Programowanie funkcyjne

HASKELL

Różne operacje na listach

```
ciagRosnacy pocz kon krok
    | pocz > kon = []
    | otherwise = pocz : ciagRosnacy ( pocz+krok ) kon krok
```

```
*Main> :t ciagRosnacy
ciagRosnacy :: (Ord t, Num t) => t -> t -> t -> [t]
```

Klasa Ord to typy, których wartości możemy porównywać (używamy w definicji >)

```
*Main> ciagRosnacy 1 22 3
[1,4,7,10,13,16,19,22]
*Main> ciagRosnacy 1 30 3
[1,4,7,10,13,16,19,22,25,28]
*Main> ciagRosnacy (-1) 1 0.1
[-1.0,-0.9,-0.8,-0.7000000000000001,-0.6000000000000001,-0.5000000000000001,-0.40
787807814457e-16,9.999999999999987e-2,0.19999999999999987,0.2999999999999999,0.39
9998,0.8999999999999998,0.9999999999999998]
*Main> ciagRosnacy (-0.8) 1 0.1
[-0.8,-0.7000000000000001,-0.6000000000000001,-0.5000000000000001,-0.400000000000
57e-16,9.999999999999987e-2,0.19999999999999987,0.2999999999999999,0.399999999999
9999999999998,0.9999999999999998]
*Main> ciagRosnacy -0.8 1 0.1

<interactive>:24:1:
    Could not deduce (Num a0) arising from the ambiguity check for ‘it’
    from the context (Ord t,
                      Num (t -> t -> t -> [t]),
                      Num a,
                      Num t,
                      Fractional (a -> a2 -> t -> t -> t -> [t]),
                      Fractional a2)
```

Różne operacje na listach

```
wycinek _ _ [ ] = [ ]
wycinek pocz kon list
        | pocz > kon   = []
wycinek 0  kon (g : o) = g : wycinek 0 (kon−1) o
wycinek pocz kon (g : o) = wycinek (pocz−1) (kon−1) o
```

```
*Main> :type wycinek
wycinek :: (Ord a, Num a) => a -> a -> [t] -> [t]
```

W definicji funkcji mieszamy swobodnie dozory z dopasowywaniem do wzorca
Znak podkreślenia pełni rolę zmiennej anonimowej – użyty może być wielokrotnie, zawsze oznacza nową daną

wycinek pocz kon lista = **drop** pocz (**take** (kon + 1) lista)

```
*Main> wycinek 1 2 [0,1,2,3]
[1,2]
*Main> wycinek 5 10 [2,4..]
[12,14,16,18,20,22]
*Main> wycinek 5 10 ['a'..]
"fghijk"
*Main> wycinek 10 15 (ciagRosnacy (-20) 20 0.5)
[-15.0,-14.5,-14.0,-13.5,-13.0,-12.5]
*Main>
```

Różne operacje na listach

```
ponumeruj _ [] = []
ponumeruj start (g : o) = ( start , g) : ponumeruj ( start +1) o
```

```
*Main> :type ponumeruj
ponumeruj :: Num t => t -> [t1] -> [(t, t1)]
```

[t1] oznacza typ list składających się z elementów typu t1

```
*Main> ponumeruj 1 [10,12..25]
[(1,10),(2,12),(3,14),(4,16),(5,18),(6,20),(7,22),(8,24)]
*Main> ponumeruj (-2) [10,12..20]
[(-2,10),(-1,12),(0,14),(1,16),(2,18),(3,20)]
*Main> ponumeruj 5 ['a'..'g']
[(5,'a'),(6,'b'),(7,'c'),(8,'d'),(9,'e'),(10,'f'),(11,'g')]
*Main>
```

Funkcja *map*

map :: (a -> b) -> [a] -> [b]

map f [] = []

map f (x:xs) = (f x) : (map f xs)

----

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

```
Prelude> [x^2 | x <-[1..5]]
[1,4,9,16,25]
Prelude> [x^2-2*x+1 | x<-[1..8]]
[0,1,4,9,16,25,36,49]
```

Funkcja *filter*

filter :: (a -> Bool) -> [a] -> [a]

filter _ [] = []

filter p (x:xs) | p x = x : filter p xs

| otherwise = filter p xs

----

filter p xs = [x | x<- xs, p x]

```
Prelude> [x | x <-[(-10)..10], x<0]
[-10,-9,-8,-7,-6,-5,-4,-3,-2,-1]
Prelude> [x | x <-[1..10], even x]
[2,4,6,8,10]
```

