Тема 4, част 2

Област на действие на дефинициите в Haskell. Локални дефиниции

Локални дефиниции в Haskell

Пример 1

Дефиниция на 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 (take minLength
                                  intList1)
                            (take minLength
                                  intList2)
    rear      = drop minLength intList1 ++
                drop minLength intList2
```

Ще обърнем специално внимание, че в тази дефиниция стойността на minLength се пресмята само един път, а се използва четири пъти.

Следва още едно решение на последната задача, в което обръщенията към take и drop са заменени с подходящи обръщения към splitAt.

```
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.

Пример

```
isOdd, isEven :: Int -> Bool

isOdd n
  | n<=0          = False
  | otherwise     = isEven (n-1)

isEven n
  | n<0           = False
  | n==0          = True
  | otherwise     = isOdd (n-1)
```

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

Нека разгледаме още един път дефиницията на функцията maxsq от един от предишните примери:

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

В тази дефиниция областта на действие на дефинициите на sqx, sqy и sq и на променливите x и y е означена с големия правоъгълник, а областта на действие на променливата z е означена с малкия правоъгълник.

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.

Пример

```
maxsq x y
  | sq x > sq y   = sq x
  | otherwise     = sq y
      where
      sq x = x*x
```

Тук малкият вътрешен правоъгълник „отрязва“ тази част от големия, която съвпада с областта на действие на “вътрешната” променлива с име x.

Примерна задача: конструиране и отпечатване на касова бележка.

Да се състави програмна система на Haskell, която моделира процеса на съставяне и отпечатване на касови бележки в резултат на сканирането на бар кодовете на стоките, избрани от купувачите.

Анализ на задачата

Сканиращите устройства на касите на супермаркетите имат за цел разпознаването на бар кодовете на избраните от купувачите стоки и формирането на съответни списъци от бар кодове от типа на
`[1234, 4719, 3814, 1112, 1113, 1234]` .

Всеки списък от посочения тип трябва да се конвертира в сметка (касова бележка) от вида

Haskell Stores

```
Dry Sherry, 1lt.............5.40
Fish Fingers................1.21
Orange Jelly................0.56
Hula Hoops (Giant).........1.33
Unknown Item................0.00
Dry Sherry, 1lt.............5.40

Total.......................13.90
```

Бар кодовете и цените могат да се моделират например с цели числа (ще смятаме, че цените са изразени в пенсове). Имената на стоките ще представяме като символни низове.

Следователно, ще ни бъдат необходими следните типове:

```
-- Types of names, prices (pence) and bar-codes.

type Name    = String
type Price   = Int
type BarCode = Int
```

Конвертирането ще се осъществи на основата на данните, съдържащи се в „базата от данни“ на супермаркета, която свързва бар кодовете, имената и цените на стоките. За представянето на тази „база от данни“ ще използваме списък от типа

```
-- The database linking names prices and bar codes.

type Database = [ (BarCode,Name,Price) ]
```

Примерна „база от данни“:

```
codeIndex :: Database
codeIndex = [ (4719, "Fish Fingers" , 121),
              (5643, "Nappies" , 1010),
              (3814, "Orange Jelly", 56),
              (1111, "Hula Hoops", 21),
              (1112, "Hula Hoops (Giant)", 133),
              (1234, "Dry Sherry, 1lt", 540) ]
```

Задачата на нашата програма е да конвертира списъка от бар кодове в списък от двойки от вида **(Name,Price)**, след което да конвертира новополучения списък в подходящ символен низ, който да бъде отпечатан както беше показано по-горе.

Ще бъде полезно да разполагаме със следните дефиниции на типове:

```
-- The lists of bar codes, and of Name,Price pairs.

type TillType = [BarCode]
type BillType = [(Name,Price)]
```

Нека приемем за определеност, че дължината на редовете в сметката ще бъде равна на 30:

```
-- The length of a line in the bill.

lineLength :: Int
lineLength = 30
```

При тези условия е необходимо да бъдат дефинирани следните функции:

```
makeBill :: TillType -> BillType
-- Converts a list of bar codes to a list of
-- name/price pairs.
```

```
formatBill :: BillType -> String
-- Converts a list of name/price pairs to
-- a formatted bill.


produceBill :: TillType -> String
-- Combines the effects of makeBill and formatBill.

produceBill = formatBill . makeBill
```