**Тема 11. Функционално програмиране. Основни конструкции в езиците за функционално програмиране. Функции. Списъци. Функции от по-висок ред. “Мързеливо” (lazy) оценяване.**

**Характерни особености на функционалния стил на програмиране. Основни компоненти на функционалните програми. Дефиниции. Изрази. Програмиране на условия. Рекурсивни и итеративни изчислителни процеси.**

Програмирането във функционален стил се състои от дефиниране на функции и прилагане (апликация) на функции върху подходящи аргументи, които могат да бъдат функции или обръщения към функции.

Предимства:
- програмира се на високо ниво на абстракция, което намалява опасността от допускане на технически грешки;
- може да се извършва лесна проверка и поправка на съответните програми поради липсата на странични ефекти;
- могат да бъдат доказвани строго (с математически средства) свойства на функционалните програми.

Недостатъци:
- строгата функционалност понякога изисква многократно пресмятане на едни и същи изрази;
- донякъде е неестествено използването функционален стил за решаване на задачи от процедурен (алгоритмичен) характер.

**Функцията** е програмна част, която пресмята и връща стойност. Може да се създават (дефинират) функции с различен брой аргументи (0, 1, 2 или повече). Аргументите на функцията, както и върнатата от нея стойност, могат да бъдат от различни типове. Строгите или “чисти” функции нямат никакъв страничен ефект - обектите в строгите функционални езици (напр. Haskell) не се изменят и на практика нямат състояние. Наричат се още immutable.

**Величините** са средство за изразяване на стойностите, с които работи една програма. Характеризират с име, тип и текуща стойност. Типът представлява множество от допустими стойности заедно с определена съвкупност от операции, приложими върху тези стойности.

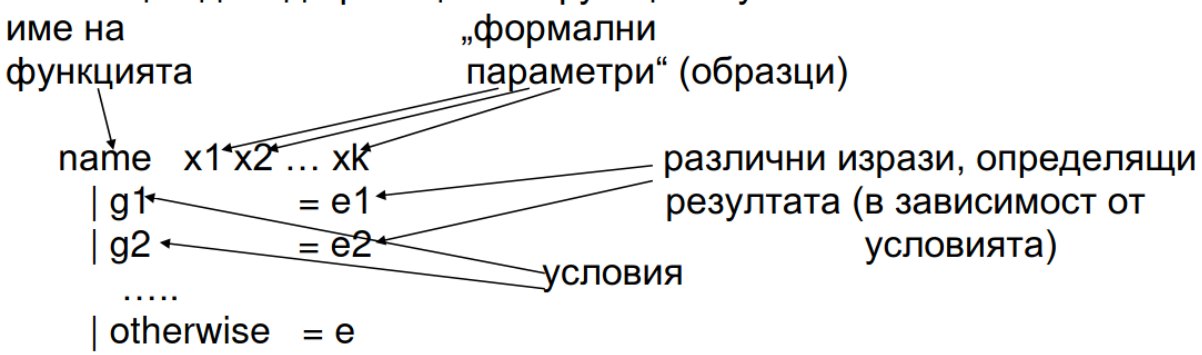
Дефинирането на функция с параметри има следния вид:

name x1 x2 ... xk = e

, където x1, x2, ..., xk са формални параметри, а `e` е израз, който указва правилото за пресмятане на връщаната стойност. Дефиницията на функция може да включва поредица от изрази от посочения вид, с помощта на които се описват отделните случаи на действието на функцията. В тях съответните x1, x2, ..., xk се наричат образци.

Условието („охраняващ“ израз, guard) е Булев израз. Условия се използват, когато трябва да се опишат различни случаи в дефиницията на функция.

Общ вид на дефиниция на функция с условия:




Когато трябва да се приложи дадена функция към дадено множество от изрази, е необходимо да се установи кой от поредните случаи в дефиницията на функцията е приложим. За да се отговори на този въпрос, последователно се оценяват охраняващите изрази, докато се достигне до първия срещнат, чиято оценка е True. Съответният израз от дясната страна на равенството определя резултата.

Рекурсията е техника за програмиране, при която в дефиницията на дадена функция има обръщение към същата функция. Съществуват два типа изчислителни процеси, за описанието на които се използват функции, които са синтактично рекурсивни.

