Функции от по-висок ред в Haskell

Функция от по-висок ред се нарича всяка функция, която получава поне една функция като параметър (аргумент) или връща функция като резултат.

Функциите като параметри

Тук ще покажем как могат да се дефинират някои често използвани функции от по-висок ред за работа със списъци. Тези функции са дефинирани в Prelude.hs, т.е. нашите дефиниции ще бъдат направени само с учебна цел.

Прилагане на дадена функция към всички елементи на даден списък (map)

Декларация на типа:

```
map :: (a -> b) -> [a] -> [b]

input              input   output
function           list    list
```

Дефиниция (първи вариант):

```
map f xs = [f x| x <- xs]
```

Дефиниция (втори вариант):

```
map f []     = []
map f (x:xs) = f x : map f xs
```

Филтриране на елементите на даден списък (filter)

Декларация на типа:

```
filter :: (a -> Bool) -> [a] -> [a]

          input           input   output
          property        list    list
```

Дефиниция (първи вариант):

```
filter p xs = [x | x <- xs, p x]
```

Дефиниция (втори вариант):

```
filter p []      = []
filter p (x:xs)
   | p x       = x : filter p xs
   | otherwise =     filter p xs
```

Комбиниране на zip и map (zipWith)

Вече разгледахме полиморфната функция `zip :: [a] -> [b] -> [(a,b)]`, която комбинира два дадени списъка в списък от двойки от съответните елементи на тези списъци.

Ще дефинираме нова функция, zipWith, която комбинира ефекта от действието на zip и map.

Декларация на типа

Функцията zipWith ще има три аргумента: една функция и два списъка. Двата списъка (вторият и третият аргумент на zipWith) са от произволни типове (съответно [a] и [b]). Резултатът също е списък от произволен тип ([c]). Първият аргумент на zipWith е функция, която се прилага върху аргументи – съответните елементи на двата списъка и връща съответния елемент на резултата, т.е. това е функция от тип **`a -> b -> c`**.

Следователно

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Дефиниция

```
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f  _      _     = []
```

Примери

Нека plus и mult са функции, дефинирани както следва:

```
plus :: Int -> Int -> Int
plus a b = a+b

mult :: Int -> Int -> Int
mult a b = a*b
```

Тогава

```
zipWith plus [1,2,3]       [4,5,6] → [5,7,9]
zipWith mult [1,2,3,4,5] [6,7,8] → [6,14,24]
```

или също

```
zipWith (+) [1,2,3]       [4,5,6] → [5,7,9]
zipWith (*) [1,2,3,4,5] [6,7,8] → [6,14,24]
```

Забележка. Функциите map, filter и zipWith са дефинирани в Prelude.hs.

Комбиниране/акумулиране на елементите на даден списък (foldr1 и foldr)

Тук ще разгледаме група функции от по-висок ред, които реализират операцията *комбиниране* (*акумулиране*) на елементите на даден списък, използвайки подходяща функция.

Тази операция е достатъчно обща и се реализира от множество стандартни функции, включени в Prelude.hs.

Действие на функцията foldr1

Дефиницията на тази функция включва два случая:

- foldr1, приложена върху дадена функция f и списък от един елемент [a], връща като резултат a;
- Прилагането на foldr1 върху функция и по-дълъг списък е еквивалентно на
  foldr1 f [e_1,e_2, … , e_k]
  = e_1 \`f\` (e_2 \`f\` (... \`f\` e_k) ...)
  = e_1 \`f\` (foldr1 f [e_2, … , e_k])
  = f e_1 (foldr1 f [e_2, ... , e_k])

Съответната дефиниция на Haskell изглежда както следва:

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]     = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

Примери

foldr1 (+) [3,98,1] = 102
foldr1 (||) [False,True,False] = True
foldr1 min [6] = 6
foldr1 (*) [1 .. 6] = 720

Забележка. Функцията foldr1 не е дефинирана върху втори аргумент – празен списък.

Разглежданата функция може да бъде модифицирана така, че да получава един допълнителен аргумент, който определя стойността, която следва да се върне при опит за комбиниране по зададеното правило на елементите на празния списък.

