Функционалният стил в програмирането (в примери на Haskell)

Б. Банчев

Характерни особености на функционалния стил:

- интензивно използване на функции; функции-стойности (аргументи, резултати); множество начини за образуване на стойности-функции (композиране, частично прилагане, ...)
- декларативно описване на действия, включително определения на функции; (по принцип) отсъстват императивни конструкции като присвояване и цикли; изобщо, отсъства понятието „оператор“ или „команда“, има само изрази
- динамично (в хода на програмата) образуване на стойности, включително съставни и дори такива с неопределено голям обем (рекурсивни)
- пряко (без прибягване до указатели или моделиране с други средства) представяне на разнообразни съставни стойности: n-ки, списъци, дървета и др.; съставните стойности се третират цялостно
- значително се улеснява формалното третиране на програмата като математически обект; много по-удобно се изказват и доказват твърдения за свойствата на програмата, за преобразования, на които тя може да бъде подлагана и т. н.
- става възможно истински да се отделя описването на действията от това на данните, върху които те се прилагат; с това програмата става по-лесно управляема (разбирана, изменяна), а нейните части – по-абстрактни и следователно по-универсални

Съвременните функционални езици имат следните черти:

- строга типовост; всяка стойност има определен тип и не се допуска използване на един тип в ролята на друг (чрез неявно или явно преобразуване)
- развита типова система (разнообразие от предопределени и въвеждани от програмиста типове) с автоматична изводимост на типовете (от изпълняващата система) и естествена полиморфност (многозначност)

- развита декларативност по отношение на задаването и на данни, и на действия с тях; това позволява например по-рядко да се извършват проверки за валидност, избиране на алтернативи и др. под. по време на действието на програмата, а и дори когатотакива действия са неизбежни, те да бъдат добре организирани, а не потенциално хаотични; за целта се използват разнообразни механизми, например структурнодекомпозиционно съпоставяне и охраняеми действия

- богат, адекватен на високото семантично (смислово) равнище на езика, синтаксис

- тенденция към избягване на непосредственото използване на рекурсия, като се въвеждат подходящи функции от висок ред (такива, чиито аргументи и/или резултати са също функции)

- средства за създаване на абстракции от различен вид, вкл. абстрактни даннови типове

- средства за модулно програмиране

- при част от езиците (Haskell, Clean, Miranda) – отложени пресмятания и работа с (потенциално) безкрайни даннови обекти

1 Списъци: пряко задаване, прогресии, генератори

```
[3,2,-4,7]                      -- списък от цели числа
3:2:-4:7:[]                     -- същият списък, с изрично прилагане
                                -- на конструкторите ([] и :)
3:2:[-4,7]                      -- все същият списък: към [-4,7] се
                                -- добавят (отпред) 2 и 3
[]                              -- празен списък
[[2,5,1],[5,-3,0,12],[],[3]]    -- списък от списъци от цели числа

[('b',5,"един низ"),('d',7,"друг низ")]  {- списък от n-ки, именно --
                                 тройки от тип (Char,Integer,String) -}
[2,if b then x else y,8,f 15]   -- могат да участват произволни изрази

[5..14]               -- еквив. на [5,6,7,8,9,10,11,12,13,14]
[2.7..9.1]            -- еквив. на [2.7,3.7,4.7,5.7,6.7,7.7,8.7]
[5,7..14]             -- еквив. на [5,7,9,11,13] (нарастване 2)
[5.1,4.7..2.5]        {- еквив. на [5.1,4.7,4.3,3.9,3.5,3.1,2.7]
                         (нарастване -0.4)                    -}
['d','g'..'r']        {- еквив. на "dgjmp" (низовете са списъци от
                         литери, които също са наредено множество) -}

[x| x <- nub "differentiate", x `elem` "aeiou"]
                     {- списък (низ) от гласните от низа,
                       всяка по веднъж ("iea") -}
```

```
[(x,y)| x<-[5,8,3], y<-['#','*']]
                      {- еквив. на [(5,'#'),(5,'*'),(8,'#'),
                                    (8,'*'),(3,'#'),(3,'*')] -}
[(x, ord x)| x <- sort (nub "Use Haskell, not Pascal!"), isLower x]
                      {- списък от лексикографски подредените
                        малки букви от низа, всяка по веднъж,
                        по двойки с номерата им:
                        [('a',97),('c',99),('e',101), ...]   -}
xs == [x| x<-xs, x`mod`5 == 0]
                      {- true (false), когато всички елементи
                        на xs се (не се) делят на 5          -}
```