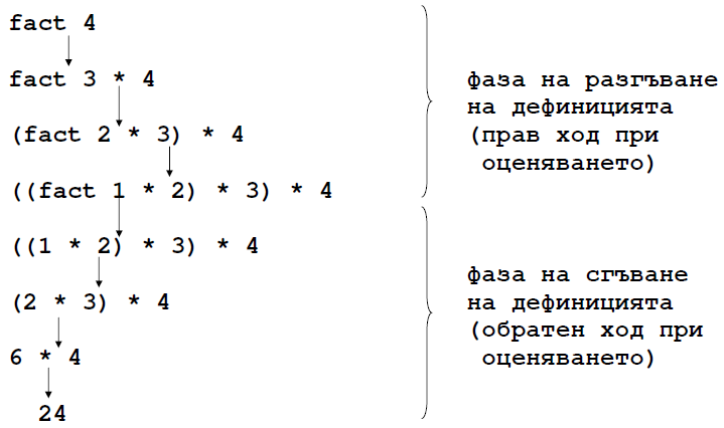
Процесите от първия тип се наричат **рекурсивни**. При тях се поражда верига от извиквания на дефинираната функция с все по-прости аргументи, докато се достигне базов (граничен) случай, след което започва обратно пресмятане на генерираните вече обръщения. Например, в задачата за пресмятане на факториел имаме:

- 0! = 1 (базов случай)
- n! = n * (n - 1)! , за всички n>2 (общ случай)

Примерна реализация на Haskell:

```
fact 0 = 1
fact n = fact (n-1) * n
```

Оценяването на израз, обръщение към горната функция с параметър 4, изглежда така:




Процесите от втория тип се наричат **итеративни**. При тях е нужна допълнителна функция с аргументи, типични за тялото на цикъл (state variables):

- брояч (counter) – променлива, която да държи броя на итерациите (извикванията)
- текуща стойност (result) – променлива, която да държи резултатът от умножението на текущата итерация

В примера за факториела първоначално result и counter ще имат стойност 1 и ще се изменят на всяка итерация като: counter = counter + 1, а product = product * counter.

```
fact2 :: Integer -> Integer
fact2 n = f_iter n 1 1

f_iter :: Integer -> Integer -> Integer -> Integer
f_iter n counter product
 | (counter > n) = product
 | otherwise = f_iter n (counter + 1) (product*counter)
```

Тук процесът на оценка няма фази и броя на операциите е константен:
fact2 4 -> f_iter 4 1 1 -> f_iter 4 2 2 -> f_iter 4 3 2 -> f_iter 4 4 6 -> f_iter 4 5 24 -> 24

**Съставни типове данни. Вектори. Списъци. Локални дефиниции. Програмиране със списъци.**

**Векторът** (tuple) представлява наредена n-торка от елементи, при това броят n на тези елементи и техните типове трябва да бъдат определени предварително. Допуска се елементите на векторите да бъдат от различни типове. Пример: Намиране на минималното и максималното от две цели числа.

```
minAndMax :: Int -> Int -> (Int,Int)
minAndMax x y
 | x>=y = (y,x)
 | otherwise = (x,y)
```

**Списъкът** в Haskell е редица от (променлив брой) елементи от определен тип. За всеки тип t в езика е дефиниран също и типът [t], който включва списъците с елементи от t.
Запис на списъците в Haskell:
[ ]: празен списък (списък без елементи). Принадлежи на всеки списъчен тип.
[e1,e2, ... , en]: списък с елементи e1, e2, ... , en.

Дефиниране на списък чрез аритметична последователност (arithmetic sequence):

- [n .. m] е списъкът [n, n+1, ... , m]; ако n>m, списъкът е празен.

[2 .. 7] = [2,3,4,5,6,7]
[‘a’ .. ‘d’] = “abcd”

- [n,p .. m] е списъкът, чийто първи два елемента са n и p, последният му елемент е m и стъпката на нарастване на елементите му е p-n.

[7,6 .. 3] = [7,6,5,4,3]

Възможно големината на стъпката да не позволява достигането точно на m. Тогава последният елемент на списъка съвпада с най-големия/най-малкия елемент на редицата, който е по-малък/по-голям или равен на m.

Дефиниране на списък чрез определяне на неговия обхват (List Comprehension)

Синтаксис:
**[expr | q1, ... , qk]** , където **expr** е израз, а **qi** може да бъде:

- **генератор** от вида **p <- lExp**, където **p** е образец и **lExp** е израз от списъчен тип
- **тест (филтър)**, **bExp**, който е булев израз

В qi могат да участват променливите, използвани в q1,q2, ... , qi-1.