Новополучената функция се нарича ***foldr*** (което означава ***fold***, т.е. комбиниране/акумулиране, извършено чрез групиране от дясно – bracketing to the **r**ight).

Дефиниция на функцията foldr:

```
foldr :: (a -> a -> a) -> a -> [a] -> a
```

binary operation over type a

starting value of type a

list of values to be combined

the result of type a

```
foldr f s []     = s
foldr f s (x:xs) = f x (foldr f s xs)
```

Примери

```
concat :: [[a]] -> [a]
concat xs = foldr (++) [] xs

and :: [Bool] -> Bool
and bs = foldr (&&) True bs
```

Забележки

1. В действителност типът на функцията foldr е по-общ:

   `foldr :: (a -> b -> b) -> b -> [a] -> b`

2. В стандартната прелюдия на Haskell е включена и функция ***foldl***, която е подобна на ***foldr***, но осъществява комбиниране/акумулиране, извършено чрез групиране от ляво – bracketing to the **l**eft.

По-сложни примери за употреба на foldr

Пример 1. Обръщане на реда на елементите на списък (алтернативна дефиниция на вградената функция reverse).

```
rev :: [a] -> [a]
rev xs = foldr snoc [] xs
```

Функцията snoc е подобна на cons (:), но добавя първия си аргумент след последния елемент на списъка, зададен като втори аргумент:

```
snoc :: a -> [a] -> [a]
snoc x xs = xs ++ [x]
```

Пример 2. Сортиране чрез вмъкване.

```
iSort :: [Int] -> [Int]
iSort xs = foldr ins [] xs

ins :: Int -> [Int] -> [Int]
ins x [] = [x]
ins x (y:ys)
   | x <= y    = x:(y:ys)
   | otherwise = y : ins x ys
```

Функциите като върнати стойности

Дефиниции на функции
на функционално ниво

Дефинирането на някаква функция на функционално ниво предполага действието на тази функция да се опише не в термините на резултата, който връща тя при прилагане към подходящо множество от аргументи, а като директно се посочи връзката ѝ с други функции.

Например, ако вече са дефинирани функциите

f :: b -> c и

g :: a -> b , то

тяхната композиция fg = f . g (т.е. функцията, за която е изпълнено fg x = (f . g) x = f (g x) за всяко x от тип a) може да се дефинира с използване на вградения оператор ‘.’ от тип

```
(.) :: (b -> c) -> (a -> b) -> (a -> c):
```

```
fg f g = (f . g)
```

Пример 1. Двукратно прилагане на функция.

```
twice :: (a -> a) -> (a -> a)
twice f = (f . f)
```

Нека например succ е функция, която прибавя 1 към дадено цяло число:

```
succ :: Int -> Int
succ n = n + 1
```

Тогава

```
(twice succ) 12
 ⟶ (succ . succ) 12
 ⟶ succ (succ 12)
 ⟶ 14
```

Пример 2. n-кратно прилагане на функция.

```
iter :: Int -> (a -> a) -> (a -> a)
iter n f
   | n>0       = f . iter (n-1) f
   | otherwise = id
```

Тук id е вградената функция – идентитет.

Дефиниране на функция, която връща функция като резултат

Пример. Функция, която за дадено цяло число n връща като резултат функция на един аргумент, която прибавя n към аргумента си.

```
addNum :: Int -> (Int -> Int)
addNum n = addN
           where
           addN m = n+m
```

Така оценка на обръщението към функцията addNum ще бъде функцията с име addN. От своя страна функцията addN е дефинирана в клаузата where.

Използваният подход може да бъде окачествен като *индиректен*: най-напред посочваме името на функцията – резултат и едва след това дефинираме тази функция.

Ламбда нотация
(ламбда изрази)

Вместо да именуваме и да дефинираме някаква функция, която бихме искали да използваме, можем да запишем директно тази функция.

Например в случая на дефиницията на addNum резултатът може да бъде дефиниран като

```
\m -> n+m
```

Последният запис е еквивалентен на израза

```
(lambda (m) (+ n m))
```

в езика Racket.

И в Haskell изрази от този вид се наричат ***ламбда изрази***, а функциите, дефинирани чрез ламбда изрази, се наричат *анонимни функции*.

