

## 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,-0.4000000000000001,-0.39999999999999998,0.9999999999999999,0.8999999999999999,0.7999999999999999,0.6999999999999999,0.5999999999999999,0.4999999999999999,0.3999999999999999,0.2999999999999999,0.1999999999999999,0.0999999999999999,0.0,0.0999999999999999,0.1999999999999999,0.2999999999999999,0.3999999999999999,0.4999999999999999,0.5999999999999999,0.6999999999999999,0.7999999999999999,0.8999999999999999,0.9999999999999999,1.0]
*Main> ciagRosnacy (-0.8) 1 0.1
[-0.8,-0.7000000000000001,-0.6000000000000001,-0.5000000000000001,-0.4000000000000001,-0.39999999999999998,0.9999999999999999,0.8999999999999999,0.7999999999999999,0.6999999999999999,0.5999999999999999,0.4999999999999999,0.3999999999999999,0.2999999999999999,0.1999999999999999,0.0999999999999999,0.0,0.0999999999999999,0.1999999999999999,0.2999999999999999,0.3999999999999999,0.4999999999999999,0.5999999999999999,0.6999999999999999,0.7999999999999999,0.8999999999999999,0.9999999999999999,1.0]
*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]),
                      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> |
```

### Różne operacje na listach

```
parami _ [] = []
parami [] _ = []
parami (g1 : o1) (g2 : o2) = (g1, g2) : parami o1 o2
```

```
*Main> :t parami
parami :: [t] -> [t1] -> [(t, t1)]
```

```
*Main> parami "Dama" "Kier"
[(('D','K'),('a','i'),('m','e'),('a','r'))]
*Main> parami [1..5] [21..25]
[(1,21),(2,22),(3,23),(4,24),(5,25)]
*Main> parami [1..5] [21..30]
[(1,21),(2,22),(3,23),(4,24),(5,25)]
*Main> parami [1..50] [21..25]
[(1,21),(2,22),(3,23),(4,24),(5,25)]
*Main> parami [1..5] ['a'..]
[(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e')]
*Main> |
```

### 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 (ciagRoznacy 1 100 1)
5050
*Main> iloczyn [1..6]
720
*Main> iloczyn (ciagRoznacy 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> |
```

### 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
```

#### Przykład.

```
Prelude> foldr (*) 1 [4,8,5]
160
```

### 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
```

### 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
```

### Przykłady

- `sum xs = foldr (+) 0 xs`

- `product xs = foldr (*) 1 xs`

- `xs ++ ys = foldr (:) ys xs`

- `concat xss = foldr (++) [] xss`

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

```
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]
```

### 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=31
```

### Rachunek lambda

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

- **Zmienne** przedmiotowe są lambda-termami (lambda-wyrażeniami), to terminy proste,
- Jeśli  $M$  i  $N$  są lambda-termami, to  $(MN)$  też jest lambda-termem,
- Jeśli  $M$  jest lambda-termem i  $x$  jest zmienną, to  $(\lambda x.M)$  jest lambda-termem.

Wyrażenia postaci  $(MN)$  nazywamy **aplikacją**,  
Wyrażenia postaci  $(\lambda x.M)$  to **lambda-abstrakcja termu  $M$** .

### Rachunek lambda

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

**Przykłady:**

```
(\x.x+1)1 → 2
(\xy.x+y)3 → \y.3+y
(\xy.x+y)3 4 → 7
```

Abstrakcję  $(\lambda x.M)$  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  $fx y$ . 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.

### $a \rightarrow b \rightarrow c$

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  $\rightarrow$  jest operatorem, który wiąże prawostronnie, zapis  $a \rightarrow b \rightarrow c$

jest równoważny zapisowi:

$a \rightarrow (b \rightarrow c)$

To znaczy, że jest to funkcja biorąca jeden argument typu  $a$  i zwracająca wartość typu  $b \rightarrow c$ , czyli funkcję.

**Wszystkie funkcje w Haskellu są jednoargumentowe**

### Funkcje anonimowe

**Funkcja anonimowa** jest funkcją bez nazwy.

W Haskellu:  $\lambda$  zastępujemy przez  $\backslash$   
. $\rightarrow$  zastępujemy przez  $\rightarrow$

**Przykłady:**

$\lambda x.f x$      $\backslash x \rightarrow f x$

$\lambda x.\lambda y.f x y$      $\backslash x \rightarrow \backslash y \rightarrow f x y$   
krócej:  $\lambda x y.f x y$      $\backslash x y \rightarrow f x y$

## Funkcje anonimowe

Wszystkie funkcje są funkcjami jednorgumentowymi!

Przykład. Funkcja

`dodaj :: Integer -> Integer -> Integer`

`dodaj = \x y -> x+y`

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

`zwiększ1 = 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> zwiększ1 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]
```

## 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
```

## (.) 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`

<code>f x =</code>	<code>Main&gt; (g . f) 1</code>	<code>g (f 1)</code>
<code>case x of</code>	<code>25</code>	
<code>0 -&gt; 1</code>	<code>Main&gt; (g . f) 2</code>	
<code>1 -&gt; 5</code>	<code>4</code>	
<code>2 -&gt; 2</code>	<code>Main&gt; (f . g) 1</code>	<code>f (g 1)</code>
<code>_ -&gt; -1</code>	<code>5</code>	
	<code>Main&gt; (f . g) 2</code>	<code>-1</code>

### Funkcja *curry*

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

`curry f = \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`

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

### 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
```

### 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!