Тема 7

Функции от по-висок ред в Haskell

Шаблони на пресмятания със списъци
(Patterns of Computation over Lists)

Голяма част от функциите за работа със списъци, които разгледахме досега, могат да бъдат отнесени към малък брой типове. При това за съставянето на тези типове функции могат да бъдат използвани съответни ***шаблони на пресмятания*** (***patterns of computation***).

Примери за шаблони на пресмятания (програмни шаблони)

- Прилагане на една и съща операция върху всички елементи на даден списък (***mapping***)
  Примери:
  - Реализацията на хоризонталното завъртане flipH (в системата за манипулиране на картинки)
  - Извличането на вторите елементи на двойките от даден списък (при работата с библиотечната „база от данни“)

- Извличане на елементите на даден списък, които удовлетворяват дадено условие (***filtering***)
  Примери:
  - Извличането на тези двойки, първият елемент на които съвпада с дадено име (при работата с библиотечната „база от данни“)
  - Извличането на цифрите/буквите от даден символен низ

- Комбиниране / акумулиране на елементите на даден списък (***folding***)
  Примери:
  - Конкатенацията на елементите на даден списък от списъци
  - Събирането / умножаването на елементите на даден списък от числа

Реализацията на тези и други шаблони на пресмятания (patterns of computation) може да се извърши с помощта на подходящи ***функции от по-висок ред***.

Дефиниция

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

Наличието на средства за дефиниране и използване на функции от по-висок ред съществено увеличава изразителната сила на съответния език за програмиране.

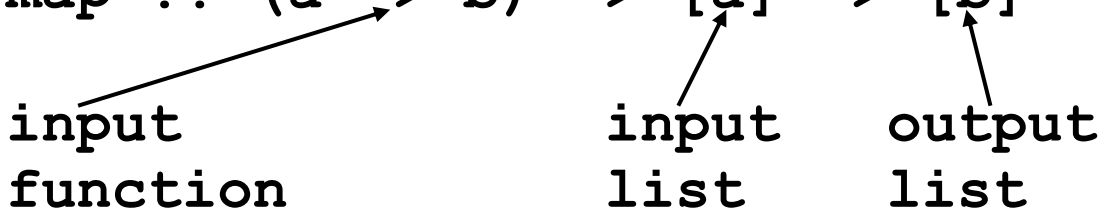
Функциите като параметри

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
```

Примери

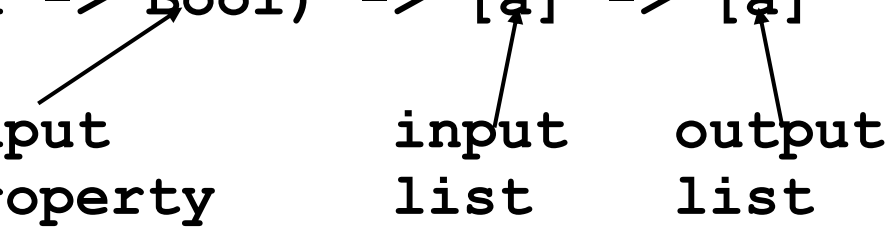
```
doubleAll :: [Int] -> [Int]
-- Удвоява елементите на даден списък.
doubleAll xs = map double xs
  where
    double :: Int -> Int
    double x = 2*x

convertChrs :: [Char] -> [Int]
-- Конвертира знаковете от даден списък
-- в техните ASCII кодове.
convertChrs xs = map ord xs
```

```
type Picture = [String]
flipH :: Picture -> Picture
-- Завърта дадена картинка около ординатната ос
-- („завърта“ хоризонталните координати на точките
-- от картинката).
flipH pic = map reverse pic
```

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
```

Примери

```
isEven :: Int -> Bool
isEven n = (n `mod` 2 == 0)

isSorted :: [Int] -> Bool
isSorted xs = (xs == iSort xs)

filter isEven [2,3,4,5] ⟶ [2,4]
filter isSorted [[2,3,4,5],[3,2,5],[],[3]]
 ⟶ [[2,3,4,5],[],[3]]
```

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]
```

и също

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

Нека се върнем още един път на примерната система за манипулиране на картинки. В нея дефинирахме функцията

```
sideBySide :: Picture -> Picture -> Picture
sideBySide p q = [pline ++ qline
     | (pline,qline) <- zip p q]
```

Нова дефиниция на функцията sideBySide:

```
sideBySide pic1 pic2 = zipWith (++) pic1 pic2
```

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
```

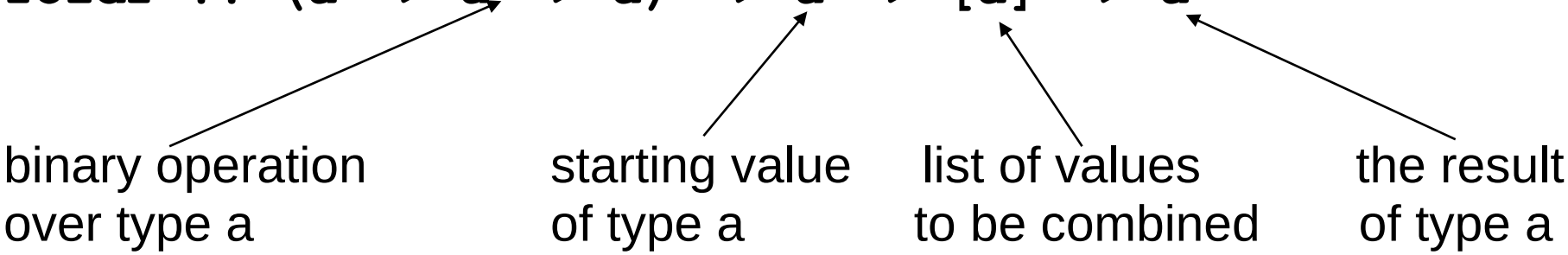
Забележка. Функцията foldr1 не е дефинирана върху втори аргумент – празен списък.

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
```




```
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
```