2 Определяне на функции: уравнения, съпоставяне, охрани, срезове

```
max3 a b c = max (max a b) c     -- префиксно обръщение
max3 a b c = a `max` b `max` c   -- същото в инфиксен запис

max0 = max 0                     {- частично прилагане на max;
                                   еквив. на  max0 x = max 0 x -}

gt:: Int -> Int -> Bool          -- изрично задаване на тип
gt = (>)                         -- срез на >:  gt x y  е еквив. на
                                 -- x>y  и на  (>) x y
pos = (>0)                       -- ляв срез:  pos x  е еквив. на  x>0
neg = (0>)                       -- десен срез:  neg x  е еквив. на  0>x

                                 -- дължина на списък:
length []     = 0                -- двойка уравнения с шаблони за
length (_:xs) = 1 + length xs    -- разграждане на списък по
                                 -- конструкторите му

euc a b                          -- намиране на н.о.д. (алг. на Евклид):
  | a<b  = euc a (b-a)           -- определение чрез поредица от
  | a>b  = euc (a-b) b           -- ,,охраняеми изрази``
  | a==b = a

fib n                            -- n-то число на Fibonacci:
  | n==0 = 0                     -- пряка рекурсия
  | n==1 = 1
  | n>1  = fib(n-2)+fib(n-1)

fib' = fst . fpair  where            -- използване на стойности от вида
  fpair n                            -- двойка (,), на локално определени
    | n==0      = (0,1)              -- функции (where) и на композиция
    | otherwise = fstep(fpair(n-1))  -- (.)
    where  {fstep(a,b) = (b,a+b)}
```

```
                                    -- ,,бързо‘‘ подреждане
qsort []     = []
qsort (x:xs) = qsort [y| y<-xs, y<=x] ++ [x] ++ qsort [y| y<-xs, y>x]
```

3 Етюд върху факториела: няколко определения

```
rfac n = if n==0 then 1 else n * rfac (n-1)  -- пряка рекурсия

efac 0 = 1                                   -- двойка уравнения,
efac n = n * efac (n-1)                      -- едното рекурсивно

cfac = \n -> case n of                       -- cfac се определя като
         0     ->  1                         -- равна на безименна функция,
         (m+1) ->  (m+1) * cfac m            -- взаимнорекурсивна с cfac

pfac n = product [1..n]                      {- ,,просто‘‘ определение:
                                                вж. по-долу определението
                                                на product              -}
```

4 Полиморфни функции

```
id x = x              -- идентитет, от тип  a -> a
fst (x,y)   = x       -- ,,първи‘‘, от тип  (a,b) -> a
snd (x,y)   = y       -- ,,втори‘‘, от тип  (a,b) -> b
head (x:_)  = x       -- ,,глава‘‘, от тип  [a] -> a
tail (_:xs) = xs      -- ,,опашка‘‘, от тип [a] -> [a]
```

Функцията *композиция* (`.`), определена с `f . g = \x -> f (g x)` има тип `(.) :: (b -> c) -> (a -> b) -> a -> c` .

Също полиморфна е вече срещаната функция `length`, както и функциите `map`, `filter`, `zipWith`, които се срещат в следващите раздели – все стандартно предоставяни с Haskell функции, а също и доста от функциите, които въвеждаме в примерите.

