Тема 8

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 fxg – функцията, за която е изпълнено (fxg f g) 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)
```

както следва:

```
fxg 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:

```
id :: a -> a
id x = x
```

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
```

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

Забележка. Символът „\“ е избран за означаване на ламбда изразите, защото той наподобява на гръцката буква **λ**.

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 в резултат ще се получи функция на един аргумент y, която удвоява аргумента си, т.е. функция, която може да бъде записана като `(\y -> 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

t_1 `->` t_2 `->` … `->` t_n `->` t

и тази функция е приложена към аргументи

e_1 `::` t_1`,` e_2 `::` t_2`, ... ,` e_k `::` t_k (където $k \leq n$),

то типът на резултата се определя чрез изключване на типовете t_1, t_2, … , t_k:

~~t_1~~ `->` ~~t_2~~ `->` `...` `->` ~~t_k~~ `->` t_{k+1} `->` `...` `->` t_n `->` t,

т.е. резултатът е от тип

t_{k+1} `->` t_{k+2} `->` `...` `->` t_n `->` 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 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 за оценяване в такива случаи гласи, че сечението на оператора ***op*** „добавя“ аргумента си по начин, който завършва от синтактична гледна точка 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.