Вектори, списъци и символни низове в Haskell

Вектори (n-торки, tuples)

Векторът (tuple) представлява наредена n-торка от елементи, при това броят n на тези елементи и техните типове трябва да бъдат определени предварително. Допуска се елементите на векторите да бъдат от различни типове.

Възможно е да се дефинира тип “вектор” от вида (t_1,t_2, … ,t_n), който включва векторите (v_1,v_2, … ,v_n), за които v_1::t_1, v_2::t_2, … , v_n::t_n.

Примери

```
type ShopItem = (String,Int)
i1 :: ShopItem
i2 :: ShopItem
i1 = ("Salt: 1 kg",139)
i2 = ("Sugar: 0.5 kg",28)
```

Haskell поддържа множество **селектори** за стойностите от тип вектор. Такива са например функциите ***fst*** и ***snd***:

```
fst (x,y) = x
snd (x,y) = y
```

При дефиниране на функции за работа с вектори често освен (вместо) селекторите се използва апаратът на съпоставянето по образец (pattern matching).

Пример 1

```
addPair :: (Int,Int) -> Int
addPair p = fst p + snd p
```

Пример 2

```
addPair :: (Int,Int) -> Int
addPair (x,y) = x+y
```

Образците могат да съдържат литерали и вложени образци, например

```
addPair (0,y) = y
addPair (x,y) = x+y

shift :: ((Int,Int),Int) -> (Int,(Int,Int))
shift ((x,y),z) = (x,(y,z))
```

Списъци

Списъкът в Haskell е редица от (променлив брой) елементи от определен тип. За всеки тип t в езика е дефиниран също и типът [t], който включва списъците с елементи от t.

Запис на списъците в Haskell:

[]: празен списък (списък без елементи). Принадлежи на всеки списъчен тип.

$[e_1,e_2, \ldots , e_n]$: списък с елементи $e_1, e_2, \ldots , e_n$.

Други форми на запис на списъци от числа, знакове (characters) и елементи на изброими типове:

- [n .. m] е списъкът [n, n+1, … , m]; ако n>m, списъкът е празен.

[2 .. 7] = [2,3,4,5,6,7]
[3.1 .. 7.0] = [3.1,4.1,5.1,6.1]
[‘a’ .. ‘m’] = “abcdefghijklm”

- [n,p .. m] е списъкът, чийто първи два елемента са n и p, последният му елемент е m и стъпката на нарастване на елементите му е p-n.

[7,6 .. 3] = [7,6,5,4,3]
[0.0,0.3 .. 1.0] = [0.0,0.3,0.6,0.9]
[‘a’,’c’ .. ‘n’] = “acegikm”

- Както се вижда от примерите, и в двата случая по-горе е възможно големината на стъпката да не позволява достигането точно на m. Тогава последният елемент на списъка съвпада с най-големия/най-малкия елемент на редицата, който е по-малък/по-голям или равен на m.

Определяне на обхвата на списък (List Comprehension)

Синтаксис:

`[expr | q1, … , qk]` , където **`expr`** е израз, а **`qi`** може да бъде

- ***генератор*** от вида **`p <- lExp`**, където **`p`** е образец и **`lExp`** е израз от списъчен тип
- ***тест***, **`bExp`**, който е булев израз

При това в q_i могат да участват променливите, използвани в q_1, q_2, … , q_{i-1}.

Пример 1

Да предположим, че стойността на ex е [2,4,7]. Тогава записът **`[2*n | n <- ex]`** означава списъка [4,8,14].

Пример 2

`[isEven n | n <- ex] => [True,True,False]`

Пример 3

`[2*n | n <- ex, isEven n, n>3] => [8]`

Пример 4

```
addPairs :: [(Int,Int)] -> [Int]
addPairs pairList = [m+n | (m,n) <- pairList]

addPairs [(2,3),(2,1),(7,8)] => [5,3,15]
```

Пример 5

```
addOrdPairs :: [(Int,Int)] -> [Int]
addOrdPairs pairLst = [m+n | (m,n) <- pairLst, m<n]

addOrdPairs [(2,3),(2,1),(7,8)] => [5,15]
```

Пример 6

Следващата функция намира всички цифри в даден низ:

```
digits :: String -> String
digits st = [ch | ch <- st, isDigit ch]
```

Тук isDigit е функция, дефинирана в Prelude.hs (isDigit :: Char **->** Bool), която връща стойност True за тези знакове, които са цифри (‘0’, ‘1’, … , ‘9’).