Funkcja *zip*

zip :: [a]-> [b] -> [(a,b)]

zip [] _ = []

zip _ [] = []

zip (x:xs) (y:ys) = (x,y): zip xs ys

```
*Main> zip ['a','b','c'] [1,2,3]
[('a',1),('b',2),('c',3)]
*Main> zip [1,2,3] [[],[1],[2],[3]]
[(1,[]),(2,[1]),(3,[2])]
```

Funkcja *zip* - zastosowania

pairs :: [a]->[(a,a)]

pairs xs = zip xs (tail xs)

```
*Main> pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]
*Main> sorted [1,2,3,4]
True
*Main> sorted [1,7,2,9,4]
False
```

sorted :: Ord a =>[a] -> Bool

sorted xs = and [x<=y | (x,y) <- pairs xs]

```
Prelude> [x<=y | (x,y) <- pairs [1,2,3,4,5]]
[True,True,True,True]
```

Funkcja *zip* - zastosowania

position :: Eq a => a -> [a] -> [Int]

position x xs = [i | (y,i) <- zip xs [0..n], x==y]

where n = length xs -1

```
*Main> positions False [True, False, False, True]
[1,2]
*Main> positions 'a' ['f','a','s','l','a']
[1,4]
*Main> positions 'a' ['a','s','l','a']
[0,3]
```

Funkcja *zipWith* (kombinacja zip i map)

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _ _ = []

```
Prelude> zipWith (+) [1,2,3][4,5,6]
[5,7,9]
Prelude> zipWith (*) [1,2,3][4,5,6]
[4,10,18]
Prelude> let g x y = x^2+y^2
Prelude> zipWith g [1,2,3][4,5,6]
[17,29,45]
```

Różne operacje na listach

suma [] = 0
suma (g : o) = g + (suma o)

iloczyn [] = 1
iloczyn (g : o) = g∗(iloczyn o)

polacz [] = []
polacz (g : o) = g ++ (polacz o)

```
*Main> :type suma
suma :: Num a => [a] -> a
*Main> :type iloczyn
iloczyn :: Num a => [a] -> a
*Main> :type polacz
polacz :: [[t]] -> [t]
*Main>
```

[[t]] oznacza typ list składających się z list składających się z typu t

```
*Main> suma [1..10]
55
*Main> suma (ciagRosnacy 1 100 1)
5050
*Main> iloczyn [1..6]
720
*Main> iloczyn (ciagRosnacy 1 10 1)
3628800
*Main> polacz ["Ala","ska"]
"Alaska"
*Main> polacz ["Program","owanie"," fun","kcyjne"]
"Programowanie funkcyjne"
*Main>
```

- Funkcje suma, iloczyn, polacz mają ten sam schemat operowania na liście danych – różnią się tylko wykonywaną operacją oraz elementem początkowym odpowiednim dla danej operacji (neutralnym).
- Można zdefiniować funkcję uniwersalną redukuj, która realizuje ten schemat, a jako parametry przyjmuje daną operację i element neutralny (i oczywiście listę)

redukuj f elNeutralny [ ] = elNeutralny
redukuj f elNeutralny (g : o) = f g (redukuj f elNeutralny o)

```
*Main> :type redukuj
redukuj :: (t -> t1 -> t1) -> t1 -> [t] -> t1
```

```
*Main> redukuj (+) 0 [1,2,3,4,5,6]
21
*Main> redukuj (+) 0 [1..10]
55
*Main> redukuj (*) 1 [1..10]
3628800
*Main> redukuj (*) 2 [1..6]
1440
*Main> redukuj (++) "" ["Ala","ska"]
"Alaska"
*Main> redukuj (++) [] ["Ala","ska"]
"Alaska"
*Main>
```

Zwijanie list (list folding)

foldr

foldr :: (a -> b -> b) -> b -> [a] -> b

foldr f b [] = b (od prawej)

foldr f b (x:xs) = f x (foldr f b xs)

