

# Programowanie w języku Haskell

## Strona domowa

<http://www.haskell.org>

## Popularne implementacje

- Hugs, GHC (The Glasgow Haskell Compiler)

## Źródła wykładu

- 1 Graham Hutton: *Programming in Haskell*, Cambridge University Press 2007  
<http://www.cs.nott.ac.uk/~gmh/book.html> (slajdy)
- 2 Fethi Rabhi, Guy Lapalme: *Algorithms: A Functional Programming Approach*, Addison-Wesley 1999
- 3 Hal Daume: *Yet Another Haskell Tutorial*  
<http://www.cs.utah.edu/~hal/docs/daume02yaht.pdf>
- 4 Paul Hudak et al.: *A Gentle Introduction to Haskell 98*  
<http://www.haskell.org/tutorial/>

## Przykładowy plik źródłowy (prog.hs)

```
x = 5
increment n = n + 1
main = putStrLn "Hello World"
```

## Kompilacja

```
C:\> ghc prog.hs -o prog.exe
C:\> prog.exe
Hello World
```

## Uruchomienie w trybie interaktywnym

```
C:\> ghci
ghci> :load prog.hs
ghci> x
5
ghci> x + 2
7
```

```
ghci> increment 1
2
ghci> main
Hello World
ghci> let y = 2
ghci> let decrement n = n - 1
ghci> decrement y
1
ghci> :reload
```

## Biblioteka standardowa (Prelude.hs)

```
ghci> (2 * 3)^2
36
ghci> sqrt 4
2.0
ghci> 'a' == 'b'
False
ghci> pi
3.141592653589793
```

# Funkcje

## Definiowanie wartości

```
pi = 3.141592653589793
```

## Definiowanie funkcji

```
square x = x ^ 2
```

```
triangleArea a h = 0.5 * a * h
```

```
ghci> square 2
```

```
4.0
```

```
ghci> triangleArea 4 2
```

```
4.0
```

## Uwaga

Nazwy wartości i funkcji muszą zaczynać się małą literą. Dalej mogą wystąpić litery, cyfry, znaki podkreślenia (`_`) lub apostrofu (`'`)

## Aplikacja funkcji

Matematyka	Haskell
$f(x)$	<code>f x</code>
$f(x, y)$	<code>f x y</code>
$f(g(x))$	<code>f (g x)</code>
$f(x, g(y))$	<code>f x (g y)</code>
$f(x) * g(y)$	<code>f x * g y</code>

## Uwaga

Aplikacja funkcji ma wyższy priorytet niż jakikolwiek inny operator

```
ghci> triangleArea 4 2 * square 2 + 1  
17.0
```

## Klauzula *where*

Definiuje wartości i funkcje użyte wewnątrz wyrażenia:

- 1 `volume r = a * pi * cube`  
    `where a = 4 / 3`  
        `cube = r ^ 3`
- 2 `volume r = a * pi * cube r`  
    `where a = 4 / 3`  
        `cube x = x ^ 3`

## Wyrażenie *let ... in*

```
volume r = let a = 4 / 3  
           cube = r ^ 3  
           in a * pi * cube
```

## Lokalność definicji po *where* (*let*)

```
a = 1
volume r = ...
    where a = 4 / 3
fun x = a * x
```

```
ghci> fun 5
5
```

## *where* i *let* razem

Definicje po *let* przesłaniają te po *where*:

```
fun x = let y = x + 1
        in y
        where y = x + 2
```

```
ghci> fun 5
6
```

## Wyrażenie warunkowe *if ... then ... else*

```
sgn x = if x < 0 then -1
        else if x == 0 then 0
        else 1
```

## Uwaga

Nie ma konstrukcji *if ... then*

## Strażnicy (*guards*)

```
sgn x | x < 0    = -1
      | x == 0   = 0
      | x > 0    = 1
```

```
sgn x | x < 0      = -1
      | x == 0     = 0
      | otherwise = 1
```

$$\operatorname{sgn}(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & x > 0 \end{cases}$$
$$\operatorname{sgn}(x) = \begin{cases} -1, & x < 0 \\ 0, & x = 0 \\ 1, & \text{wpp} \end{cases}$$



Kolejność strażników ma znaczenie:

```
f a b | a >= b    = "wieksze lub rowne"  
      | a == b    = "rowne"  
      | otherwise = "niewieksze"  
      | a < b     = "mniejsze"
```

```
ghci> f 1 1  
"wieksze lub rowne"  
ghci> f 1 5  
"niewieksze"
```

## Uwaga

`otherwise = True`

## Dopasowanie wzorca (*pattern matching*)

❶ `conj a b = if a == True && b == True  
              then True else False`

❷ `conj True True = True  
   conj a      b   = False`

❸ `conj True True = True  
   conj _      _   = False`

Kolejność wzorców ma znaczenie:

```
conj _      _      = False  
conj True True = True
```

```
ghci> conj True True  
False
```

## Wyrażenie *case ... of*

- 1 `fun 1 = 2`  
`fun 2 = 3`  
`fun _ = -1`
- 2 `fun n = case n of`  
`1 -> 2`  
`2 -> 3`  
`_ -> -1`

Argumentem *case* może być dowolne wyrażenie:

```
fun n = case n < 0 of
  True  -> "ujemna"
  False -> "dodatnia"
```

## Przykład

$$n! = \begin{cases} 1, & n = 0 \\ n * (n - 1)!, & n \neq 0 \end{cases}$$

- 1 `fact n = if n == 0 then 1  
          else n * fact (n - 1)`
- 2 `fact n | n == 0      = 1  
      | otherwise = n * fact (n - 1)`
- 3 `fact 0 = 1  
fact n = n * fact (n - 1)`
- 4 `fact n = case n of 0 -> 1  
                  _ -> n * fact (n - 1)`

## Definiowanie operatorów dwuargumentowych

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

$x \#^{\wedge} y = x == y - 1$

```
ghci> 1 & 2
```

```
1.5
```

```
ghci> 1 #^ 2
```

```
True
```

## Uwagi

- Operatory dwuargumentowe są funkcjami „zapisywanymi” pomiędzy argumentami
- Nazwy operatorów składają się z jednego lub więcej symboli

## Konwersja operator–funkcja

Nazwa operatora dwuargumentowego może być zapisana przed argumentami jeśli zostanie ujęta w nawiasy ( )

```
sum a b = a + b  
sum' a b = (+) a b
```

Nazwa funkcji dwuargumentowej może być zapisana pomiędzy argumentami jeśli zostanie ujęta w odwrotne apostrofy ' '

```
a = triangleArea 4 2  
b = 4 'triangleArea' 2
```

# Formatowanie kodu

## Ogólne zasady

- 1 Definicje najwyższego poziomu zaczynają się w tej samej kolumnie:

```
abs x = if x < 0 then -x else x
a = 5
```

- 2 Definicja może być „złamana” w dowolnym miejscu pod warunkiem, że wcięcia będą większe niż w pierwszej linii:

```
abs
  x = if
    x < 0
    then -x else x
a =
  5
```

- 3 Jeżeli po *where* lub *let* występuje więcej niż jedna definicja lokalna, wszystkie muszą zaczynać się w tej samej kolumnie:

