Тема 20. Функционално програмиране. Списъци. Потоци и отложено оценяване

1. Списъци. Представяне. Основни операции със списъци. Функции от по-висок ред за работа със списъци

Ще използваме ***Haskell***, за да покажем семантиката на списъците във ФП.

a. Списъци. Представяне.

деф. (списък)

1. Празният списък **[]** е списък от тип **[a]**
2. Ако h е елемент от тип **a** и **t** е списък от тип **[a]**, то **(h : t)** е списък от тип **[a]**
   - **h** наричаме **глава** на списъка
   - **t** наричаме **опашка** на списъка

Така списъкът е последователност с произволна дължина от елементи от еднакъв тип.

заб.

i. Някои ФП езици позволяват и нехомогенни списъци

ii. **cons - (:) :: a -> [a] -> [a]** е дясноасоциативна двуместна операция

```
["Иван", 4.5] :: ⊥ -- хетерогенност
[1]:2 :: ⊥ -- хетерогенност
[[1,2],[3],[4,5,6]] :: [[Int]] -- ок
```

b. Основни операции със списъци

- **head :: [a] -> a** – връща главата на (непразен) списък

```
head [[1, 2], [3, 4]] -->* [1, 2]
```

- **tail :: [a] -> [a]** – връща опашката на (непразен) списък

```
tail [[1, 2], [3, 4]] -->* [3, 4]
```

- **null :: [a] -> Bool** – проверява дали списък е празен
- **length :: [a] -> Int**– дължина на списък

заб. Много е удобно да използваме образци за списъци, напр. :

```
head (h : _) = h
tail (_ : t) = t
null [] = True
null _ = False
length [] = 0
length (_ : t) = 1 + length t
```

Ще разгледаме и някои по-често използвани рекурсивни функции над списъци, като примери за **рекурсия над списъци**.

- конкатенация(append)

```
(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = (x : (xs ++ ys))
```

- обръщане на списък

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x : xs) = reverse xs ++ [x]
```

- проверка за принадлежност на елемент

```
elem :: Eq a => a -> [a] -> Bool
elem _ [] = False
elem y (x : xs) = if x == y then True else elem y xs
```

- достъп до i-ти елемент

```
ithElem :: [a] -> Int -> a
ithElem (x : xs) 0 = x
ithElem (_ : xs) i = ithElem xs (i - 1)
```

c. Функции от по-висок ред за работа със списъци

- **map**(трансформация)

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f  (x : xs) = f x : map f xs
```

Напр.

```
map (^2) [1, 2, 3] → [1, 4, 9]
map (\(x : _) -> x) [[1, 2, 3], [4, 5, 6], [7, 8, 9]] → [1, 4, 7]
```

- **filter**(филтриране)

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x : xs)
| p x = x : rest
| otherwise = rest
where rest = filter p xs
```

Напр.

```
filter (\x -> x `mod` 2 == 0) [1, 2, 3, 42, 56, 69] → [2, 42, 56]
```

- **foldl**(свиване наляво) - първо обработва главата, после опашката

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ acc [] = acc
foldl op acc (x : xs) = foldl op (op acc x) xs
```

Напр.

```
myReverse :: [a] -> [a]
myReverse = foldl (\acc x -> (x : acc)) []
```

- **foldr**(свиване надясно) - първо обработва опашката, после главата

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ acc [] = acc
foldr op acc (x : xs) = op x (foldr op xs acc)
```

Напр.

```
map :: (a -> b) -> [a] -> [b]
map f lst = foldr (\x acc -> ((f x) : acc)) [] lst
```

2. Безкрайни потоци и безкрайни списъци. Основни операции и функции от по-висок ред. Отложено оценяване. Работа с безкрайни потоци

a. Понятия

деф. (Обещание)

Функция, която ще се изчисли и върне някаква стойност в бъдещ момент от изпълнението на програмата. Нарича се още promise и отложена функция.

деф. (Поток)

Списък, чиито елементи се изчисляват отложено. По-точно поток е празен списък [] или (:) x xs, където:

- x е обещание за глава
- xs е обещание за опашка

Тъй като стратегията на оценяване в Haskell е нормална(и следователно аргументите са обещания, които се изпълняват при нужда), то директно от дефиницията за поток следва, че всички списъци са потоци и всички потоци са списъци.

Затова основните операции са същите като при списъци, тук разликата е че можем да ги изпълняваме и върху ***безкрайни списъци***.

*Но невинаги има смисъл да ги прилагаме - **length** напр. няма смисъл за безкрайни потоци, оценката й няма да завърши.*

b. Генериране на безкрайни списъци(потоци)

Можем да ги генерираме както чрез list comprehension, така и чрез рекурсия.

```
iter :: (a -> a) -> a -> [a]
iter f z = z : iter f (f z)
```

Така дефинираме [z, f(z), f(f(z)), …]

```
take 10 $ iter (\x -> x `div` 2) 100
-- → [100, 50, 25, 12, 6, 3, 1, 0, 0, 0]
```

```
odds = [x | x <- [1, 2 ..], x `mod` 2 /= 0]
take 10 $ odds
-- →[1,3,5,7,9,11,13,15,17,19]
```

- **Директни дефиниции на потоци**
  Можем да използваме функциите по-висок ред за работа с потоци за директно дефиниране на потоци, напр.:

```
powers2 = 1 : map (* 2) powers2
-- Така дефинираме редицата от степените на двойката
take 5 $ powers2 → [1,2,4,8,16]

notdiv k = filter (\x -> x `mod` k > 0) [1 ..]
take 5 $ notdiv 3 → [1,2,4,5,7]

-- Така  дефинираме редицата от числата на Фибоначи
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
take 8 $ fibs → [0,1,1,2,3,5,8,13]
```