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.

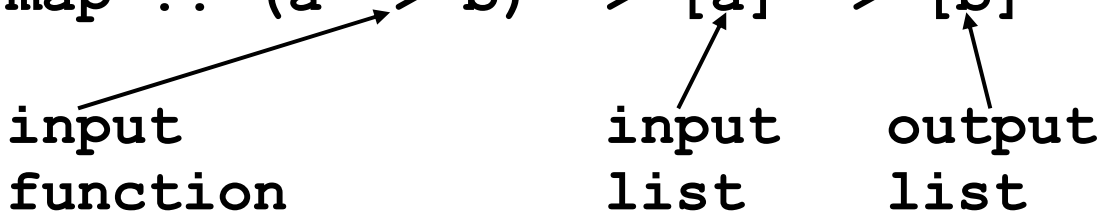
Функциите като параметри

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]
```



Дефиниция (първи вариант):

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

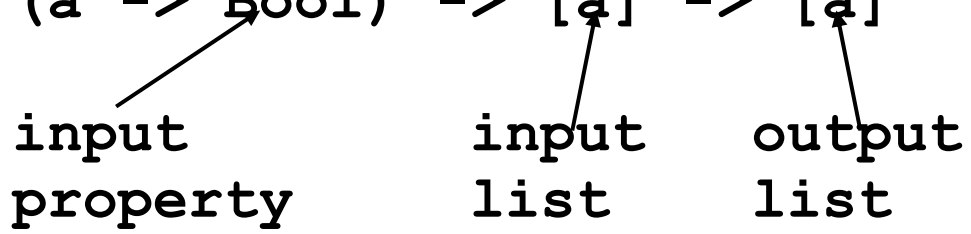
Дефиниция (втори вариант):

```
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]
```



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 полиморфичната 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]
```

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

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 f . g (т.е. функцията, за която е изпълнено (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)` .

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 Scheme.

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

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 числото 2 към аргумента си.
(2+) Функцията, която прибавя числото 2 към аргумента си.
(>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.

“Мързеливо” оценяване (lazy evaluation)

“Мързеливото” оценяване (lazy evaluation) е стратегия на оценяване, която по стандарт стои в основата на работата на всички интерпретатори на Haskell. Същността на тази стратегия е, че интерпретаторът оценява даден аргумент на дадена функция само ако (и доколкото) стойността на този аргумент е необходима за пресмятането на целия резултат. Нещо повече, ако даден аргумент е съставен (например е вектор или списък), то се оценяват само тези негови компоненти, чиито стойности са необходими от гледна точка на получаването на резултата. При това дублиращите се подизрази се оценяват по не повече от един път.

Най-важно от гледна точка на оценяването на изрази в Haskell е прилагането на функции. И тук основната идея е еквивалентна на оценяването чрез заместване в Scheme:

Оценяването на израз, който е обръщение към функцията f с аргументи a_1, a_2, … , a_k, се състои в заместване на формалните параметри от дефиницията на f съответно с изразите a_1, a_2, … , a_k и оценяване на така получения частен случай на тялото на дефиницията.

Например, ако

```
f x y = x+y
```

, то

```
f (9-3) (f 34 3)
⟶ (9-3) + (f 34 3)
```

При това изразите (9-3) и (f 34 3) не се оценяват преди да бъдат предадени като аргументи на f.

Доколкото операцията събиране изисква аргументите й да бъдат оценени, оценяването на горния израз продължава по следния начин:

```
f (9-3) (f 34 3)
⟶  ...
⟶ 6 + (34 + 3)
⟶ 6 + 37
⟶ 43
```

В конкретния случай бяха оценени и двата аргумента (фактически параметъра), но това не винаги е така.

Ако например дефинираме

`g x y = x + 12` , то

```
g (9-3) (g 34 3)
⟶ (9-3) + 12
⟶ 6 + 12
⟶ 18
```

Тук x се замества с (9-3), но y не участва в дясната страна на равенството, определящо стойността на g, следователно аргументът (g 34 3) не се оценява.

Така демонстрирахме едно от преимуществата на "мързеливото" оценяване: ***аргументи, които не са необходими, не се оценяват***.

Последният пример не е особено смислен, тъй като вторият аргумент не се използва в никакъв случай и по същество е излишен в дефиницията на g.

По-интересен е следният пример:

```
switch :: Int -> a -> a -> a
switch n x y
   | n>0       = x
   | otherwise = y
```

Ако цялото число n е положително, резултатът съвпада с оценката на x; в противен случай тя съвпада с оценката на y. С други думи, винаги при оценяване на обръщение към функцията switch се оценяват аргументът n (т.е. първият аргумент) и точно един от останалите аргументи.