<code>f x = a * g x</code>	<code>f x = a * g x</code>
<code>  where a = 2</code>	<code>  where a = 2</code>
<code>      g x = x ^ 2</code>	<code>      g x = x ^ 2</code>

- 4 Wyrażenia po *of* oraz *do* muszą zaczynać się w tej samej kolumnie:

<code>f x = case x of</code>	<code>f x = case x of</code>
<code>  1 -&gt; 2</code>	<code>  1 -&gt; 2</code>
<code>  2 -&gt; 3</code>	<code>  2 -&gt; 3</code>

## Uwagi

- Rozmiar znaku tabulatora: 8
- Można *explicite* używać nawiasów i średników:

```
f x = case x of { 1 -> 2 ; 2 -> 3 }
```



## Komentarze

```
{- To jest komentarz blokowy, który może rozciągać  
się na {- A to jest komentarz zagnieżdżony -}  
kilka linii. -}
```

```
-- ten komentarz rozciąga się do końca linii  
main = putStrLn "Hello World" -- wyświetl tekst
```

## Programowanie piśmienne (*literate programming*)

- Tekst w pliku z rozszerzeniem `*.lhs` jest domyślnie komentarzem
- Linie kodu zaczynają się od `>` lub umieszczone są pomiędzy `\begin{code}` i `\end{code}`

## Plik prog.lhs

Ten program pokazuje na ekranie "Hello World"

```
> main = putStrLn "Hello World"
```

i nie robi nic poza tym.

## Plik prog.lhs (styl L<sup>A</sup>T<sub>E</sub>X-owy)

Ten program pokazuje na ekranie "Hello World"

```
\begin{code}
```

```
main = putStrLn "Hello World"
```

```
\end{code}
```

i nie robi nic poza tym.

# Typy

## Podstawowe typy danych

Int (skończonej precyzji)	56
Integer (nieskończonej precyzji)	732145682358
Float/Double	3.14159265
Bool	False
Char	'a'
String	"Ala"

## Typy złożone

- Krotka (*tuple*):

(Int, Char)	(1, 'a')
(Int, Char, Float)	(1, 'a', 3.4)
((Bool, String), Int)	((True, "Ala"), 2)
([Int], Char)	([-1, 4, 2], 'c')

- Lista:

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

- Funkcja:

```
Int -> Char
[Int] -> (Char, Float) -> Bool
arg1 -> arg2 -> ... -> argn -> wynik
```

## Uwagi

- Krotka ma określony rozmiar, ale może zawierać elementy różnego typu
- Lista ma nieokreślony rozmiar, ale jej elementy muszą być tego samego typu

## Standardowe operatory

<code>a == b</code> , <code>a != b</code>	równe, nierówne
<code>a &lt; b</code> , <code>a &gt; b</code>	mniejsze, większe
<code>a &lt;= b</code> , <code>a &gt;= b</code>	nie większe, nie mniejsze
<code>a &amp;&amp; b</code> , <code>a    b</code> , <code>not a</code>	koniunkcja, alternatywa, negacja
<code>a + b</code> , <code>a - b</code>	suma, różnica
<code>a * b</code> , <code>a ^ b</code>	mnożenie, potęgowanie
<code>a / b</code> , <code>a 'mod' b</code> , <code>a 'div' b</code>	dzielenie, reszta, część całkowita

## Uwagi

- Oba argumenty muszą być tego samego typu (nie dotyczy `^`)
- Napisy, listy i krotki są uporządkowane *leksykograficznie*

## Operator specyfikowania typu

```
x = 4
```

```
x :: Int
```

```
y = 5 :: Float
```

```
isB :: Char -> Bool
```

```
isB c = (c == 'B') || (c == 'b')
```

## Uwaga

Specyfikowanie typu jest zazwyczaj opcjonalne. Haskell sam wywnioskuje typ danego identyfikatora:

```
ghci> let isB c = (c == 'B') || (c == 'b')
```

```
ghci> :type isB
```

```
isB :: Char -> Bool
```

## Przykład

```
1 roots :: (Float, Float, Float) -> (Float, Float)
  roots (a, b, c) =
    if d < 0 then error "pierwiastki urojone"
    else (r1, r2)
      where r1 = e + sqrt d / (2 * a)
            r2 = e - sqrt d / (2 * a)
            d  = b * b - 4 * a * c
            e  = -b / (2 * a)
```

```
ghci> roots (-2, 3, -1)
(0.5,1.0)
```

```
2 roots :: Float -> Float -> Float -> (Float, Float)
  roots a b c = j.w.
```

```
ghci> roots (-2) 3 (-1)
(0.5,1.0)
```

## Typy polimorficzne

```
ghci> let first (x, y) = x
ghci> :type first
first :: (t, t1) -> t
```

- *fst*

```
fst :: (a, b) -> a
fst (x, y) = x
```

- *snd*

```
snd :: (a, b) -> b
snd (x, y) = y
```

```
ghci> let first3 (x, _, _) = x
ghci> :type first3
first3 :: (t, t1, t2) -> t
```



# Listy

## Definiowanie ciągów arytmetycznych

```
ghci> [1 .. 10]  
[1,2,3,4,5,6,7,8,9,10]
```

```
ghci> [1, 3 .. 10]  
[1,3,5,7,9]
```

```
ghci> [1, 4 ..]  
[1,4,7,10,13,16,19,22,25,28,31,34,37,40,43,46,49,52,55,  
58,61,64,67,70,73,76,79,82,85,88,91,94,97,100,103,106,...]
```

```
ghci> ['a' .. 'z']  
"abcdefghijklmnopqrstuvwxyz"
```

## Uwaga

Typ *String* to lista wartości typu *Char*

## *List comprehensions*

$$\{x^2 : x \in \{1, \dots, 5\}\} = \{1, 4, 9, 16, 25\}$$

```
ghci> [x ^ 2 | x <- [1 .. 5]]  
[1,4,9,16,25]
```

```
ghci> [(x, y) | x <- [1, 2, 3], y <- [4, 5]]  
[(1,4), (1,5), (2,4), (2,5), (3,4), (3,5)]
```

```
ghci> [(x, y) | y <- [4, 5], x <- [1, 2, 3]]  
[(1,4), (2,4), (3,4), (1,5), (2,5), (3,5)]
```

```
ghci> [(x, y) | x <- [1 .. 3], y <- [x .. 3]]  
[(1,1), (1,2), (1,3), (2,2), (2,3), (3,3)]
```

## Przykład

```
firsts :: [(a, b)] -> [a]
firsts ps = [x | (x, _) <- ps]
```

```
ghci> firsts [(1,2), (3,4), (5,6)]
[1,3,5]
```

## Strażnicy w *list comprehensions*

```
ghci> [x | x <- [1 .. 10], even x]
[2,4,6,8,10]
```

```
factors :: Int -> [Int]
factors n = [x | x <- [1 .. n], mod n x == 0]
```

```
ghci> factors 15
[1,3,5,15]
```

## Konstruktor list

Operator `(:)` konstruuje listę z głowy (*head*) i ogona (*tail*)

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

```
ghci> 3 : [4, 5]  
[3,4,5]
```

```
ghci> True : []  
[True]
```

```
ghci> "ab" : ["cd", "efg"]  
["ab", "cd", "efg"]
```

```
ghci> 1 : 2 : 3 : []  
[1,2,3]
```