5 Типове-синоними. Използване на типове при определяне на функции

```
type Name = String              -- Name става синоним на String
type Table a b = (a,[b])        -- параметризирано определение на синоним
type Index = Table String Int   -- определение чрез вече въведен синоним
```

Определяне и използване на „база от данни“ за библиотечно заемане. NB! Определенията на типове за функциите са съществени

```
type Person = String            -- заемател (име)
type Book = String              -- заглавие на книга
```

```
type Library = [(Person,Book)]  -- списък от записи заемател-книга:
                                -- състояние на справочника за
                                -- заеманията

        -- кои книги от db са заети от prs:
books :: Library -> Person -> [Book]
books db prs = [b| (p,b)<-db, p==prs]

        -- кои заемателя са заели от db книгата bk:
borrowers :: Library -> Book -> [Person]
borrowers db bk = [p| (p,b)<-db, b==bk]

        -- заемане на книга (състоянието на справочника се променя;
        -- всъщност получаваме нов справочник!):
mkLoan :: Library -> Person -> Book -> Library
mkLoan db prs bk = (prs,bk):db

        -- връщане на книга (както по-горе, получаваме нов справочник!):
rtLoan :: Library -> Person -> Book -> Library
rtLoan db prs bk = [x| x<-db, x/=(prs,bk)]

        -- примерно състояние на справочника за библиотека
lib =
  [("Georgi","Algoritmi"), ("Ina","Voyna i mir"),
   ("Georgi","Programirane"), ("Diana","Divi razkazi"),
   ("Kamen","Voyna i mir")]
```

6 „Конструирани“ типове

```
        -- ,,изброяване‘‘ (три конструктора, всичките без аргументи)
data Colours = Red | Green | Blue

        {- тип, описващ множество;  два конструктора -- NilSet без
           аргументи, за празно множество, и ConsSet с два аргумента
           (по-вярно, ConsSet е конструктор с един аргумент и с резултат
           функция, която (трябва да) добавя съответния елемент (от тип a)
           към множеството (от тип Set a)) -}
data Set a = NilSet | ConsSet a (Set a)

        {- косвено рекурсивно определение;  човек (Person) е или дете
           (Child), или възрастен (Adult);  във втория случай биографията
           се допълва от (възможно празен) списък от деца, които са също
           Person -}
data Person = Child Name | Adult Name Address Biog
data Biog   = NonParent String | Parent String [Person]

        -- стойностите на типа са ,,единични‘‘ (S) или ,,двойни‘‘ (P):
data SingleOrPair a b = S a | P a b
```

```
        -- примерен списък от горния тип:
xs = [S 1, P 2 "ab", P 3 "cde", S 4, P 5 "f"]
        {- генератор, избиращ измежду елементите на xs само онези,
          които отговарят на конструктора P -}
[y| P x y <- xs]            -- резулт. е ["ab","cde","f"]
```

7 Етюд: ефективно реализиране на опашка чрез два списъка

Единият списък е за добавяне в опашката, другият – за извличане от нея. Когато вторият списък се окаже празен, първият се копира в обратен ред в него, а самият той става празен.

```
data Queue a = Qu [a] [a]

emptyQ = Qu [] []

isEmptyQ (Qu [] []) = True
isEmptyQ _          = False

addQ x (Qu xs ys) = Qu (x:xs) ys

remQ (Qu xs (y:ys)) = (y, Qu xs ys)
remQ (Qu xs [])     = remQ (Qu [] (reverse xs))
remQ (Qu [] [])     = error "remQ"
```

8 Класове, представители. Наследено представяне

```
class Eq a where                          -- клас за типове a, чиито
  (==), (/=) :: a -> a -> Bool            -- стойности могат да се
  x/=y = not (x==y)                       -- сравняват за равенство
  x==y = not (x/=y)

data Ordering = LT | EQ | GT       -- тип, чиито стойности са
                                   -- наредбени отношения

class (Eq a) => Ord a where                    -- типовете с наредба (Ord)
  compare              :: a -> a -> Ordering   -- предполагат и равенство
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a

data Bool = False | True           -- Булевият тип
instance Eq Bool where             -- описване на Bool като представител
  True==True   = True              -- на класа Eq
  False==False = True
  _==_         = False
```

```
        -- примерът за множество от по-горе, усъвършенстван
data Eq a => Set a = NilSet | ConsSet a (Set a)

        -- две двоични дървета, чийто тип е определен както тук,
        -- могат да бъдат сравнявани за равенство, ако това е изпълнено
        -- за стойностите във възлите им:
data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Eq
```