$\oplus$,b,[e0,e1,e2] → (e0 $\oplus$(e1 $\oplus$(e2 $\oplus$b)))

foldl

foldl :: (a -> b -> a) -> a -> [b] -> a

foldl f b [] = b (od lewej)

foldl f b (x:xs) = foldl f (f b x) xs

$\oplus$,b,[e0,e1,e2] → (((b$\oplus$e0)$\oplus$e1)$\oplus$e2)

foldr (right-associative)

Przykład

Stosujemy foldr (+) 0 do listy [3, 8, 12, 5]

i otrzymujemy sumę elementów listy 3 + (8 + (12 + (5 + 0)))

```
Prelude> foldr (+) 0 [3,8,12,5]
28
Prelude> foldr (+) 2 [3,8,12,5]
30
Prelude> foldr (*) 1 [4,8,5]
160
Prelude> foldr (*) 2 [4,8,5]
320
```

foldr (right-associative)

```
Prelude> foldr (-) 1 [4,8,5]
0
```

==> 4 - (foldr (-) 1 [8,5])

==> 4 - (8 - foldr (-) 1 [5])

==> 4 - (8 - (5 - foldr (-) 1 []))

==> 4 - (8 - (5 - 1))

==> 4 - (8 - 4)

==> 4 - 4

==> 0

4-(8-(5-1))=0

foldl (left-associative)

```
Prelude> foldl (-) 1 [4,8,5]
-16
```

==> foldl (-) (1 - 4) [8,5]

==> foldl (-) ((1 - 4) - 8) [5]

==> foldl (-) (((1 - 4) - 8) - 5) []

==> ((1 - 4) - 8) - 5

==> ((-3) - 8) - 5

==> (-11) - 5

==> -16

(((1-4)-8)-5)=-16

Przykłady

- sum xs = foldr (+) 0 xs
- product xs = foldr (*) 1 xs
- xs ++ ys = foldr (:) ys xs
- concat xss = foldr (++) [] xss
- fact n = foldr (*) 1 [1..n]

```
Prelude> foldr (+) 0 [1..5]
15
Prelude> foldr (*) 1 [1..5]
120
Prelude> foldr (:) [1..5] [6..9]
[6,7,8,9,1,2,3,4,5]
Prelude> foldr (++) [] [[1,2],[2,3]]
[1,2,2,3]
Prelude> foldr (*) 1 [1..5]
120
```

foldr1 , foldl1

foldr1 :: (a -> a -> a) -> [a]- > a

foldr1 op [x] = x

foldr1 op (x:xs) = op x (foldr1 op xs)

foldr1 (#) [x_1,x_2,...,x_n] = x_1 # (x_2 # (... (x_{n-1} # x_n)...))

---

foldl1 :: (a -> a -> a) -> [a]- > a

foldr1 op [x] = x

foldl1 op (x:xs) = foldl op x xs

foldl1 (#) [x_1,x_2,...,x_n] = (...((x_1 # x_2) # x_3...# x_{n-1}) # $_{xn}$

foldr1 , foldl1

```
*Main> foldr1 (+) [1,2,3]
6
*Main> foldl1 (+) [1,2,3]
6
*Main> foldr1 (-) [1,2,3]
2
*Main> foldl1 (-) [1,2,3]
-4
```

foldr1 (-) [1,2,3] = 1-(2-3) = 2

foldl1 (-) [1,2,3] = (1-2)-3 = -4

foldr1

foldr1 f xs = foldr f (last xs) (init xs)

```
Prelude> foldr1 (+) [1,2,3,4,5]
15
Prelude> foldr (+) (last [1,2,3,4,5]) (init [1,2,3,4,5])
15
```

```
Prelude> foldr1 (||) [False, True, False]
True
Prelude> foldr1 (&&) [False, True, False]
False
Prelude> foldr1 (min) [5]
5
Prelude> foldr1 (min) [5,4,6,2,0,3]
0
Prelude> foldr1 (min) []
*** Exception: Prelude.foldr1: empty list
```

Funkcje w strukturach danych

double x = 2 * x
square x = x * x
inc x = x + 1

apply [] x = x
apply (f:fs) x = f (apply fs x)

Main> apply [double, square, inc] 3
32

inc 3=4
square 4=16
double 16=32

Rachunek lambda

Przyjmijmy, że mamy pewien przeliczalny nieskończony zbiór *zmiennych przedmiotowych*.