## Listy a dopasowanie wzorca

- 1 

```
test :: String -> Bool
test ['a',_,_] = True
test _         = False
```

```
ghci> test "abc"
```

```
True
```

```
ghci> test "abcd"
```

```
False
```

- 2 

```
test ('a':_) = True
test _       = False
```

```
ghci> test ['a'..'g']
```

```
True
```

- 3 

```
test (x:_) = x == 'a'
test []    = False
```

## Standardowe operacje na listach

- *length*

`length [] = 0`

`length (x:xs) = 1 + length xs`

`ghci> length [4, 3, 7]`

`3`

`length [4, 3, 7] =>`

`length (4 : [3, 7]) => 1 + length [3, 7]`

`1 + length (3 : [7]) => 1 + 1 + length [7]`

`1 + 1 + length (7 : []) => 1 + 1 + 1 + length []`

`=> 1 + 1 + 1 + 0`

`=> 3`

- operator indeksowania

```
(!!) :: [a] -> Int -> a  
(x:_) !! 0 = x  
(_:xs) !! n = xs !! (n - 1)
```

```
ghci> "abcde" !! 2  
'c'
```

- operator konkatencji

```
(++) :: [a] -> [a] -> [a]  
[] ++ ys = ys  
(x:xs) ++ ys = x : (xs ++ ys)
```

```
ghci> "abc" ++ "de"  
"abcde"
```

- *head*

```
head :: [a] -> a
```

```
head (x:_) = x
```

```
ghci> head [1, 2, 3]
```

```
1
```

- *tail*

```
tail :: [a] -> [a]
```

```
tail (_:xs) = xs
```

```
ghci> tail [1, 2, 3]
```

```
[2,3]
```

## Uwaga

```
ghci> let head (x:_) = x
```

```
ghci> head []
```

```
*** Exception: Non-exhaustive patterns in function head
```



- *init*

```
init :: [a] -> [a]
init [x]      = []
init (x:xs) = x : init xs
```

```
ghci> init [1, 2, 3]
[1,2]
```

- *last*

```
last :: [a] -> a
last [x]      = x
last (_:xs) = last xs
```

```
ghci> last [1, 2, 3]
3
```

- *take*

```
take :: Int -> [a] -> [a]
take 0 _      = []
take _ []     = []
take n (x:xs) = x : take (n - 1) xs
```

```
ghci> take 2 "abcde"
"ab"
```

- *drop*

```
drop :: Int -> [a] -> [a]
drop 0 xs      = xs
drop _ []     = []
drop n (_:xs) = drop (n - 1) xs
```

```
ghci> drop 2 "abcde"
"cde"
```

- *elem*

```
elem :: Eq a => a -> [a] -> Bool
elem x []                = False
elem x (y:ys) | x == y   = True
                | otherwise = elem x ys
```

```
ghci> elem 1 [1, 2, 3]
True
ghci> elem 0 [1, 2, 3]
False
```

- *sum*

```
sum :: Num a => [a] -> a
sum []                = 0
sum (x:xs)            = x + sum xs
```

```
ghci> sum [1, 2, 3]
6
```

- *reverse*

```
reverse :: [a] -> [a]
reverse []      = []
reverse (x:xs) = (reverse xs) ++ [x]
```

```
ghci> reverse "abcde"
"edcba"
```

- *maximum*

```
maximum :: Ord a => [a] -> a
maximum [x]                = x
maximum (x:y:ys) | x > y    = maximum (x:ys)
                  | otherwise = maximum (y:ys)
```

```
ghci> maximum [1, 2, 3]
3
```

## Ćwiczenie

Napisz funkcję konkatenującą listy na podanej liście:

```
ghci> concat ["ab", [], "c"]  
"abc"
```

```
concat :: [[a]] -> [a]  
...
```

- *zip*

```
zip :: [a] -> [b] -> [(a, b)]
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip _        _    = []
```

```
ghci> zip [1, 2] "abc"
[(1, 'a'), (2, 'b')]
```

- *unzip*

```
unzip :: [(a, b)] -> ([a], [b])
unzip []           = ([], [])
unzip (x, y):ps    = (x:xs, y:ys)
                    where (xs, ys) = unzip ps
```

```
ghci> unzip [(1, 'a'), (2, 'b')]
([1, 2], "ab")
```

## Przykład

```
perms [] = [[]]
perms xs = [x:p | x <- xs,
                p <- perms (removeFirst x xs)]
  where
    removeFirst x [] = []
    removeFirst x (y:ys)
      | x == y      = ys
      | otherwise   = y : removeFirst x ys
```

```
ghci> perms "abc"
["abc", "acb", "bac", "bca", "cab", "cba"]
```

# Funkcje wyższego rzędu

Funkcja wyższego rzędu (*higher-order*) przyjmuje jako argumenty lub zwraca w wyniku inne funkcje

## Funkcje *map* i *zipWith*

- *map*

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

```
map f [] = []
```

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

```
ghci> map sqrt [1, 4, 9]
[1.0,2.0,3.0]
```

```
ghci> map fst [('a', 1), ('b', 2)]
"ab"
```

```
ghci> map reverse ["abc", "def", "ghi"]
["cba","fed","ihg"]
```



- *zipWith*

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

```
ghci> zipWith (>) "aba" "baba"
[False,True,False]
```

## Przykład

Damir Medak, Gerhard Navratil: *Haskell-Tutorial*

$$[(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$$

$$2F = \left| \sum_{i=1}^n (x_i - x_{i+1})(y_i + y_{i+1}) \right|, \quad x_{n+1} = x_1, y_{n+1} = y_1$$

```
poly  (x1, y1), (x2, y2), (x3, y3), (x4, y4)
```

```
list1  x1, x2, x3, x4  map fst poly
```

```
list2  x2, x3, x4, x1  tail list1 ++ [head list1]
```

```
list3  y1, y2, y3, y4  map snd poly
```

```
list4  y2, y3, y4, y1  tail list3 ++ [head list3]
```

```
parts = zipWith (*)  
      (zipWith (-) list1 list2)  
      (zipWith (+) list3 list4)
```

```
area = abs ((sum parts) / 2)
```

```
area :: [(Float, Float)] -> Float
```

```
area []      = error "to nie jest wielokat"
```

```
area [x]     = error "punkt nie ma pola"
```

```
area [x,y]   = error "linia nie ma pola"
```

```
area poly    = abs ((sum parts) / 2) where
```

```
    parts = zipWith (*)
```

```
            (zipWith (-) list1 list2)
```

```
            (zipWith (+) list3 list4)
```

```
    list1 = map fst poly
```

```
    list2 = tail list1 ++ [head list1]
```

```
    list3 = map snd poly
```

```
    list4 = tail list3 ++ [head list3]
```

```
ghci> area [(1,1), (1,2), (2,2), (2,1)]
```

```
1.0
```

## Zwijanie list (*list folding*)

- *foldr*  $\oplus, b, [e_0, e_1, e_2] \rightarrow (e_0 \oplus (e_1 \oplus (e_2 \oplus b)))$   
`foldr :: (a -> b -> b) -> b -> [a] -> b`  
`foldr f b [] = b`  
`foldr f b (x:xs) = f x (foldr f b xs)`
- *foldl*  $\oplus, b, [e_0, e_1, e_2] \rightarrow (((b \oplus e_0) \oplus e_1) \oplus e_2)$   
`foldl :: (a -> b -> a) -> a -> [b] -> a`  
`foldl f b [] = b`  
`foldl f b (x:xs) = foldl f (f b x) xs`