9 Етюд: пермутации, подредици, подсписъци

```
-- perm1 :: Eq a => [a] -> [[a]]
perm1 [] = [[]]
perm1 xs = [x:ps| x<-xs, ps<-perm1(xs\\[x])]

-- perm2 :: [a] -> [[a]]
-- типът е по-общ от този на perm1, защото не се изисква
-- принадлежност към класа Eq
perm2 []     = [[]]
perm2 (x:xs) = [ps++[x]++qs| rs <- perm2 xs, (ps,qs) <- splits rs]
  where
  splits []     = [([],[])]
  splits (y:ys) = ([],y:ys) : [(y:ps,qs)| (ps,qs) <- splits ys]

-- списък от последователните подредици, в лексикографски ред
subSeq [] = [[]]
subSeq xs = [take n ys| ys <- sufSeq xs, n <- [1 .. length ys]]

-- списък от суфиксите, от най-дългия към най-късия (лекс. ред)
sufSeq [] = [[]]
sufSeq xs = [drop n xs| n <- [0 .. length xs - 1]]

-- списък от префиксите, от най-късия към най-дългия (лекс. ред)
-- prfSeq [] = [[]]
-- prfSeq xs = [take n xs| n <- [1 .. length xs]]

-- списък от подсписъците (лекс. ред)
subLst []     = [[]]
subLst [x]    = [[x]]
subLst (x:xs) = [x] : [y++ys| y <- [[x],[]], ys <- subLst xs]
```

10 Функции от висок ред

```
map f []     = []
map f (x:xs) = f x : map f xs

curry f x y = f (x,y)
uncurry f p = f (fst p) (snd p)
```

```
flip f x y = f y x

-- foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v []      =  v                        -- ,,прилагане отдясно‘‘
foldr f v (x:xs)  =  f x (foldr f v xs)

-- foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v []      =  v                        -- ,,прилагане отляво‘‘
foldl f v (x:xs)  =  foldl f (f v x) xs

-- unfold :: (a->Bool) -> (a->b) -> (a->a) -> a -> [b]
unfold p f g x                                -- ,,образуване на списък‘‘
  | p x       = []
  | otherwise = f x : unfold p f g (g x)

-- unfold' :: (a->b) -> (a->a) -> a -> [b]
unfold' f g x = f x : unfold' f g (g x)     -- винаги безкрайни списъци
-- или само:  unfold' = unfold (\_ -> False)
```

Функциите от висок ред често служат за пряко, нерекурсивно, определяне на други функции (рекурсията се поглъща от тях), за получаване на комбинàторна форма на определение на функция и други преобразования. Следват примери за това.

```
abslist = map abs        -- списък от абсол. стойности на елем. на списък
posits = filter (>0)     -- списък само от положителните елем. на списък

sum = foldr (+) 0                           -- сбор и произведение на
product = foldr (*) 1                       -- целочислен списък

length = foldl (\n _ -> n+1) 0              -- дължина на списък
reverse = foldl (flip (:)) []               -- обръщане на списък

map f = foldr ((:).f) []                    -- вж. по-горе
map f = unfold' (f.head) tail               -- (друго определение)

-- ,,пресяване‘‘ на списък относно функция-предикат p:
filter p = foldr (\a b -> if p a then a:b else b) []

map f xs = [f x| x<-xs]                     -- много рекурсивни функции
filter p xs = [x| x<-xs, p x]               -- могат да се изразяват и
                                            -- чрез генератори на списъци

isort = foldr ins []                        -- подреждане чрез вмъкване
  where
  x `ins` [] = [x]
  x `ins` yy@(y:ys)
    | x<=y      = x:yy
    | otherwise = y: x `ins` ys
```

```
poly c x = foldl (\a b -> a*x+b) 0.0 c     -- пресмятане на полином с в x
```