Пример 7

Едно определение на обхвата на списък може да бъде част от дефиницията на функция, например

```
allEven xs = (xs == [x | x <- xs, isEven x])
allOdd xs  = ([] == [x | x <- xs, isEven x])
```

Символни низове (типът String)

Символните низове са списъци от знакове (characters), т.е. типът String е специализация на списъците:

type String = [Char]

Генерични функции (полиморфизъм)

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

Локални дефиниции

Пример 1

Функция, която връща като резултат сумата от квадратите на две числа.

```
sumSquares :: Int -> Int -> Int
sumSquares n m
   = sqN + sqM
     where
     sqN = n*n
     sqM = m*m
```

Пример 2

Най-напред ще дефинираме функцията addPairwise, която събира съответните елементи на два списъка от числа, като за целта изчерпва елементите на по-късия списък и игнорира останалите елементи на по-дългия.

Например: addPairwise [1,7] [8,4,2] = [9,11].

```
addPairwise :: [Int] -> [Int] -> [Int]
addPairwise intList1 intList2
   = [m+n | (m,n) <- zip intList1 intList2]
```

Сега ще дефинираме нова функция, addPairwise', която действа подобно на addPairwise, но включва в резултата и всички останали елементи на по-дългия списък.

Например: addPairwise' [1,7] [8,4,2,67] = [9,11,2,67].

```
addPairwise' :: [Int] -> [Int] -> [Int]
addPairwise' intList1 intList2
   = front ++ rear
     where
     minLength       = min (length intList1)
                           (length intList2)
     front           = addPairwise front1 front2
     rear            = rear1 ++ rear2
     (front1,rear1) = splitAt minLength intList1
     (front2,rear2) = splitAt minLength intList2
```

Общ вид на дефиниция на функция с използване на условия (условно равенство) с клауза where:

$f\ p_1\ p_2\ \ldots\ p_k$
  | g1 = e_1
  . . .
  | otherwise = e_r
  where
  $v_1\ a_1\ \ldots\ a_n = r_1$
  $v_2 = r_2$
  . . . . .

Клаузата where тук е присъединена към цялото условно равенство, т.е. към всички негови клаузи.

От горния запис се вижда, че локалните дефиниции могат да включват както дефиниции на променливи, така и дефиниции на функции (такава е например дефиницията на функцията v_1). Възможно е в клаузата where да бъдат включени и декларации на типовете на локалните обекти (променливи и функции).

Пример

```
maxsq x y
    | sqx > sqy   = sqx
    | otherwise   = sqy
      where
      sqx = sq x
      sqy = sq y
      sq :: Int -> Int
      sq z = z*z
```

let изрази

Възможно е да се дефинират локални променливи с област на действие, която съвпада с даден израз.

Например изразът

```
let x = 3+2 in x^2 + 2*x - 4
```

има стойност 31.

Ако в един ред са включени повече от една дефиниции, те трябва да бъдат разделени с точка и запетая, например

```
let x = 3+2; y = 5-1 in x^2 + 2*x -y
```

Област на действие на дефинициите

Един скрипт на Haskell включва поредица от дефиниции. Областта на действие на дадена дефиниция съвпада с частта от програмата, в която може да се използва тази дефиниция. Всички дефиниции на най-високо ниво в Haskell имат за своя област на действие целия скрипт, в който са включени. С други думи, дефинираните на най-високо ниво имена могат да бъдат използвани във всички дефиниции, включени в скрипта. В частност те могат да бъдат използвани в дефиниции, които се намират преди техните собствени в съответния скрипт.

Локалните дефиниции, включени в дадена клауза where, имат за област на действие условното равенство, част от което е клаузата where.

В случай, че даден скрипт съдържа повече от една дефиниция за дадено име, във всяка точка на скрипта е валидна (видима) “най-локалната” от тези дефиниции.