## 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
```

## Filtrowanie list

- *filter*

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

```
ghci> filter even [3, 2, 1, 4]
[2, 4]
```

- *any*

```
any :: (a -> Bool) -> [a] -> Bool
any _ []                = False
any p (x:xs) | p x      = True
                  | otherwise = any p xs
```

```
ghci> any even [3, 2, 1, 4]
True
```

- *takeWhile*

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

```
ghci> takeWhile (/= 'l') "kot Ali"
"kot A"
```

- *dropWhile*

```
dropWhile _ [] = []
dropWhile p (x:xs) | p x = dropWhile p xs
                  | otherwise = (x:xs)
```

```
ghci> dropWhile (/= 'l') "kot Ali"
"li"
```

## 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)
```

```
ghci> apply [double, square, inc] 3  
32
```

## Pytanie

Jaki jest typ funkcji apply?

```
apply :: ...
```

## Wyrażenia lambda (funkcje anonimowe)

```
ghci> (\x -> x + x) 3  
6
```

```
ghci> (\x y -> x + y) 2 3  
5
```

```
ghci> (\(x, y) -> (y, x + 1)) (1, 2)  
(2,2)
```

```
ghci> sum (map (\_ -> 1) "Ala ma kota")  
11
```



## Operator złożenia funkcji

$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$   
 $f . g = \lambda x \rightarrow f (g x)$

```
ghci> (reverse . reverse) "abcde"  
"abcde"  
ghci> (even . sum) [1..10]  
False
```

## Funkcje *uncurry*, *curry*, *flip*

- *uncurry*

$\text{uncurry} :: (a \rightarrow b \rightarrow c) \rightarrow ((a, b) \rightarrow c)$   
 $\text{uncurry } f = \lambda (x, y) \rightarrow f x y$

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

- *curry*

```
curry :: ((a, b) -> c) -> (a -> b -> c)
curry f = \x y -> f (x, y)
```

```
ghci> curry fst 'a' 1
'a'
```

- *flip*

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f = \x y -> f y x
```

```
ghci> flip (-) 2 1
-1
```

## Częściowe aplikowanie (*partial application*)

Równoważne:

- 1 `add :: Int -> Int -> Int`  
`add x y = x + y`
- 2 `add :: Int -> Int -> Int`  
`add = \x y -> x + y`
- 3 `add :: Int -> (Int -> Int)`  
`add = \x -> (\y -> x + y)`

```
ghci> map (add 1) [1, 2, 3]  
[2,3,4]
```

## Sekcje (*sections*)

Sekcja to częściowo zaaplikowany operator

```
ghci> map (1+) [1, 2, 3]  
[2,3,4]
```

```
ghci> map (>2) [1, 2, 4]  
[False,False,True]
```

```
ghci> map ("ab" ++) ["cd", "ef"]  
["abcd","abef"]
```

## Uwaga

```
ghci> map (-1) [1, 2, 3]  
błąd...  
ghci> map (flip (-) 1) [1, 2, 3]  
[0, 1, 2]
```

# Typy definiowane przez użytkownika

## Synonimy typów

Synonimy są skrótami dla już istniejących typów

Równoważne:

```
❶ roots :: (Float, Float, Float) -> (Float, Float)
```

```
❷ type Poly2 = (Float, Float, Float)  
   type Root2 = (Float, Float)
```

```
roots :: Poly2 -> Root2
```

## Uwaga

Nazwy typów danych (i synonimów) muszą zaczynać się dużą literą

## Typy użytkownika

```
data Polynom = Poly Float Float Float
```

```
Polynom    <-- nazwa typu
```

```
Poly       <-- nazwa konstruktora (funkcja)
```

```
Float      <-- typ 1go, 2go i 3go argumentu Poly
```

```
roots :: (Float, Float, Float) -> (Float, Float)
```

```
roots (a, b, c) = ...
```

```
roots' :: Polynom -> (Float, Float)
```

```
roots' (Poly a b c) = ...
```

```
p1 :: Polynom
```

```
p1 = Poly 1.0 2.0 (-1.0)
```

## Uwaga

Nazwa konstruktora może być taka sama jak nazwa typu

```
data Poly = Poly Float Float Float
```

## Typy użytkownika a dopasowanie wzorca

```
data PointType = Point Float Float
```

```
p = Point 1 2
```

```
xPoint (Point x _) = x
```

```
yPoint (Point _ y) = y
```

```
ghci> xPoint (Point 1 2)
```

```
1.0
```

```
ghci> yPoint p
```

```
2.0
```

```
data LineType = Line Point Point

dist (Line p1 p2) = sqrt ((xPoint p1 - xPoint p2)^2
                          + (yPoint p1 - yPoint p2)^2)

dist' (Line (Point x1 y1) (Point x2 y2)) =
    sqrt ((x1 - x2)^2 + (y1 - y2)^2)
```

## Przykład

```
firstQuad [] = True
firstQuad ((Point x y):ps) = (x >= 0) && (y >= 0) &&
    (firstQuad ps)
```

```
ghci> firstQuad [Point 1 2, Point 3 2, Point (-1) 1]
False
```



## 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)
```

```
area :: Shape -> Float
```

```
area (Rectangle p1 p2) =  
    abs (xPoint p1 - xPoint p2) *  
    abs (yPoint p1 - yPoint p2)
```

```
area (Circle _ r) = pi * r^2
```

```
area (Triangle p1 p2 p3) =  
    sqrt (h * (h - a) * (h - b) * (h - c))  
where  
    h = (a + b + c) / 2.0  
    a = dist p1 p2  
    b = dist p1 p3  
    c = dist p2 p3  
    dist (Point x1 y1) (Point x2 y2)  
        = sqrt ((x1 - x2)^2 * (y1 - y2)^2)
```

## Konstruktory bezargumentowe

```
data Day = Mon | Tue | Wed | Thu | Fri |  
         Sat | Sun
```

```
nameOfDay :: Day -> String
```

```
nameOfDay d = case d of  
    Mon -> "Poniedzialek"  
    Tue -> "Wtorek"  
    Wed -> "Sroda"  
    Thu -> "Czwartek"  
    Fri -> "Piatek"  
    Sat -> "Sobota"  
    Sun -> "Niedziela"
```

```
ghci> nameOfDay Fri  
"Piatek"
```

## Typy parametryzowane

```
1 data PairType a = Pair a a
```

```
p = Pair 2 5 :: PairType Int
```

```
fstPair :: PairType a -> a  
fstPair (Pair x _) = x
```

```
ghci> fstPair p  
2
```

2 `data` PairType a b = Pair a b

```
p = Pair 1 'a' :: PairType Int Char
```

```
sndPair :: PairType a b -> b
```

```
sndPair (Pair _ y) = y
```

```
ghci> sndPair p  
'a'
```

