

## 1. Programowanie deklaratywne

W językach deklaratywnych programista **deklaruje zależności oraz cele** nie podając wprost sposobu osiągnięcia wyniku. Innymi słowy opisujemy co chcemy osiągnąć, nie opisując sposobu.

## 2. Programowanie funkcyjne

- Jest podparadygmatem programowania deklaratywnego. Opisujemy pożądany wynik **przy pomocy funkcji**. Kompilator wyłącznie oblicza wartość funkcji głównej.
- **Funkcje czyste** zawsze przyjmują tę samą wartość dla tych samych argumentów - nie są funkcjami w rozumieniu programowania funkcyjnego.
- W programowaniu funkcyjnym podstawowym narzędziem jest **rekurencja**, a także **rachunek lambda** (definiowanie funkcji bez nadawanie jej nazwy) czy **przezroczystość referencyjna** (funkcja, która daje wynik, może być parametrem  $f(a) = b$ ,  $g(b,c) = g(f(a),c) = d$ ).
- **Leniwa ewaluacja**, czyli obliczanie wartości wyrażenia dopiero gdy jest ono potrzebne (umożliwia to stworzenie nieskończonych matematycznie struktur)
- **Silne typowanie i matematyczna ścisłość** (łatwość w dowodzeniu poprawności)

## 3. Operatory

Zapis:  $\text{mod } 4 \ 3 = 4 \ \text{'mod'} \ 3 = 1$  to samo dla `div`

Zapis:  $(+) \ 5 \ 2 = 5 + 2 = 7$  oraz  $(-, *, /, ==, <, <=, >=, /=, ^, **, \text{mod}, \text{div}, ||, \&\&, \text{not})$

Potęga całkowita:  $2.5^3$  potęga ułamkowa:  $2.5^{**3.5}$

Dwuargumentowe:  $x \ \& \ y = (x + y) / 2$

## 4. Typy podstawowe

**Int** - z zakresu  $-2^{29}..29^{28}$

**Integer** - dowolna liczba całkowita

**Float** - zmiennoprzecinkowa pojedynczej precyzji

**Double** - podwójnej precyzji

**Char** - pojedynczy znak 'x'

**String [Char]** - łańcuch znaków "xxx"

**Bool** - typ wyliczeniowy logiczny

**Ordering** - Typ relacji między elementami posiadający trzy wartości (LT, EQ, GT) - `compare`

**Num** - rodzina typów liczbowych     **Eq** - rodzina typów dla których zdefiniowane jest `==` i `/=`

## 5. Typy strukturalne

**Lista** - elementy tego samego typu (rozmiar nieokreślony)

`[Int] = [1, 2, 3]` `[Char] = ['a', 'b', 'c']` `[[Int]] = [[1], [1,4], []]` `[(String, Bool)] = [("Ala" True)]`

**Krotka** - elementy różnych typów (rozmiar określony - stały)

`(Int, Char) = (1, 'a')` `((Bool, String), Int) = ((True, "Ala"), 2)` `[(Int], Char) = ([1, 2], 'c')`

## 6. Typy funkcji

nazwa funkcji :: `typArg1 -> typArg2 -> typArgN -> TypWartościZwracanej`

`inc :: Int -> Int`

## 7. Typy polimorficzne (rodzina typów)

`kw :: Num a => a -> a`

(gdzie `a` to zmienna typowa, ograniczona przez "`Num a =>`" co oznacza klasę typów liczbowych i kilku innych, `::` znaczy "jest typu")

## 8. Przykłady typów i inne wyniki operacji

Operator (typy jego argumentów) `:t (||) :: Bool -> Bool -> Bool`

Wynik funkcji `:t True && False :: Bool`

Krotka (typy jej składowych) `:t (True, "pf") :: (Bool, [Char])`

Funkcji (typy wejściowe i wyjściowe) `:t fst :: (a, b) -> a`

Liczby całkowitej `:t 6 :: Num a => a`

Liczby zmiennoprzecinkowej `:t pi :: Floating a => a`

Znaku `:t 'a' :: Char`

Stringu `:t "zdanie" :: [Char]`

`:t (+) :: Num a => a -> a -> a` typ operatora/funkcji

`:t (+) 2 3 :: Num a => a` typ wyniku operatora/funkcji

`map (^2) [1,2,7,5]` będzie `[1, 4, 49, 25]`

`map (>2) [1, 2, 7, 5]` będzie `[False, False, True, True]`

`filter (>2) [1, 2, 7, 5]` będzie `[7, 5]`

`:t map` będzie `(a -> b) -> [a] -> [b]` czyli `(a -> b)` funkcja zmieniająca `a` na `b`, `[a]` jakaś tablica, `[b]` czyli wynik tej funkcji z tą tablicą