Нека сега функцията h е дефинирана както следва:

```
h :: Int -> Int -> Int
h x y = x + x
```

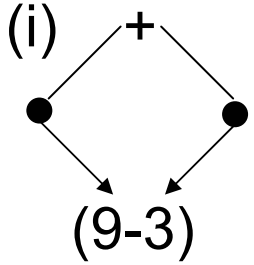
Тогава

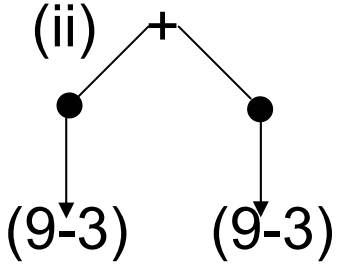
```
h (9-3) (h 34 3)
⟶ (9-3) + (9-3)
⟶ 6 + 6
⟶ 12
```

Изглежда, че в последния пример аргументът (9-3) се оценява двукратно, но принципите на “мързеливото” оценяване гарантират, че ***дублираните аргументи (т.е. многократните включвания на аргументи) се оценяват по не повече от един път***.

В реализацията на Hugs това се постига, като изразите се представят чрез подходящи графи и пресмятанията се извършват върху тези графи. В такъв граф всеки аргумент се представя чрез единствен възел и към този възел може да сочат много дъги.

Например изразът, до който се свежда оценяването на h (9-3) (h 34 3), се представя чрез (i), а не чрез (ii).





Като следващ пример ще проследим оценяването на обръщение към функцията
`pm (x,y) = x+1` :

```
pm (3+2,4-17)
⟶ (3+2) + 1
⟶ 6
```

Тук се оценява само част (първият елемент) на двойката, която е аргумент на обръщението към функцията pm. С други думи, аргументът се използва, но се оценява само реално необходимата негова част.

Правила за оценяване и “мързеливо” оценяване

Дефиницията на една функция се състои от поредица от условни равенства. Всяко условно равенство може да съдържа множество клаузи и може да включва в специална where клауза произволен брой локални дефиниции. Всяко равенство описва в лявата си страна различни случаи на действието на функцията, която е предмет на дефиницията, т.е. описва резултата, който се връща при прилагане на функцията към различни образци.

Общ вид на дефиниция на функция

```
f p1 p2 ... pk
   | g1          = e1
   | g2          = e2
     ...
   | otherwise = er
   where
v1   a1,1    = r1
  ...
f q1 q2 ... qk
   = ...
.....
```

Оценяването на f a_1 a_2 … a_k има три основни аспекта.

1. Съпоставане по образец

Аргументите се оценяват с цел да се установи кое от условните равенства е приложимо. При това оценяването на аргументите не се извършва изцяло, а само до степен, която е достатъчна, за да се прецени дали те са съпоставими със съответните образци.

Ако аргументите са съпоставими с образците p_1, p_2, … , p_k, то оценяването продължава с използване на първото равенство; в противен случай се прави проверка за съпоставимост на аргументите с образците от второто равенство и тази проверка от своя страна може да предизвика по-нататъшно оценяване на аргументите.

Този процес продължава, докато се намери множество от съпоставими с аргументите образци или докато се изчерпат условните равенства от дефиницията (тогава се получава **`Program error`**).

Нека например е дадена дефиницията:

```
f :: [Int] -> [Int] -> Int
f []     ys     = 0                    (f.1)
f (x:xs) []     = 0                    (f.2)
f (x:xs) (y:ys) = x+y                  (f.3)
```

Тогава оценяването на израза f [1..3] [1..3] се извършва по следния начин:

```
f [1..3] [1..3]                                    (1)
⟶ f (1:[2..3]) [1..3]                              (2)
⟶ f (1:[2..3]) (1:[2..3])                          (3)
⟶ 1 + 1                                            (4)
⟶ 2
```

На стъпка (1) няма достатъчно информация, за да може да се прецени дали има съпоставимост с (f.1). Следващата стъпка на оценяване води до (2) и показва, че няма съпоставимост с (f.1).

Първият аргумент от (2) е съпоставим с първия образец от (f.2), следователно трябва да се направи проверка за втория аргумент от (2) и втория образец от (f.2). Следващата стъпка на оценяването, извършена в (3), показва, че вторият аргумент не е съпоставим с втория образец от (f.2). На тази стъпка се вижда също, че е налице съпоставимост на аргументите с (f.3), следователно е налице (4).

2. Условия (guards)

Нека за определеност предположим, че първото условно равенство от дефиницията е съпоставимо с обръщението към f, което трябва да се оцени. Тогава образците p_1, p_2, … , p_k в условното равенство се заместват с изразите a_1, a_2, … , a_k. След това трябва да се определи коя от клаузите в дясната страна е приложима. За целта условията се оценяват последователно, докато се намери първото условие със стойност True; като резултат се връща стойността на съответната на това условие клауза.