## Uwaga

Parametryzowane mogą być również synonimy typów, np.

```
type List a = [a]
```

## Typ *Maybe*

```
data Maybe a = Nothing | Just a
```

```
safediv :: Int -> Int -> Maybe Int  
safediv _ 0 = Nothing  
safediv m n = Just (m 'div' n)
```

```
safehead :: [a] -> Maybe a  
safehead [] = Nothing  
safehead (x:xs) = Just x
```

```
ghci> safediv 3 2  
Just 1  
ghci> safediv 3 0  
Nothing  
ghci> safehead "haskell"  
Just 'h'  
ghci> safehead []  
Nothing
```

## Typ *Either*

```
data Either a b = Left a | Right b
```

```
foo :: Bool -> Either Char String
```

```
foo x = case x of  
    True -> Left 'a'  
    _     -> Right "a"
```

```
ghci> foo True
```

```
Left 'a'
```

```
ghci> foo False
```

```
Right "a"
```

## Typy rekurencyjne

*Liczba naturalna to "zero" lub jej następnik*

```
data Nat = Zero | Succ Nat
```

```
n = Zero
```

```
n1 = Succ Zero
```

```
n2 = Succ (Succ Zero)
```

```
add :: Nat -> Nat -> Nat
```

```
add m Zero = m
```

```
add m (Succ n) = Succ (add m n)
```

```
nat2int :: Nat -> Int
```

```
nat2int Zero = 0
```

```
nat2int (Succ n) = 1 + nat2int n
```

```
ghci> nat2int (add n1 n2)
```

```
3
```



## Przykład – lista

*Lista jest pusta, albo składa się z głowy i listy*

```
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
```

```
ghci> len l
```

```
3
```

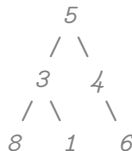
## Przykład – drzewo binarne

*Drzewo binarne 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))
```



```
depth :: Tree a -> Int
```

```
depth Empty = 0
```

```
depth (Node _ l r) = 1 + max (depth l) (depth r)
```

```
ghci> depth t
```

```
3
```

## Przechodzenie drzewa

Sposoby przechodzenia drzewa:

- *preorder* – wierzchołek zostaje odwiedzony zanim odwiedzone zostaną jego poddrzewa
- *inorder* – wierzchołek zostaje odwiedzony po odwiedzeniu lewego i przed odwiedzeniem jego prawego poddrzewa
- *postorder* – wierzchołek zostaje odwiedzony po odwiedzeniu jego lewego i prawego poddrzewa

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

```
preorder :: Tree a -> [a]
```

```
preorder Empty = []
```

```
preorder (Node a l r) = [a] ++ preorder l ++ preorder r
```

```
inorder Empty = []
```

```
inorder (Node a l r) = inorder l ++ [a] ++ inorder r
```

```
postorder Empty = []
```

```
postorder (Node a l r) = postorder l ++ postorder r ++ [a]
```

```
ghci> preorder t
```

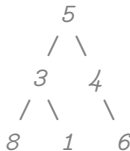
```
[5, 3, 8, 1, 4, 6]
```

```
ghci> inorder t
```

```
[8, 3, 1, 5, 4, 6]
```

```
ghci> postorder t
```

```
[8, 1, 3, 6, 4, 5]
```



## Ćwiczenie

Zdefiniuj typ reprezentujący drzewa o dowolnej liczbie poddrzew:

```
data Tree a = ...
```

Napisz funkcję obliczającą głębokość takiego drzewa:

```
depth :: Tree a -> Int  
...
```

## Jeszcze o konstruktorach

```
data PairType a b = Pair a b
```

```
ghci> :type Pair  
Pair :: a -> b -> PairType a b
```

```
ghci> :type Pair 'a'  
Pair 'a' :: b -> PairType Char b
```

```
ghci> :type Pair 'a' "hello"  
Pair 'a' "hello" :: PairType Char [Char]
```

# Parsery funkcyjne

## Typ reprezentujący parsery

Parser to funkcja przyjmująca napis i zwracająca

- 1 wartość

```
type Parser a = String -> a
```

- 2 wartość i nieskonsumowaną część napisu

```
type Parser a = String -> (a, String)
```

- 3 j.w. i lista pusta oznacza porażkę, a jednoelementowa sukces

```
type Parser a = String -> [(a, String)]
```

## Podstawowe parsery

- parser *item* kończy się niepowodzeniem jeżeli wejściem jest [], a w przeciwnym razie konsumuje pierwszy znak

```
item :: Parser Char
item []      = []
item (x:xs) = [(x, xs)]
```

- parser *failure* zawsze kończy się niepowodzeniem

```
failure :: Parser a
failure _ = []
```

- parser *return v* zwraca wartość *v* bez konsumowania wejścia

```
return :: a -> Parser a
return v = \inp -> [(v, inp)]
```



- parser  $p \text{ +++ } q$  zachowuje się jak parser  $p$  jeżeli ten kończy się powodzeniem, a w przeciwnym razie jak parser  $q$

```
(+++)  
p +++ q = \inp -> case p inp of  
    [] -> q inp  
    [(v, out)] -> [(v, out)]
```

- funkcja *parse* aplikuje parser do napisu

```
parse :: Parser a -> String -> [(a, String)]  
parse p inp = p inp
```

## Przykłady

```
ghci> parse item ""  
[]
```

```
ghci> parse item "abc"  
[('a',"bc")]
```

```
ghci> parse failure "abc"  
[]
```

```
ghci> parse (return 1) "abc"  
[(1,"abc")]
```

```
ghci> parse (item +++ return 'd') "abc"  
[('a',"bc")]
```

```
ghci> parse (failure +++ return 'd') "abc"  
[('d',"abc")]
```

## Operator sekwencji

```
(>>=) :: Parser a -> (a -> Parser b) -> Parser b  
p >>= f = \inp -> case parse p inp of  
    [] -> []  
    [(v, out)] -> parse (f v) out
```

*Idea działania:*

```
                                porażka  
-----> porażka  
  
(p >>= f) inp --> p inp  
                                sukces (v,out)  
-----> (f v) out
```

## Wyrażenie *do*

Równoważne:

1 `p1 >>= (\v1 -> (p2 >>= (\v2 -> return (g v1 v2))))`

2 `do v1 <- p1  
 v2 <- p2  
 return (g v1 v2)`

*Tzn. zaaplikuj parser p1 i rezultat nazwij v1, następnie zaaplikuj parser p2 i jego rezultat nazwij v2, na koniec zaaplikuj parser 'return (g v1 v2)'*

## Uwagi

- Wartość zwrócona przez ostatni parser jest wartością całego wyrażenia, chyba że któryś z wcześniejszych parserów zakończył się niepowodzeniem
- Rezultaty pośrednich parserów nie muszą być nazywane, jeśli nie będą potrzebne

## Przykład

```
p :: Parser (Char, Char)
p = do x <- item
      item
      y <- item
      return (x, y)
```

```
ghci> parse p "abcdef"
[('a','c'), "def"]
```

```
ghci> parse p "ab"
[]
```

## Dalsze prymitywy

- parser *sat p* konsumuje i zwraca pierwszy znak jeśli ten spełnia predykat *p*, a w przeciwnym razie kończy się niepowodzeniem

```
sat :: (Char -> Bool) -> Parser Char
sat p = do x <- item
        if p x then return x else failure
```

- parsery cyfr i wybranych znaków

```
digit :: Parser Char
digit = sat isDigit
```

```
char :: Char -> Parser Char
char x = sat (== x)
```

- funkcja *many* aplikuje parser wiele razy, kumulując rezultaty na liście, dopóki parser nie zakończy się niepowodzeniem

```
many :: Parser a -> Parser [a]
many p = many1 p +++ return []
```

- funkcja *many1* aplikuje parser wiele razy, kumulując rezultaty na liście, ale wymaga aby przynajmniej raz parser zakończył się sukcesem

```
many1 :: Parser a -> Parser [a]
many1 p = do v <- p
            vs <- many p
            return (v:vs)
```

```
ghci> parse (many digit) "123abc"  
[("123","abc")]
```

```
ghci> parse (many digit) "abcdef"  
[("", "abcdef")]
```

```
ghci> parse (many1 digit) "abcdef"  
[]
```



## Przykład

Parser kumulujący cyfry z napisu w formacie „[cyfra,cyfra,...]”

