# Programowanie funkcyjne

## HASKELL cz.2

## Podstawowe przykłady

```
silnia1  0 = 1
silnia1  n = n ∗ silnia1  (n – 1)
```

-------------------------------------------------------------------

```
silnia2 : : Integer − > Integer
silnia2  0 = 1
silnia2  n = n ∗ silnia2  (n – 1)
```

```
silnia3 n = silniaPOM n 1
        where silniaPOM 0 x = x
              silniaPOM n x = silniaPOM (n-1) (x*n)
```

--------------------------------------------------------

```
silnia4 :: Integer -> Integer
silnia4 n = silniaPOM n 1
        where silniaPOM 0 x = x
              silniaPOM n x = silniaPOM (n-1) (x*n)
```

```
silnia1a n
  | n==0 = 1
  | otherwise = n ∗ silnia1a (n − 1)
```

----------------------------------------------------------------------

```
silnia1b n = if n == 0  then 1
                        else n ∗ silnia1b (n − 1)
```

```
Ok, modules loaded: Main.
*Main> silnia1 10
3628800
*Main> silnia2 10
3628800
*Main> silnia3 10
3628800
*Main> silnia4 10
3628800
*Main> silnia2 2.5

<interactive>:14:9:
    No instance for (Fractional Integer) arising from the literal ‘2.5’
    In the first argument of ‘silnia2’, namely ‘2.5’
    In the expression: silnia2 2.5
    In an equation for ‘it’: it = silnia2 2.5
*Main> silnia4 2.5

<interactive>:15:9:
    No instance for (Fractional Integer) arising from the literal ‘2.5’
    In the first argument of ‘silnia4’, namely ‘2.5’
    In the expression: silnia4 2.5
    In an equation for ‘it’: it = silnia4 2.5
*Main> silnia3 2.5
```

```
*Main> :type silnia1
silnia1 :: (Num a, Eq a) => a -> a
*Main> :type silnia2
silnia2 :: Integer -> Integer
*Main> :type silnia3
silnia3 :: (Num a, Eq a) => a -> a
*Main> :type silnia4
silnia4 :: Integer -> Integer
*Main>
```

**Num** - rodzina typów liczbowych
**Eq** – rodzina (klasa) typów, dla których zdefiniowane jest porównywanie (operatory == i  /=)

Inne przykładowe klasy:
**Show** — klasa typów, których wartości można wypisywać na ekranie,
**Enum** — klasa typów wyliczeniowych (dla których muszą istnieć m.in. operacje brania poprzednika i następnika).

*Main> silnia3 2.5

nigdy się nie skończy - ani poprawnym wynikiem, ani przepełnieniem stosu, bo stosu nie używa (!)

W definicjach funkcji silnia3 i silnia4 użyto *rekurencji ogonowej*, bo funkcja zdefiniowana rekurencyjnie występuje w swojej własnej definicji jako funkcja, a więc jest ostatnim wywołaniem potrzebnym do obliczenia wyniku.

Przy takiej konstrukcji (i przy referencyjnej przezroczystości) nie ma potrzeby budowania stosu.

Rekurencja ogonowa zastępuje efektywnie pętle - podobnie jak one nie zużywa dodatkowej pamięci za każdym „obrotem”.

## Rekurencja a iteracja

**silnia** n = if n==0 then 1
else n***silnia**(n-1)

3!

silnia 3 = 3*silnia 2
silnia 2 = 2*silnia 1
silnia 1 = 1 silnia 0
silnia 0 = 1
silnia 1 = 1*1=1
silnia 2 = 2*1=2
silnia 3 = 3*2=6

## Rekurencja a iteracja

**silnia** n = **silniaPOM** n 1
**silniaPOM** n x = if n== 0 then x
else **silniaPOM** (n-1) (n*x)

3!

silnia 3 = silniaPOM 3 1
silniaPOM 3 1 = silniaPOM 2 3*1
silniaPOM 2 3 = silniaPOM 1 2*3
silniaPOM 1 6 = silniaPOM 0 1*6
silniaPOM 0 6 = 6

## Rekurencja a iteracja

W definicji „iteracyjnej” wprowadzamy dodatkową funkcję, której argumenty odgrywają rolę **akumulatorów**, czyli służą do przechowywania wyników częściowych funkcji.

