквивалентно на ((2^3)*4).

В Haskell всеки (вграден) оператор има своя ***сила на свързването*** (binding power) – цяло неотрицателно число (цяло число между 0 и 9), което определя неговия приоритет.

Например умножението (*) има приоритет (сила на свързването) 7, събирането (+) има приоритет 6 и т.н.

Асоциативност

Асоциативността е свойство на операторите, което определя реда на изпълнение на поредица от еднакви оператори.

Например в аритметиката операциите (операторите) събиране и умножение са ***асоциативни***, т.е. редът на изпълнение на поредица от събирания и умножения е без значение:

(a + b) + c = a + (b + c),

т.е. записът a + b + c може да се интерпретира еднозначно;

(a * b) * c = a * (b * c),

т.е. записът a * b * c може да се интерпретира еднозначно.

Изваждането и делението обаче не са асоциативни, защото

$(a - b) - c \neq a - (b - c)$ и

$(a / b) / c \neq a / (b / c)$

В Haskell повечето (но не всички) оператори се характеризират или като ***ляво асоциативни***, или като ***дясно асоциативни***.

Ако един оператор е ляво асоциативен, то всяка поредица от последователни обръщения към него се интерпретира като заградена със скоби от ляво.

Ако един оператор е дясно асоциативен, то всяка поредица от последователни обръщения към него се интерпретира като заградена със скоби от дясно.

Например, изваждането и делението са ляво асоциативни, т.е.
a – b – c се интерпретира като (a – b) – c и
a / b / c се интерпретира като (a / b) / c.

Обратно, степенуването е дясно асоциативно, т.е.
a ^ b ^ c се интерпретира като a ^ (b ^ c).

Забележка 1. Най-висок приоритет в Haskell има прилагането на функции, което стандартно се записва в префиксна форма:
f v_1 v_2 … v_n .

Това в частност означава, че записът **f n+1** се интерпретира като **(f n) + 1**.

Забележка 2. Знакът "–" е означение едновременно на инфиксен и префиксен оператор, затова възниква опасност от колизия при използването му в случаи от типа на **f -12**. Този конкретен израз се интерпретира като означение на разликата (изваждането) f-12, а не като прилагане на f към числото -12.

Дефиниране на оператори

Haskell позволява на потребителя да дефинира нови оператори по същия начин, както се извършва дефинирането на функции.

Имената на операторите могат да включват ASCII символите
! # $ % & * + . / < > ? \ ^ | : - ~
Името на оператор не може да започва с двоеточие.

Например дефиницията на оператора &&& като функция за намиране на минималното от две цели числа може да изглежда по следния начин:

```
(&&&) :: Int -> Int -> Int
x &&& y
   | x>y       = y
   | otherwise = x
```

Приоритетът и асоциативността на един оператор, дефиниран от потребителя, могат да бъдат специфицирани явно, например:

infixl 7 &&& означава, че операторът &&& има лява асоциативност и приоритет 7;
infixr 6 ^^ означава, че операторът ^^ има дясна асоциативност и приоритет 6.

Генерични функции. Полиморфизъм

Много от вградените функции в Haskell са ***полиморфни*** или ***генерични***, т.е. действат върху аргументи от различни типове.

"**Полиморфизъм**" буквално означава "наличие на много форми". ***Една функция е полиморфна, когато има много типове.***

Такива са например голяма част от функциите за работа със списъци.

Пример

Функцията length връща като резултат дължината (броя на елементите) на даден списък, независимо от типа на неговите елементи.

Следователно може да се запише:

```
length :: [Bool] -> Int
length :: [Int] -> Int
length :: [[Char]] -> Int
```

и т.н.

Обобщеният запис, който капсулира (encapsulates) горните, е length :: [a] -> Int

Тук ***a*** е ***променлива на тип*** (типова променлива, type variable), т.е. променлива, която означава произволен тип.

Типовете от вида на [Bool] -> Int, [Int] -> Int, [[Int]] -> Int и т. н. са ***екземпляри*** на типа [a] -> Int.

Забележка. Променливата ***a*** в записа по-горе може да означава произволен тип, но всички нейни включвания в дадена дефиниция означават един и същ тип.

Някои функции за работа със списъци, реализирани в Prelude.hs:

`:`	`a -> [a] -> [a]`	Add a single element to the front of a list. `1:[2,3] => [1,2,3]`
`++`	`[a] -> [a] -> [a]`	Join two lists together. `"ab"++"cde" => "abcde"`
`!!`	`[a] -> Int -> a`	`xs!!n` returns the nth element of `xs`, starting at the beginning and counting from 0. `[14,7,3]!!1 => 7`
`concat`	`[[a]] -> [a]`	Concatenate a list of lists into a single list. `concat [[2,3],[],[4]] => [2,3,4]`
`length`	`[a] -> Int`	The length of the list. `length "word" => 4`

head	[a] -> a	The first element of the list. head “word” => ‘w’
last	[a] -> a	The last element of the list. last “word” => ‘d’
tail	[a] -> [a]	All but the first element of the list. tail “word” => “ord”
init	[a] -> [a]	All but the last element of the list. init “word” => “wor”
replicate	Int -> a -> [a]	Make a list of n copies of the item. replicate 3 ‘c’ => “ccc”
take	Int -> [a] -> [a]	Take n elements from the front of a list. take 3 “Peccary” => “Pec”

drop	Int -> [a] -> [a]	Drop n elements from the front of a list. drop 3 “Peccary” => “cary”
splitAt	Int->[a]->([a],[a])	Split a list at a given position. splitAt 3 “Peccary” => (“Pec”,”cary”)
reverse	[a] -> [a]	Reverse the order of the elements. reverse [1,2,3] => [3,2,1]
zip	[a]->[b]->[(a,b)]	Take a pair of lists into a list of pairs. zip [1,2] [3,4,5] => [(1,3),(2,4)]

unzip	**[(a,b)] -> ([a],[b])**	**Take a list of pairs into a pair of lists.** **unzip [(1,5),(2,6)] =>** **([1,2],[5,6])**
and	**[Bool] -> Bool**	**The conjunction of a list of Booleans.** **and [True,False] => False**
or	**[Bool] -> Bool**	**The disjunction of a list of Booleans.** **or [True,False] => True**
sum	**[Int] -> Int** **[Float] -> Float**	**The sum of a numeric list.** **sum [2,3,4] => 9**
product	**[Int] -> Int** **[Float] -> Float**	**The product of a numeric list.** **product [0.1,0.4 .. 1]** **=> 0.028**