```
p :: Parser String
p = do char '['
      d <- digit
      ds <- many (do char ','
                     digit)
      char ']'
      return (d:ds)
```

```
ghci> parse p "[1,2,3]"
("123","")
```

```
ghci> parse p "[1,2,3"
[]
```

## Wyrażenia arytmetyczne

Niech wyrażenie może być zbudowane z cyfr, operacji '+' i '\*' oraz nawiasów. Operacje '+' i '\*' są prawostronnie łączne, a '\*' ma wyższy priorytet niż '+'.

Gramatyka bezkontekstowa:

$\text{expr} ::= \text{term} \text{ '+' } \text{expr} \mid \text{term}$

$\text{term} ::= \text{factor} \text{ '*' } \text{term} \mid \text{factor}$

$\text{factor} ::= \text{digit} \mid \text{'(' expr ') '}$

$\text{digit} ::= \text{'0'} \mid \text{'1'} \mid \dots \mid \text{'9'}$

Reguły gramatyki można jeszcze uprościć ( $e$  oznacza pusty napis):

$\text{expr} ::= \text{term} ('+' \text{ expr} \mid e)$

$\text{term} ::= \text{factor} ('*' \text{ term} \mid e)$

$\text{factor} ::= \text{digit} \mid '(' \text{ expr} ')'$

$\text{digit} ::= '0' \mid '1' \mid \dots \mid '9'$

Parser obliczający wartości wyrażeń arytmetycznych:

```
expr :: Parser Int
expr = do t <- term
      do char '+'
        e <- expr
        return (t + e)
      +++
      return t
```

```
term :: Parser Int
term = do f <- factor
      do char '*'
        t <- term
        return (f * t)
      +++
      return f

factor :: Parser Int
factor = do d <- digit
         return (digitToInt d)
      +++
      do char '('
        e <- expr
        char ')'
        return e
```

```
eval :: String -> Int
eval inp = case parse expr inp of
    [(n, [])] -> n
    [(_, out)] -> error ("nieskonsumowane " ++ out)
    [] -> error "bledne wejście"
```

```
ghci> eval "2*3+4"
10
```

```
ghci> eval "2*(3+4)"
14
```

```
ghci> eval "2*3-4"
*** Exception: nieskonsumowane -4
```

```
ghci> eval "-1"
*** Exception: bledne wejście
```

# Klasy typów

## Rodzaje polimorfizmu

- Polimorfizm parametryczny

```
elem x [] = False
elem x (y:ys) | x == y = True
               | otherwise = elem x ys
```

*elem :: ... a -> [a] -> Bool*

- Przeładowanie (*overloading*)

*(==) :: ... a -> a -> Bool*

*Jeśli Integer to*    *x == y*    = integerEq x y

*Jeśli krotka to*    *x == y*    = tupleEq x y

*Jeśli lista to*    *x == y*    = listEq x y

## Definiowanie klas typów

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

*Typ a jest instancją klasy Eq jeżeli istnieją dla niego operacje == i /=*

```
ghci> :type (==)  
(==) :: Eq a => a -> a -> Bool
```

*Jeżeli typ a jest instancją Eq, to (==) ma typ a -> a -> Bool*

```
ghci> :type elem  
elem :: Eq a => a -> [a] -> Bool
```

## Deklarowanie instancji klas typów

```
data Bool = False | True
```

```
instance Eq Bool where
    False == False = True
    True == True = True
    _ == _ = False
```

*Bool jest instancją Eq i definicja operacji  
(==) jest następująca (metoda)*



```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

```
ghci> elem Empty [(Node 1 Empty Empty), Empty]
No instance for (Eq (Tree a))
arising from a use of 'elem'
```

```
instance Eq a => Eq (Tree a) where
    Empty == Empty = True
    (Node a1 l1 r1) == (Node a2 l2 r2) = (a1 == a2) &&
                                           (l1 == l2) &&
                                           (r1 == r2)
    _ == _ = False
```

```
ghci> elem Empty [(Node 1 Empty Empty), Empty]
True
```

## Metody domyślne

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y      = not (x == y)
```

## Dziedziczenie (*inheritance*)

```
class Eq a => Ord a where
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min           :: a -> a -> a
    x < y               = x <= y && x /= y
```

*Ord jest podklasą Eq (każdy typ klasy Ord musi być też instancją klasy Eq)*

## Uwaga

Dziedziczenie może być wielokrotne

## Pamiętaj o kontekście

```
qsort :: [a] -> [a]
qsort []      = []
qsort (x:xs) = qsort (filter (< x) xs)
               ++ [x]
               ++ qsort (filter (>= x) xs)
```

```
ghci> :load qsort.hs
No instance for (Ord a) arising from use of '>='
Possible fix: add (Ord a) to the type signature(s)
for 'qsort'
```

```
qsort :: Ord a => [a] -> [a]
```

## Podstawowe klasy typów (Prelude.hs)

- Eq, Ord, Show, Read, Num, Enum

## Klasa *Show*

```
class Show a where  
    show :: a -> String
```

```
ghci> show 123  
"123"
```

## Klasa *Read*

```
read :: Read a => String -> a
```

```
ghci> read "123"  
Ambiguous type variable 'a'
```

```
ghci> (read "123") :: Float  
123.0
```

```
ghci> read "123" + 7  
130
```

## Klasa *Num*

```
class (Eq a, Show a) => Num a where
    (+), (-), (*) :: a -> a -> a
    negate      :: a -> a
    abs, signum  :: a -> a
    x - y        = x + negate y
    negate x     = 0 - x
```

```
ghci> 1.1 + 2.2
3.3
ghci> negate 3.3
-3.3
ghci> abs (-3)
3
ghci> signum (-3)
-1
```

## Klauzula *deriving*

```
data Tree a = Empty | Node a (Tree a) (Tree a)
             deriving (Eq, Show)
```

*Automatycznie utworzy instancje:*

```
instance Eq a => Eq (Tree a) where ...
instance Show a => Show (Tree a) where ...
```

```
ghci> (Node 1 Empty Empty) == Empty
False
ghci> show (Node 1 Empty Empty)
"Node 1 Empty Empty"
```

## Uwaga

Klauzula `deriving` może być użyta do tworzenia instancji klas:  
`Eq`, `Ord`, `Show`, `Read`, `Enum`

## Klasa *Enum*