- Zmienne przedmiotowe są lambda-termami (lambda-wyrażeniami), to termy proste,
- Jeśli M i N s ą lambda-termami, to (MN) też jest lambda-termem,
- Jeśli M jest lambda-termem i x jest zmienną, to (λx.M) jest lambda-termem.

Wyrażenia postaci (MN) nazywamy aplikacją,
Wyrażenia postaci (λx.M) to lambda-abstrakcja termu M.

Rachunek lambda

Intuicyjny sens aplikacji (MN) to zastosowanie operacji M do argumentu N.

Przykłady:

(λx.x+1) 1 → 2
(λxy.x+y) 3 → λy.3+y
(λxy.x+y) 3 4 → 7

Abstrakcję (λx.M) interpretujemy jako definicję operacji (funkcji), która argumentowi x przypisuje M. Zmienna x może występować w M, tj. M zależy od x.

Narzuca się analogia z procedurą (funkcją) o parametrze formalnym x i treści M.

Rachunek lambda

Niech f oznacza funkcję zależną od dwóch argumentów x,y.

W matematyce wartość tej funkcji zapisujemy f(x,y),

a w rachunku lambda jako fxy. To znaczy, że f interpretujemy jako jednoargumentową funkcję, która dowolnemu argumentowi x przyporządkowuje jednoargumentową funkcję f_x, taką, że $f_x(y) = f(x,y)$.

Takie reprezentowanie funkcji wieloargumentowych przez jednoargumentowe nazywa się po angielsku „currying” od nazwiska: Haskell B. Curry.

a -> b -> c

Z definicji funkcji można wywnioskować, że ten zapis oznacza funkcję dwuparametrową o typie pierwszego argumentu **a,** drugiego – **b** oraz wyniku **c**.

Biorąc pod uwagę to, że -> jest operatorem, który wiąże prawostronnie, zapis   a -> b -> c

jest równoważny zapisowi:

    a -> (b -> c)

To znaczy, że jest to funkcja biorąca jeden argument typu **a** i zwracająca wartość typu **b -> c**, czyli funkcję.

ale f x y oznacza (f x) y, nie f (x y) !

Wszystkie funkcje w Haskellu są jednoargumentowe

Currying

- Funkcja o dowolnej liczbie argumentów może być przedstawiona jako jednoargumentowa funkcja wyższego rzędu.
- Intuicyjnie, dodawanie ma typ:

  (+) :: Num a => (a, a) -> a

  jednak może być przedstawione także jako:

  (+) :: Num a => a -> a -> a
- Tak rozumiane dodawanie jest jednoargumentową funkcją, która zwraca inną jednoargumentową funkcję

Funkcja *curry*

curry :: ((a, b) -> c) -> (a -> b -> c)

curry g x y = g (x,y)   (curry g = \x y -> f (x, y))

```
Prelude> fst 'a' 1

<interactive>:35:5:
    Couldn't match expected type ‘(a0 -> t, b0)’
                with actual type ‘Char’
    Relevant bindings include it :: t (bound at <interactive>:35:1)
    In the first argument of ‘fst’, namely ‘'a'’
    In the expression: fst 'a' 1
    In an equation for ‘it’: it = fst 'a' 1
Prelude> fst ('a', 1)
'a'
Prelude> curry fst 'a' 1
'a'
```

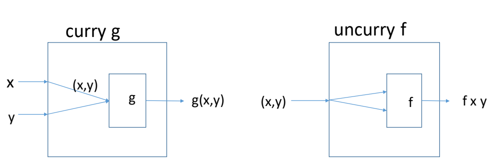
Funkcja *uncurry*

uncurry :: (a -> b -> c) -> ((a, b) -> c)

uncurry f (x,y) = f x y   (uncurry f = \(x, y) -> f x y)

```
Prelude> uncurry (+) (1,2)
3
Prelude> map (uncurry (:)) [('a',"bc"),('d',"ef")]
["abc","def"]
```

Currying i uncurrying



Funkcje anonimowe

Funkcja anonimowa jest funkcją bez nazwy.

W Haskellu: **λ** zastępujemy przez ****
 . zastępujemy przez **->**

Przykłady:

λx.fx   **\x -> f x**

λx.λy.fxy   **\x -> \y -> f x y**

krócej: λxy.fxy   **\x y -> f x y**

Funkcje anonimowe

Wszystkie funkcje są funkcjami jednorgumentowymi!