Дефиницията на функцията addNum с използване на ламбда израз придобива вида

```
addNum n = (\m -> n+m)
```

Частично прилагане на функции

Нека разгледаме като пример функцията за умножение на две числа, дефинирана както следва:

```
multiply :: Int -> Int -> Int
multiply x y = x*y
```

Ако тази функция бъде приложена към два аргумента, като резултат ще се получи число, например multiply 2 3 връща резултат 6.

Какво ще се случи, ако multiply се приложи към един аргумент, например числото 2?

Отговорът е, че като резултат ще се получи функция на един аргумент y, която удвоява аргумента си, т.е. връща резултат 2*y.

Следователно, **всяка функция на два или повече аргумента може да бъде приложена частично към по-малък брой аргументи**. Тази идея дава богати възможности за конструиране на функции като оценки на обръщения към други функции.

Пример. Функцията doubleAll, която удвоява всички елементи на даден списък от цели числа, може да бъде дефинирана както следва:

```
doubleAll :: [Int] -> [Int]
doubleAll = map (multiply 2)
```

Тип на резултата от частично прилагане на функция

Правило на изключването

Ако дадена функция f е от тип
$\mathbf{t_1 \; \text{->} \; t_2 \; \text{->} \; \dots \; \text{->} \; t_n \; \text{->} \; t}$
и тази функция е приложена към аргументи
$\mathbf{e_1 :: t_1,\ e_2 :: t_2,\ \dots ,\ e_k :: t_k}$ (където $k \le n$),
то типът на резултата се определя чрез изключване на типовете t_1, t_2, … , t_k:
$\mathbf{\cancel{t_1} \; \text{->} \; \cancel{t_2} \; \text{->} \; \dots \; \text{->} \; \cancel{t_k} \; \text{->} \; t_{k+1} \; \text{->} \; \dots \; \text{->} \; t_n \; \text{->} \; t}$,
т.е. резултатът е от тип
$\mathbf{t_{k+1} \; \text{->} \; t_{k+2} \; \text{->} \; \dots \; \text{->} \; t_n \; \text{->} \; t}$.

Примери

```
multiply 2 :: Int -> Int
multiply 2 3 :: Int

doubleAll :: [Int] -> [Int]
doubleAll [2,3] :: [Int]
```

Забележки

1. Прилагането на функция е ***ляво асоциативна операция***, т.е.

f x y = (f x) y   и

f x y ≠ f (x y)

2. Операторът ‘**->**’ не е асоциативен.

Например записите

```
f :: Int -> Int -> Int
```

и

```
g :: (Int -> Int) -> Int
```

означават функции от различни типове.

Частичното прилагане на функции налага нова гледна точка върху понятието “брой на аргументите на дадена функция”. От тази гледна точка може да се каже, че всички функции в Haskell имат по един аргумент. Ако резултатът от прилагането на функцията върху даден аргумент е функция, тази функция може отново да бъде приложена върху един аргумент и т.н.

Сечения на оператори
(operator sections)

Операторите (инфиксно прилаганите двуаргументи функции) в Haskell могат да бъдат прилагани частично, като за целта се задава това, което е известно, под формата на т. нар. ***сечения на оператори*** (***operator sections***).

Примери

(+2)	Функцията, която прибавя към аргумента си числото 2.
(2+)	Функцията, която прибавя аргумента си към числото 2.
(>2)	Функцията, която проверява дали дадено число е по-голямо от 2.
(3:)	Функцията, която поставя числото 3 в началото на даден списък.

(++”\n”) Функцията, която поставя *newline* в края на даден низ.

(“\n”++) Функцията, която поставя *newline* в началото на даден низ.

Общото правило гласи, че сечението на оператора ***op*** „добавя“ аргумента си по начин, който завършва от синтактична гледна точка записа на приложението на оператора (обръщението към оператора).

С други думи,
(op x) y = y op x
(x op) y = x op y

Когато бъде комбинирана с функции от по-висок ред, нотацията на сечението на оператори е едновременно мощна и елегантна. Тя позволява да се дефинират разнообразни функции от по-висок ред.

Например,

```
filter (>0) . map (+1)
```

е функцията, която прибавя 1 към всеки от елементите на даден списък, след което премахва тези елементи на получения списък, които не са положителни числа.