```
class Enum a where
```

```
    succ, pred :: a -> a
```

```
    toEnum      :: Int -> a
```

```
    fromEnum    :: a -> Int
```

```
data Day = Mon | Tue | Wed | Thu | Fri |  
          Sat | Sun deriving (Show, Enum)
```

```
ghci> succ Mon
```

```
Tue
```

```
ghci> pred Mon
```

```
*** Exception: tried to take 'pred' of first tag  
            in enumeration
```

```
ghci> (toEnum 5) :: Day
```

```
Sat
```

```
ghci> fromEnum Mon
```

```
0
```

## Przykład – baza danych

Damir Medak, Gerhard Navratil: *Haskell-Tutorial*

```
type ID      = Int
type Attrib  = (String, String)
data Object  = Obj ID [Attrib] deriving Show

object :: ID -> [Attrib] -> Object
object i as = Obj i as

getID :: Object -> ID
getID (Obj i as) = i

getAtts :: Object -> [Attrib]
getAtts (Obj i as) = as

getName :: Object -> String
getName o =
    snd (head (filter ((== "name").fst) (getAtts o)))
```



```

class Databases d where
  empty      :: d
  getLastID  :: d -> ID
  getObjects :: d -> [Object]
  setLastID  :: ID -> d -> d
  setObjects :: [Object] -> d -> d

insert :: [Attrib] -> d -> d
insert as db = setLastID i' db' where
  db' = setObjects os' db
  os' = o : os
  os  = getObjects db
  o   = object i' as
  i'  = 1 + getLastID db
select :: ID -> d -> Object
select i db =
  head (filter ((== i).getID) (getObjects db))
selectBy :: (Object -> Bool) -> d -> [Object]
selectBy f db = filter f (getObjects db)

```

```
data DBS = DB ID [Object] deriving Show
```

```
instance Databases DBS where
```

```
    empty                = DB 0 []
    getLastID (DB i os)   = i
    setLastID i (DB j os) = DB i os
    getObjects (DB i os)  = os
    setObjects os (DB i ps) = DB i os
```

```
d0, d1, d2 :: DBS
```

```
d0 = empty
```

```
d1 = insert [("name", "john"), ("age", "30")] d0
```

```
d2 = insert [("name", "mary"), ("age", "20")] d1
```

```
ghci> select 1 d1
```

```
Obj 1 [("name", "john"), ("age", "30")]
```

```
ghci> selectBy ((== "mary").getName) d2
```

```
[Obj 2 [("name", "mary"), ("age", "20")]]
```

# Programy interaktywne

## Typ reprezentujący operacje IO

- 1 funkcja zmieniająca „stan świata”

```
type IO = World -> World
```

- 2 funkcja zmieniająca „stan świata” i zwracająca wynik

```
type IO a = World -> (a, World)
```

## Akcje

Akcja to wyrażenie typu `IO a`

```
IO Char  <-- typ akcji zwracającej znak
```

```
IO ()    <-- typ akcji zwracającej pustą krotkę
```

## Typ jednostkowy

```
data () = ()
```

## Podstawowe akcje

- akcja *getChar* wczytuje znak z klawiatury, wyświetla go na ekranie i zwraca jako rezultat

```
getChar :: IO Char
```

- akcja *putChar c* wyświetla znak *c* na ekranie i zwraca pustą krotkę

```
putChar :: Char -> IO ()
```

```
ghci> putChar 'a'  
'a'
```

- akcja *return v* zwraca wartość *v* bez jakichkolwiek interakcji

```
return :: a -> IO a  
return v = \world -> (v, world)
```

## Operator sekwencji

```
(>>=) :: IO a -> (a -> IO b) -> IO b  
f >>= g = \world -> case f world of  
    (v, world') -> g v world'
```

## Uwaga

Jak w przypadku parserów zamiast operatora `>>=` można korzystać z notacji `do`

## Przykład

```
a :: IO (Char, Char)  
a = do x <- getChar  
      getChar  
      y <- getChar  
      return (x, y)
```

## Dalsze prymitywy

- *getLine*

```
getLine :: IO String
getLine = do x <- getChar
            if x == '\n' then return []
            else
                do xs <- getLine
                   return (x:xs)
```

- *putStr*

```
putStr :: String -> IO ()  
putStr []      = return ()  
putStr (x:xs) = do putChar x  
                  putStr xs
```

- *putStrLn*

```
putStrLn :: String -> IO ()  
putStrLn xs = do putStr xs  
                putChar '\n'
```

## Przykład

```
strlen :: IO ()
strlen = do putStr "Enter a string: "
           xs <- getLine
           putStr "The string has "
           putStr (show (length xs))
           putStrLn " characters"
```

```
ghci> strlen
Enter a string: Ala ma kota
The string has 11 characters
```



## Przykład

```
ghci> hangman
Think of a word:
-----
Try to guess it:
> basic
-as----
> pascal
-as--ll
> haskell
You got it!
```

```
hangman :: IO ()
hangman = do putStrLn "Think of a word:"
            word <- sgetLine
            putStrLn "Try to guess it:"
            guess word
```

```
sgetline :: IO String
sgeline = do x <- getCh
           if x == '\n' then
             do putChar x
                return []
           else
             do putChar '-'
                xs <- sgetline
                return (x:xs)
```

```
getCh :: IO Char
getCh = do hSetEcho stdin False
           c <- getChar
           hSetEcho stdin True
           return c
```

```
guess :: String -> IO ()
guess word = do putStr "> "
               xs <- getLine
               if xs == word then
                 putStrLn "You got it!"
               else
                 do putStrLn (diff word xs)
                    guess word
```

```
diff :: String -> String -> String
diff xs ys = [if elem x ys then x else '-' | x <- xs]
```

```
ghci> diff "haskell" "pascal"
"-as--ll"
```

## Definicje lokalne

```
power = do putStr "Podaj liczbe: "  
          n <- getLine  
          let x = read n  
              y = x^2  
          putStrLn (n ++ " do kwadratu: " ++ show y)
```

```
ghci> power  
Podaj liczbe: 12  
12 do kwadratu: 144
```

## Biblioteka IO (`import System.IO`)

```
data IOMode = ReadMode | WriteMode | AppendMode |  
            ReadWriteMode
```

```
type FilePath = String
```

```
openFile :: FilePath -> IOMode -> IO Handle
```

```
hClose   :: Handle -> IO ()
```

```
hIsEOF   :: Handle -> IO Bool
```

```
hGetChar :: Handle -> IO Char
```

```
hGetLine :: Handle -> IO String
```

```
hGetContents :: Handle -> IO String
```

```
getChar :: IO Char  <-- hGetChar stdin
```

```
getLine :: IO String
```

```
hPutChar    :: Handle -> Char -> IO ()
hPutStr     :: Handle -> String -> IO ()
hPutStrLn   :: Handle -> String -> IO ()

putChar     :: Char -> IO ()    <-- hPutChar stdout
putStr      :: String -> IO ()
putStrLn    :: String -> IO ()

readFile    :: FilePath -> IO String
writeFile   :: FilePath -> String -> IO ()
appendFile  :: FilePath -> String -> IO ()
```

## Przykład

```
doLoop = do
  putStrLn "Wybierz polecenie cNazwaPliku,
           zNazwaPliku lub w (wyjście):"
  cmd <- getLine
  case cmd of
    'w':_      -> return ()
    'c':fname -> do putStrLn ("Odczyt " ++ fname)
                  doRead fname
                  doLoop
    'z':fname -> do putStrLn ("Zapis " ++ fname)
                  doWrite fname
                  doLoop
    _          -> doLoop
```