11 Отложено (лениво) пресмятане. Безкрайни редици

Правила, по които се извършват пресмятанията в Haskell:

(1) пресмята се „отвън навътре“ и „отляво надясно“;

(2) всеки аргумент се пресмята само ако и тогава, когато стойността му се използва (в някой пресмятан израз);

(3) пресмятането на аргумент може да е частично (от гледна точка на частите му и на съответните функции-конструктори, които ги свързват);

(4) всеки аргумент бива пресметнат (ако изобщо бива) само по веднъж.

```
-- iterate :: (a -> a) -> a -> [a]
iterate f x  =  x : iterate f (f x)
-- друго определение:  iterate f = unfold' id f

-- a -> [a]
repeat x = x : repeat x
-- друго определение:  repeat x = xs  where xs = x:xs
-- друго определение:  repeat = iterate id
-- друго определение:  repeat = unfold' id id

        -- сбор от квадратите на елементите на списък;  въпреки
        -- привидното, не се образуват списъци от прилаганията на map
        -- върху суфиксите на списъка-аргумент
sum2p = sum . (map (^2))

        -- намиране на минимум (ефективно поради ленивото пресмятане)
findMin = head . isort

        -- съществено различни (от вз. прости числа) питагорови тройки
pyTriple = [(x,y,z)| z <- [2..], y <- [2..z-1], x <- [2..y-1],
                     x*x+y*y==z*z,  x `gcd` y `gcd` z == 1]

        -- редицата на Fibonacci
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

        -- редицата на простите числа чрез решетото на Ератостен
primes = map head (iterate sieve [2..])  where
  sieve (p:xs) = [x| x<-xs, x `rem` p /= 0]
```

12 Етюд: представяне и действия с графи

Определения на типове за представяне на граф:

```
type RGraph a = [(a,a)]       -- (ор)граф като множество (списък) от дъги
type ELGraph a = [(a,[a])]    {- (ор)граф като множество (списък) от
                  двойки връх-съседи, а съседите са множества-списъци -}
```

Определения на имената и типовете на функции върху графи:

```
        -- построяване на ELGraph от RGraph:
mkGraph :: RGraph Gnode -> ELGraph Gnode
        -- съседите на връх (или празен списък):
nbrs :: ELGraph Gnode -> Gnode -> [Gnode]
        -- добавяне на дъга към граф (резултатът е ,,уголемен‘‘ граф):
addEdge :: ELGraph Gnode -> Gnode -> Gnode -> ELGraph Gnode
        -- списък от пътищата в граф от един даден възел към друг:
routes :: ELGraph Gnode -> Gnode -> Gnode -> [[Gnode]]
```

Определения на функциите:

```
mkGraph = foldr (\(x,y) v -> addEdge v x y) []

nbrs g a = foldr (\(b,bs) v -> if b==a then bs else v) [] g

addEdge g a b
  | nbrs g a == []  = (a,[b]):g
  | otherwise       = foldr (\z@(x,xs) v ->
                               (if x==a then (x,b:xs) else z) : v)
                            [] g

routes = routes' []  where
  routes' v g a b
    | a == b    = [[a]]
    | otherwise = [a:r | x <- nbrs g a \\ v, r <- routes' (a:v) g x b]
```

Примерен граф (върховете са литери (`Char`)) и намиране на пътища в него:

```
type Gnode = Char

g = mkGraph [('a','b'),('b','d'),('c','f'),('c','e'),('e','f'),('a','c')]
routes g 'c' 'f'
```

Както по-горе за функцията `findMin`, `routes` може да се използва за намиране на *един* (кой да е) път в граф: `head . routes`.