Dodatkowa funkcja: **silniaPOM**.

Pierwszy z argumentów dodatkowej funkcji silniaPOM odgrywa rolę licznika i zmienia się co do wartości od *n* do *0* (w kolejnych wywołaniach funkcji silniaPOM mamy *n-1*), a w drugim przechowywane są iloczyny częściowe *n*x.*

## Rekurencja a iteracja

W wersjach „akumulatorowych” definicji można poprawić efektywność obliczeń przez wyeliminowanie obliczeń powtarzających się.

**(Definicja rekurencyjna)**

(Obliczanie n-tego wyrazu ciągu Fibonnaciego)

**fib** n = if n==0 then 1
else if n==1 then 1
else **fib**(n-1) + **fib**(n-2)

## Rekurencja a iteracja

**(wersja „akumulatorowa”)**

**fib** n = **fibPOM** n 1 1
**fibPOM** n f1 f2 = if n==1 then f1
else **fibPOM** (n-1) (f1+f2) f1

- Rolę licznika odgrywa pierwszy argument dodatkowej funkcji fibPOM, a jego wartość zmienia się od *n* do *1*.
- W akumulatorach reprezentowanych przez dalsze argumenty funkcji przechowywane są wartości dwóch kolejnych wyrazów ciągu Fibonacciego.

## case of

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

## (.) złożenie funkcji

```
square x = x * x

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

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

**⊕,b,[e0,e1,e2] → (e0 ⊕ (e1 ⊕ (e2 ⊕ b)))**

**foldl**

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f b [] = b
foldl f b (x:xs) = foldl f (f b x) xs

**⊕,b,[e0,e1,e2] → (((b ⊕ e0) ⊕ e1) ⊕ 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
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
inc 3=4
        32
square 4=16

double 16=32
```

## Najmniejszy element listy

```
mnmInt :: [Int] -> Int
mnmInt [] = error "empty list"
mnmInt [x] = x
mnmInt (x:xs) = min x (mnmInt xs)

min :: Int -> Int -> Int
min x y | x <= y  = x
        | otherwise = y
```

## Średnia arytmetyczna elementów listy

```
srednia :: [Int] -> Float
srednia [] = error "lista pusta"
srednia xs = fromInteger (sum xs) / fromInteger (length xs)

    sum ::[Int] -> Int
    sum [] = 0
    sum (x:xs) = x + sum xs

length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

fromInteger - konweruje INTs do Floats

## Sortowanie elementów listy

**removeFst** – usuwa pierwsze wystąpienie liczby *m* w liście
**srtInts** – sortuje elementy listy liczbowej rosnąco

```
srtInts :: [Int] -> [Int]
srtInts [] = []
srtInts xs = m : (srtInts (removeFst m xs)) where m = mnmInt xs

removeFst :: Eq a => a -> [a] -> [a]
removeFst x [] = []
removeFst x (y:ys) | x == y = ys
                   | otherwise = y : (removeFst x ys)
```

## Quicksort

```
quicksort [] =[]
quiksort (x:xs) = quicksort(filter(<x)xs) ++
                  [x]++
                  quicksort(filter(>=x)xs)
```

- Wynikiem sortowania ciągu pustego jest ciąg pusty
- (x:xs) ciąg niepusty składa się z głowy x i ogona xs
- (filter(<x)xs) z ciągu xs wybierz elementy mniejsze od x
- (filter(>=x)xs) z ciągu xs wybierz elementy większe lub równe x
- ++ połącz ciągi
- Kolejność obliczeń nie jest określona

## Map

**Map** zwraca listę zawierającą zastosowania funkcji do wszystkich elementów listy wejściowej

```
>map (^2) [1,2,3,4,5]
 [1,4,9,16,25]
>map (>3) [1..5]
```

[False,False,False,True,True]

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

## Filter

**Filter** – wybiera z listy elementy spełniające pewne warunki

>filter (>3) [1..10]

[4,5,6,7,8,9,10]

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

```
filter p [] = []
filter p (x:xs) =
if p x
then x : filter p xs
else filter p xs
```

## 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!