Ако предположим, че стойността на ex е [2, 4, 7, 12] то:
- [2*n | n <- ex] е кратък запис на списъка [4, 8, 14, 24]
- [isEven n | n <- ex] е кратък запис на списъка [True, True, False, True].
- [2*n | n <- ex, isEven n, n>3] се оценява до списъка [8, 24]

**Локалните дефиниции** са начин, по който могат да се дефинират помощни функции, чиито резултати да бъдат преизползвани в други функции.

Общ вид на дефиниция на функция с използване на условия с клауза where:

```
f p1 p2 ... pk
 | g1 = e1
 . . .
 | otherwise = er
  where
   v1 a1 ... an = r1
   v2 = r2
   . . . . .
```

Клаузата where тук е присъединена към цялото условно равенство, т.е. към всички негови клаузи.

Пример: Дефиниция на функция, която връща като резултат сумата от квадратите на две числа.

```
sumSquares :: Int -> Int -> Int
sumSquares n m = sqN + sqM
 where
  sqN = n*n
  sqM = m*m
```

Възможно е да се дефинират локални променливи с област на действие, която съвпада с даден израз.

Например изразът

*let x = 3+2 in x^2 + 2*x – 4*

има стойност 31.

Ако в един ред са включени повече от една дефиниции, те трябва да бъдат разделени с точка и запетая, например

*let x = 3+2; y = 5-1 in x^2 + 2*x -y*

Областта на действие на дадена дефиниция съвпада с частта от програмата, в която може да се използва тази дефиниция. Всички дефиниции на най-високо ниво в Haskell имат за своя област на действие целия скрипт, в който са включени. В частност те могат да бъдат използвани в дефиниции, които се намират преди техните собствени в съответния скрипт. Локалните дефиниции, включени в дадена клауза where, имат за област на действие само условното равенство, част от което е клаузата where.

**Съпоставяне по образец. Видове образци. Примитивна и обща рекурсия върху списъци.**

**Образецът** е езикова конструкция, с помощта на която се описва отделен възможен случай за даден аргумент. В ролята на образец може да се използва:

- Литерал като например 24, ‘f’ или True; даден аргумент се съпоставя успешно с такъв образец, ако е равен на неговата стойност;
- Променлива като например x или longVariableName; образец от този вид се съпоставя успешно с аргумент с произволна стойност;
- Специален символ за безусловно съпоставяне (wildcard) ‘_’, който е съпоставим с произволен аргумент;
- Вектор – образец (p1,p2, ... ,pn); за да бъде съпоставим с него, аргументът трябва да има вида (v1,v2, ... ,vn), като всяко vi трябва да бъде съпоставимо със съответното pi;
- Конструктор, приложим към даденото множество от аргументи.

Всеки списък или е празен (т.е. е равен на/съпоставим с [ ]), или е непразен. Във втория случай списъкът има глава и опашка, т.е. може да бъде записан във вида x:xs, където x е първият елемент на този списък, а xs е списъкът без първия му елемент (опашката на този списък). Операторът ‘:’ (наричан cons) от тип a -> [a] -> [a] е конструктор на списъци и всеки списък може да бъде конструиран по единствен начин с използване на [ ] и ‘:’.

Образецът от тип конструктор на списъци може да бъде или [ ], или да има вида (p:ps), където p и ps също са образци. Правилата за съпоставяне с такъв образец могат да бъдат формулирани по следния начин:

- даден списък е съпоставим с образеца [ ] точно когато е празен;
- даден списък е съпоставим с образец от вида (p:ps), когато не е празен, главата му се съпоставя успешно с образеца p и опашката му се съпоставя успешно с образеца ps.

Всеки непразен списък е съпоставим с образец от вида (x:xs).

В дефиниция, която реализира **примитивна рекурсия върху списъци**, се описват следните типове случаи:

- начален (прост, базов) случай: в горния пример тук се описва стойността на sum за [ ];

- общ случай, при който се посочва връзката между стойността на функцията за дадена стойност на аргумента (в случая sum (x:xs)) и стойността на функцията за по-проста в определен смисъл стойност на аргумента (sum xs).

Общата схема на дефиниция на функция чрез примитивна рекурсия върху списъци е следната:

```
fun [] = ....
fun (x:xs) = .... x .... xs .... fun xs ....
```

Пример: Примерна дефиниция на функцията concat от Prelude.hs:

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

При функциите, използващи **обща рекурсия върху списъци**, схемата на дефиниране е специфична и се подчинява на задачата да се даде отговор на въпроса: Когато се дефинира f (x:xs), кои стойности на f ys биха помогнали за коректното описание на резултата?