Нека например е дадена дефиницията:

```
f :: Int -> Int -> Int
f m n p
    | m>=n && m>=p = m
    | n>=m && n>=p = n
    | otherwise    = p
```

Тогава

```
f (2+3) (4-1) (3+9)
 ?? (2+3)>=(4-1) && (2+3)>=(3+9)
 ?? ⟶ 5>=3 && 5>=(3+9)
 ?? ⟶ True && 5>=(3+9)
 ?? ⟶ 5>=(3+9)
 ?? ⟶ 5>=12
 ?? ⟶ False
```

```
?? 3>=5 && 3>=12
?? ⟶ False && 3>=12
?? ⟶ False
?? otherwise   True
  12
```

3. Локални дефиниции

Стойностите в клаузите where се пресмятат при необходимост (“при поискване”): пресмятането на дадена стойност започва едва когато се окаже, че тази стойност е необходима.

Ако са дадени дефинициите:

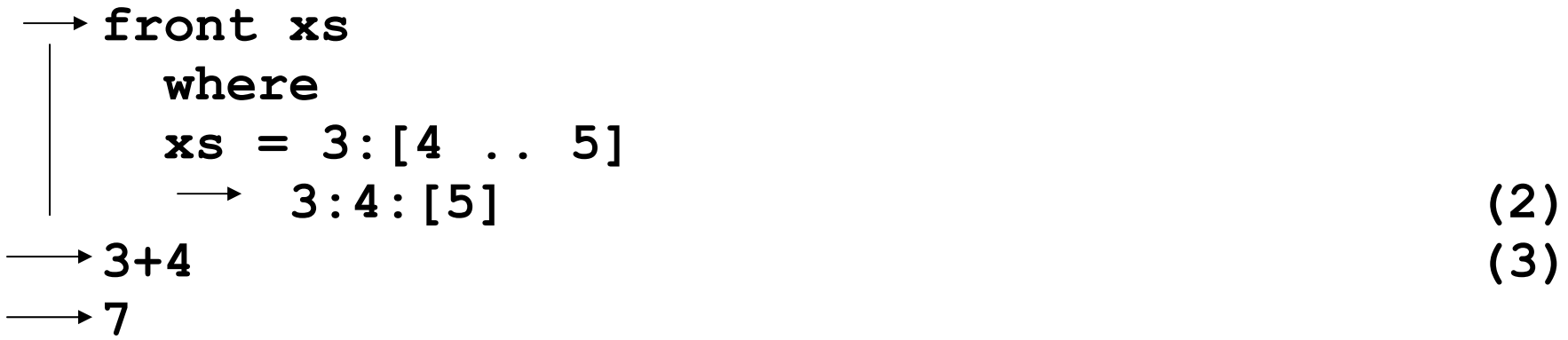
```
f :: Int -> Int -> Int
f m n
   | notNil xs = front xs
   | otherwise = n
   where
   xs = [m .. n]
```

```
front (x:y:zs) = x+y
front [x]      = x

notNil []    = False
notNil (_:_) = True
```

Тогава процесът на оценяване на f 3 5 изглежда по следния начин:

```
f 3 5
 ?? notNil xs
 ?? | where
 ?? | xs = [3 .. 5]
 ?? |   ⟶ 3:[4 .. 5]                                  (1)
 ?? ⟶ notNil (3:[4 .. 5])
 ?? ⟶ True
```



За да може да се оцени условието notNil xs, започва оценяване на xs и след една стъпка (1) показва, че това условие има стойност True.

Оценяването на front xs изисква повече информация за xs, затова оценяването на xs се извършва на (продължава с) още една стъпка и така се получава (2). Успешното съпоставяне по образец в дефиницията на front в този случай дава (3) и така се получава окончателният резултат.

Ред на оценяването

Това, което характеризира оценяването в Haskell, освен обстоятелството, че аргументите се оценяват по не повече от един път, е ***редът, в който се прилагат функции***, когато има възможност за избор.

- Оценяването се извършва в посока от външните към вътрешните изрази.

В ситуации от типа на

$$f_1\ e_1\ \underline{(f_2\ e_2\ 10)}\quad ,$$

където едно прилагане на функция включва друго, външното обръщение $f_1\ e_1\ (f_2\ e_2\ 10)$ се избира за оценяване.

- В останалите случаи оценяването се извършва в посока от ляво на дясно.

  В израза

$$\underline{f_1\ e_1} + \underline{f_2\ e_2}$$

трябва да бъдат оценени и двата подчертани израза. При това най-напред се оценява левият израз $f_1\ e_1$.