```
doRead fname = do handle <- openFile fname ReadMode
                  contents <- hGetContents handle
                  putStrLn "Pierwsze 100 znakow:"
                  putStrLn (take 100 contents)
                  hClose handle
```

```
doWrite fname = do putStrLn "Wpisz tekst:"
                  contents <- getLine
                  handle <- openFile fname WriteMode
                  hPutStrLn handle contents
                  hClose handle
```



## Rodzaje błędów IO (`import System.IO.Error`)

```
data IOError = ... <-- typ błędu
```

```
isDoesNotExistError :: IOError -> Bool  
isAlreadyInUseError  :: IOError -> Bool  
isPermissionError    :: IOError -> Bool  
isEOFError           :: IOError -> Bool
```

## Obsługa błędów

Akcja *catch* definiuje obsługę błędów, które może zgłosić akcja:

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

*Idea działania:*

```
                ok  
            -----> a  
catch a f --> a  
                błąd e  
            -----> f e
```

## Przykład

```
readLine :: Handle -> IO String
readLine h = catch (hGetLine h) (\e -> return "")
```

## ioError

Akcja *ioError* przekazuje nieobsłużony błąd dalej

```
ioError :: IOError -> IO a
```

## Przykład

```
readLine :: Handle -> IO String
readLine h = catch (hGetLine h) errorHandler
    where
        errorHandler e = if isEOFError e
                           then return ""
                           else ioError e
```

## Przykład

Argumentem *catch* może być sekwencja akcji

```
doRead :: String -> IO ()
doRead fname =
    catch (do handle <- openFile fname ReadMode
              contents <- hGetContents handle
              putStrLn "Pierwsze 100 znakow:"
              putStrLn (take 100 contents)
              hClose handle
    ) errorHandler
    where
        errorHandler e =
            if isDoesNotExistError e
            then putStrLn ("Nie istnieje " ++ fname)
            else return ()
```

# Typy monadyczne

## Klasa *Monad*

W przypadku parserów i programów interaktywnych definiowaliśmy:

- 1 `return :: a -> Parser a`  
`(>>=) :: Parser a -> (a -> Parser b) -> Parser b`
- 2 `return :: a -> IO a`  
`(>>=) :: IO a -> (a -> IO b) -> IO b`

Ogólnie:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

## Uwaga

Notacja `do` może być użyta z dowolnym typem monadycznym

# Moduły

## Przykład

*plik: Tree.hs*

```
module Tree where
data Tree a = Empty | Node a (Tree a) (Tree a)
```

```
depth :: Tree a -> Int
depth Empty          = 0
depth (Node _ l r) = 1 + max (depth l) (depth r)
```

*plik: Main.hs*

```
module Main where
import Tree
main = putStrLn (show (depth (Node 1 Empty Empty)))
```

## Uwaga

Każdy moduł zaczyna się domyślnie od `import Prelude`

## Eksportowanie wybranych nazw

```
module Tree (Tree(Empty, Node), depth) where
  lub
module Tree (Tree(...), depth) where
```

## Importowanie nazw

```
import Tree
a = depth ...   lub   Tree.depth ...
```

```
import qualified Tree
a = Tree.depth ...   <-- jedyna możliwość
```

```
import qualified Tree as T
a = Tree.depth ...   lub   T.depth ...
```

## Importowanie wybranych nazw

```
import Tree (depth)    <-- tylko depth  
import Tree hiding (depth) <-- wszystko oprócz depth
```

# Leniwe wartościowanie

Wartościowanie/redukcja wyrażeń polega na aplikowaniu stosownych definicji do momentu, gdy kolejna aplikacja nie będzie możliwa

## Przykład

`inc n = n + 1`

<code>inc (2 * 3)</code>		<code>inc (2 * 3)</code>
<code>=</code>		<code>=</code>
<code>inc 6</code>	<i>lub</i>	<code>(2 * 3) + 1</code>
<code>=</code>		<code>=</code>
<code>6 + 1</code>		<code>6 + 1</code>
<code>=</code>		<code>=</code>
<code>7</code>		<code>7</code>

## Uwaga

W Haskellu dowolne dwa sposoby wartościowania jednego wyrażenia dają zawsze tę samą wartość (pod warunkiem, że oba się zakończą)



## Strategie wartościowania

W pierwszej kolejności zawsze redukuj:

- 1 najbardziej wewnętrzne podwyrażenie, które można zredukować (*innermost reduction*)
- 2 najbardziej zewnętrzne podwyrażenie, które można zredukować (*outermost reduction*)

## Problem stopu

`inf = inf + 1`

### ① *innermost reduction*

`fst (0, inf)`  
=  
`fst (0, 1 + inf)`  
=  
`fst (0, 1 + (1 + inf))`  
=  
*itd.*

### ② *outermost reduction*

`fst (0, inf)`  
=  
0

## Uwagi

- Strategia *outermost reduction* może zwrócić wynik w sytuacji, gdy *innermost reduction* prowadzi do nieskończonych obliczeń
- Jeżeli dla danego wyrażenia istnieje kończąca się sekwencja redukcji, to *outermost reduction* również się kończy (z tym samym wynikiem)

## Liczba redukcji

square  $n = n * n$

### ① *innermost reduction*

square (1 + 2)  
=  
square 3  
=  
3 \* 3  
=  
9

### ② *outermost reduction*

square (1 + 2)  
=  
(1 + 2) \* (1 + 2)  
=  
3 \* (1 + 2)  
=  
3 \* 3  
=  
9

## Uwagi

- Strategia *outermost reduction* może wymagać większej liczby kroków niż *innermost reduction*
- Problem można rozwiązać współdzieląc wyniki wartościowania argumentów

```

square (1 + 2)
=
  _ * _      1 + 2
=
  _ * _      3
=
  9

```

## Uwagi

- Leniwe wartościowanie (*lazy evaluation*) = *outermost reduction* + współdzielenie wyników wartościowania argumentów
- Leniwe wartościowanie nigdy nie wymaga więcej kroków niż *innermost reduction*
- Haskell stosuje leniwe wartościowanie

## Listy nieskończone

```
ones :: [Int]
```

```
ones = 1 : ones
```

```
ghci> ones
```

```
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,... <Ctrl-C>
```

```
ghci> head ones
```

```
1
```

```
head ones
```

```
=
```

```
head (1 : ones)
```

```
=
```

```
1
```

## Uwaga

Korzystając z leniwego wartościowania, wyrażenia są wartościowane tylko w takim stopniu w jakim jest to potrzebne do obliczenia ich rezultatu

## Przykład

```
take 0 _      = []
take _ []     = []
take n (x:xs) =
    x : take (n - 1) xs
```

```
ghci> take 3 ones
[1,1,1]
```

```
take 3 ones
= ones
take 3 (1 : ones)
= take
  1 : take 2 ones
= ones
  1 : take 2 (1 : ones)
= take
  1 : 1 : take 1 ones
= ones
  1 : 1 : take 1 (1 : ones)
= take
  1 : 1 : 1 : take 0 ones
= take
  1 : 1 : 1 : []
=
[1,1,1]
```