Пример: Примерна дефиниция на функцията zip от Prelude.hs:

```
zip :: [a] -> [b] -> [(a,b)]
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _ = []
```

С други думи, характеристиките на общата рекурсия върху списъци са:
- използват се поне два списъка;
- базовият случай може и да не зависи само от списъка;
- съществува повече от едно рекурсивно извикване.

**Функции от по-висок ред. Функциите като параметри. Дефиниции на функции на функционално ниво. Функциите като върнати стойности. Частично прилагане на функции.**

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

Пример 1: Прилагане на дадена функция към всички елементи на даден списък (map)

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

Тук (a -> b) е типът на приеманата функция, т.е. се приема **функция като параметър**.

Пример 2: Филтриране на елементите на даден списък (filter)

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]
```

Пример 3: Комбиниране/акумулиране на елементите на даден списък (foldr1 и foldr)
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 [e1,e2, ... , ek]
= e1 `f` (e2 `f` ( ... `f` ek) ... )
= e1 `f` (foldr1 f [e2, ... , ek])
= f e1 (foldr1 f [e2, ... , ek])

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)
```

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 right)

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f s [] = s
foldr f s (x:xs) = f x (foldr f s xs)
```

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 може да се дефинира с използване на вградения оператор '.' както следва:

```
fxg f g = (f . g)
```

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.

Пример: Дефиницията на функция, която **връща функция**, с използване на ламбда израз:

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

Всяка функция на два или повече аргумента може да бъде **приложена частично** към по-малък брой аргументи.

Пример: Функция, която удвоява всички елементи на даден списък от цели числа:

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

Когато бъде комбинирана с функции от по-висок ред, нотацията на сечението на оператори 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.

**Оценяване на изрази. “Мързеливо” (lazy) оценяване. Работа с безкрайни списъци.**

"Мързеливото" оценяване (lazy evaluation) е стратегия на оценяване, според която интерпретаторът оценява даден аргумент на дадена функция само ако (и доколкото) стойността на този аргумент е необходима за пресмятането на целия резултат. Ако аргументите са съставни се оценяват само тези компоненти, чиито стойности са необходими за получаването на резултата. Дублиращи се подизрази се оценяват по не повече от един път.

Когато се оценява израз - обръщение към функцията, се оценява частен случай на тялото на дефиницията, в който формалните параметри са заместени с подадени аргументи. Например ако f x y = x+y, то :

f (9-3) (f 34 3) се оценява до (9-3) + (f 34 3)

Изразите (9-3) и (f 34 3) не се оценяват преди да има реална нужда от техните стойности. В конкретния случай ще се наложи оценка и на двата фактически параметъра, но това не винаги е така. Например за g x y = x + 5:

g (9-3) (g 34 3) се оценява до (9 - 3) + 5

Това е едно от преимуществата на „мързеливото“ оценяване – ненужните аргументи не се оценяват:

```
switch n x y
 | (n>0) = x
 | otherwise =  y
```

Тук при всяко извикване ще се оценяват стойностите на параметрите n и само един от изразите x и y.

Ако pm (x, y) = x+1, pm (3+2, 4-17) се оценява до (3+2) + 1. Макар че аргументът е вектор, отново се оценява само частта нужна за пресмятане на резултата.

Когато извикваме функция и се опитваме да съпоставим образец, аргументите трябва да се оценят с цел да се установи кое от условните равенства е приложимо. Оценяването на аргументите обаче не се извършва изцяло, а само до степен, която е достатъчна, за да се прецени дали те са съпоставими със съответните образци. Когато се намери подходящо равенството, оценяването продължава с директното му прилагане.

Едно важно следствие от "мързеливото" оценяване в Haskell е обстоятелството, че езикът позволява да се работи с **безкрайни структури**. Пълното оценяване на такава структура изисква безкрайно време, т.е. не може да завърши, но механизмът на "мързеливото" оценяване позволява да бъдат оценявани само тези части ("порции") на безкрайните структури, които са необходими.

Пример: Безкраен списък от единици.

```
ones :: [Int]
ones = 1 : ones
```

Оценяването на ones ще продължи безкрайно дълго и следователно ще трябва да бъде прекъснато от потребителя. Възможно е обаче съвсем коректно да бъдат оценени обръщения към функции с аргумент ones:

```
addFirstTwo :: [Int] -> Int
addFirstTwo (x:y:zs) = x+y
```

Тогава

```
addFirstTwo ones
-> addFirstTwo (1:ones)
-> addFirstTwo (1:1:ones)
-> 1+1
-> 2
```