Przykład. Funkcja

dodaj:: Integer -> Integer -> Integer

dodaj = \x y -> x+y

jest funkcją jednego argumentu typu Integer zwracająca funkcję jednego argumentu typu Integer zwracającą wartość typu Integer

zwieksz1 = dodaj 1

```
*Main> :t dodaj
dodaj :: Integer -> Integer -> Integer
*Main> :t dodaj 5
dodaj 5 :: Integer -> Integer
*Main> :t dodaj 5 6
dodaj 5 6 :: Integer
*Main> zwieksz1 7
8
```

```
Prelude> (\x->x+x)3
6
Prelude> (\xy->x+y)3 4

<interactive>:24:7: Not in scope: ‘x’

<interactive>:24:9: Not in scope: ‘y’
Prelude> (\x y->x+y)3 4
7
Prelude> sum (map(\_ -> 1) "Haskell")
7
```

```
Prelude> (\x -> 2+x)4
6
Prelude> :t (\x -> 2+x)4
(\x -> 2+x)4 :: Num a => a
Prelude> (\f -> 2+f 4) sin
1.2431975046920718
Prelude> :t (\f -> 2+f 4) sin
(\f -> 2+f 4) sin :: Floating a => a
Prelude> map (\x -> 2*x^3-4*x^2+7*x-12) [1,-3,10,-12]
[-7,-123,1658,-4128]
Prelude> :t map (\x -> 2*x^3-4*x^2+7*x-12) [1,-3,10,-12]
map (\x -> 2*x^3-4*x^2+7*x-12) [1,-3,10,-12] :: Num b => [b]
```

```
Prelude> map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
[3,8,9,8,7]
Prelude> map (+3) [1,6,3,2]
[4,9,6,5]
Prelude> map (\x -> x + 3) [1,6,3,2]
[4,9,6,5]
```

Funkcje anonimowe

Czasami wygodniej używać lambda wyrażeń niż funkcji z daną nazwą.

Przykład:

dodj lista = map dj lista
    where dj x=x+1

dodjeden lista = map (\x -> x+1) lista

map f xs = foldr (\x ys -> (f x):ys) [] xs

(.) Operator złożenia funkcji

(.) :: (b -> c) -> (a -> b) -> (a -> c)

f . g = \x -> f (g x)

```
Prelude> reverse "abcde"
"edcba"
Prelude> (reverse.reverse) "abcde"
"abcde"
Prelude> sum [1..10]
55
Prelude> (even.sum) [1..10]
False
```

Operator złożenia funkcji

g x = x * x

f x =
 case x of
  0 -> 1
  1 -> 5
  2 -> 2
  _ -> -1

```
Main> (g . f) 1        g (f 1)
      25
Main> (g . f) 2
      4
Main> (f . g) 1        f (g 1)
      5
Main> (f . g) 2
      -1
```

Funkcja $ (function application)

($) :: (a -> b) -> a -> b

f $ x = f x

right-associative: **f (g (z x))** jest równoważne **f $ g $ z x**

```
Prelude> sum $ map sqrt [1..130]
993.6486803921487
Prelude> sum (map sqrt [1..130])
993.6486803921487
Prelude> sqrt 3 + 4 + 9
14.732050807568877
Prelude> sqrt (3 + 4 + 9)
4.0
Prelude> sqrt $ 3 + 4 + 9
4.0
Prelude> sum (filter (> 10) (map (*2) [2..10]))
80
Prelude> sum $ filter (> 10) $ map (*2) [2..10]
80
```

Funkcja *flip*

flip :: (a -> b -> c) -> (b -> a -> c)

flip f = \x y -> f y x

```
Prelude> flip (-) 1 4
3
Prelude> div 3 4
0
Prelude> flip div 3 4
1
```

flip f = g
  where g x y = f y x

flip f x y = f y x

Literatura

- B.O'Sullivan,J.Goerzen, D.Stewart, Real World Haskell, O'REILLY, 2008.
- K.Doets, J.van Eijck, The Haskell Road to Logic, Math and programming, 2004.
- G.Brzykcy, A.Meissner, Programowanie w Prologu i programowanie funkcyjne, Wyd.PP, 1999.
- Miran Lipovaca, Learn You a Haskell for Great Good!