`:t filter` będzie `(a -> Bool) -> [a] -> [a]` czyli `(a -> Bool)` funkcja sprawdzająca dla elementu warunek i dająca `Bool`, `[a]` - dana tablica, `[a]` - wynik tej funkcji na tablicy

Konkatenacja do listy 2:lista (na początek listy) - działa tylko gdy konkatenujemy do listy

Konkatenacja `[Char]` "`ALA`"++"`OLA`"

## 9. Sposoby definiowania funkcji

Przykłady	
<code>silnia1a n = if n == 0 then 1</code> <code>                  else n * silnia1a (n - 1)</code>	
-----	
<code>silnia1b n</code>   <code>n==0 = 1</code>   <code>otherwise = n * silnia1b (n - 1)</code>	<code>0!=1</code> <code>n!=n*(n-1)!</code>
-----	
<code>silnia1c 0 = 1</code> <code>silnia1c n = n * silnia1c (n - 1)</code>	
-----	
<code>silnia1d n = case n of 0 -&gt; 1</code> <code>                      _ -&gt; n * silnia1d (n - 1)</code>	

## 10. Wersja akumulatorowa (rekurencja ogonowa)

Nie ma potrzeby budowania stosu, zastępuje pętlę, jest wydajniejsza, akumulatory przechowują wyniki częściowe funkcji głównej, pierwszy argument pełni rolę licznika

```

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

```

## 1. Definiowanie listy

**[1..5]** = [1, 2, 3, 4, 5]      **[1.0, 1.25.. 2.0]** = [1.0, 1.25, 1.5, 1.75, 2.0]    **[1..]** = [1, 2, 3, ...]

**[x^2 | x<- [1 .. 5]]** =  $x^2 : x \in \{1, \dots, 5\} = \{1, 4, 9, 16, 25\}$

**factors n** = [x | x <- [1 .. n], mod n x == 0]    **factors 20** = [1, 2, 4, 5, 10, 20]

## 2. Operacje na listach

**:: [a] -> [a]**

**head (x:\_)** = x      pierwszy element (x - pierwszy, \_ - reszta elementów)

**tail (\_:xs)** = xs      ostatni element (\_ - jakies elementy, xs - ostatni element)

**init [x]** = []      **init (x:xs)** = x : init xs      wszystkie elementy bez ostatniego

**last [x]** = x      **last (\_:xs)** = last xs      ostatni element

**length** lista - dlugosc listy    **take n** lista - wez n elementow listy    **sum** lista - suma listy

**drop x** lista - wyrzuc element x z listy      **elem x** lista - sprawdz czy x jest w liscie

**reverse** lista - odwroc liste      **min a b** - minimum z a i b

## 3. Funkcje wyzszego rzędu

**quicksort []** = []

**quicksort (x:xs)** = quicksort(**filter**(<x) xs) ++ filter wybierze elementy < od x

[x]++

quicksort(**filter**(>=x) xs)

filter wybierze elementy >= od x

**map f []** = []

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

**filter \_ []** = []

**filter p (x:xs)** = if (p x == True) then x: filter p xs

else filter p xs

## 1. Foldr i Foldl

### 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
```

$\text{myFoldl } f \ x \ [] = x$

$\text{myFoldl } f \ x \ (h:t) = \text{myFoldl } f \ (f \ x \ h) \ t$

### 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
```

$\text{myFoldr } f \ x \ [] = x$

$\text{myFoldr } f \ x \ (h:t) = f \ h \ (\text{myFoldr } f \ x \ t)$

## 2. Rachunek Lambda

### 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  $(\lambda x.M)$  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.

## 3. Funkcje anonimowe

## Funkcje anonimowe

**Funkcja anonimowa** jest funkcją bez nazwy.

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

**Przykłady:**

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

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

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

Np.:

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

`(\x y -> x*y) 2 5 = 10`

## 4. Operator złożenia funkcji

`reverse.reverse "abcde" = "abcde"`

### 1. Własny typ

`data TypKonstruktora = WartoscKonstruktora1 | WartoscKonstruktora2 | ...`

### 2. Konstruktory bezargumentowe

- `data Bool = True | False`
- `data Marka = Opel | Ford | Fiat`
- `data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun`

`przetlumacz :: Day -> String`

`przetlumacz d = case d of`

`Mon -> "Pon"`

`Tue -> "Wt"`

`.....`

### 3. Definicja własnego typu

`data NazwaTypu = NazwaKonstruktora Arg1 Arg2 Arg3 ...`

**`data BookInfo = Book Int String [String]`**

Tworzymy instancje typu BookInfo (Konstruktor Indeks Tytuł Autorzy)

`let book1 = Book 12345 "Harry Potter" ["J.K.R", "J.R.R.T"]`

`:type book1 :: BookInfo`

### 4. Synonimy typów

`type NazwaTypu = TypJuzIstniejacy`

`type Nazwisko = String`

`type NazwaTypu = (Typ1, Typ2, ...)`

`type But = (Char, Int)`

```

type Punkt = (Double, Double)
odleglosc :: Punkt -> Punkt -> Double
odleglosc (x1, y1) (x2, y2) = sqrt ( (x1-x2)^2 + (y1-y2)^2 )

```

## 5. Alternatywne konstruktory

**Alternatywne konstruktory**

```

data PointType = Point Float Float

data Shape = Rectangle PointType PointType |
            Circle PointType Float |
            Triangle PointType PointType PointType

r = Rectangle (Point 2 4) (Point 8.5 2)
c = Circle (Point 1 1.5) 5.5
t = Triangle (Point 0 0) (Point 4.5 6) (Point 9 0)

```

```

*Main> :t r
r :: Shape
*Main> :t c
c :: Shape
*Main> :t t
t :: Shape
*Main> |

```

## 6. Typy parametryzowane

```

data NazwaTypu TypParametru = NazwaKonstruktora TypParametru1 TypParametru2 ...
data PairType a = Pair a a
p = Pair 2 5
:type p :: PairType Integer

```

## 7. Typy rekurencyjne

```

data Naturalna = Zero | Succ Naturalna
n0 = Zero          n1 = Succ Zero          n2 = Succ (Succ Zero)
natToInt Zero = 0
natToInt (Succ n) = 1 + natToInt n

```

## 8. Typy rekurencyjne - listy

```

data NazwaTypu TypParametru = Konstruktor1 | Konstruktor2 Rekurencja
data List a = Empty | Cons a (List a)
l :: List Int
l = Cons 12 (Cons 8 (Cons 10 Empty))
len :: List a -> Int
len Empty = 0
len (Cons _ xs) = 1 + len xs

```

## 9. Typy rekurencyjne - drzewa

**Typy rekurencyjne – drzewa**  
Drzewo jest puste albo składa się z wartości i dwóch poddrzew

```

data Tree a = Empty | Node a (Tree a) (Tree a)
t :: Tree Int
t = Node 5 (Node 3 (Node 8 Empty Empty)
               (Node 1 Empty Empty))
      (Node 4 Empty
               (Node 6 Empty Empty))

```

```

      5
     / \
    3   4
   / \   \
  8  1   6

```

depth Empty = 0

depth (Node a l r) = 1 + max (depth l) (depth r)

preorder Empty = []

preorder (Node a l r) = [a] ++ preorder l ++ preorder r      //[5, 3, 8, 1, 4, 6]

inorder Empty = []

inorder (Node a l r) = inorder l ++ [a] ++ inorder r      //[8, 3, 1, 5, 4, 6]

postorder Empty = []

postorder (Node a l r) = postorder l ++ postorder r ++ [a]      //[8, 1, 3, 6, 4, 5]

## 1. Moduły

module Nazwa (nazwy funkcji, nazwy typow) where

definicje klas typów, funkcji itp

Gdy (..) jest puste to wówczas wszystkie typy, klasy, funkcje są dostępne

Z modułów korzystamy za pomocą: import Nazwa

module M ( A(..), B(Kb1, Kb3), f ) where ...

udostępnia typ danych ze wszystkimi jego konstruktorami, typ B z konstruktorami Kb1 i Kb3 oraz funkcje f

## 2. Klasy typów

## Klasy typów

Definicja klasy

```
class Nazwa-klasy zmienne-typowe where
  nazwa-funkcji/operatora :: typ-funkcji/operatora
  definicja-niektórych-funkcji/operatorów
```

Aby typ danych stał się egzemplarzem klasy, należy użyć konstrukcji:

```
instance Nazwa-klasy Nazwa-typu where
  przeciążenie-wymaganych-funkcji/operatorów
lub
data definicja-typu deriving (lista-klas)
```

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
```

```
class Eq a => Ord a where           //dziedziczenie
    max, min :: a -> a -> a
    .....
```

### 3. Klasa Show, Read, Enum, Monady

#### 4. Unie

```
data Unia typ1 typ2 = Jedno typ1 | Drugie typ2
    deriving(Eq, Show)
```

#### 5. Operacje wejścia/wyjścia

```
getChar
putChar c
getLine
putStr
putStrLn
return
firstLetter (x:xs) = return x
inicjaly = do
    putStr "Podaj imie: "
    a <- getLine
    putStr "Podaj nazwisko: "
    b <- getLine
    putStr ("INICJALY: " ++ firstLetter a ++ "." ++ firstLetter b ++ ".")
```

#### 6. Parser (przyjmuje napis i zwraca wartosc, wartość i nieskonsumowaną część napisu)

item - konsumuje pierwszy znak	item "anna" - [('a',"nna")]
failure - zawsze kończy się niepowodzeniem	failure "anna" - []



return v - zwraca v bez konsumowania wejścia      return 1 "as" - [(1, "as")]  
p +++ q - zachowuje sie jak parser p w przypadku powodzenia, q w przypadku porażki