

# Programowanie funkcyjne HASKELL

## Moduły cd.

base: Basic libraries

[with: show, pretty, IO, Monad, etc.]

This package contains the Prelude and its support libraries, and a large collection of useful libraries ranging from data structures to parsing, containers and debugging utilities.

**Modules**

- base (this package)
- Control
- Control.Applicative
- Control.Arrow
- Control.Category
- Control.Concurrent
- Control.Concurrent.Chan
- Control.Concurrent.MVar
- Control.Concurrent.Queue
- Control.Concurrent.QSem
- Control.Concurrent.QSemM
- Control.Exception
- Control.Exception.Base
- Control.Monad

**Data**

- Data.Bifunctor
- Data.Bifunctor
- Data.Bitraversable
- Data.Bits
- Data.Bool
- Data.Char
- Data.Coerce
- Data.Complex
- Data.Delta
- Data.Dynamic
- Data.Either
- Data.Eq
- Data.Fixed
- Data.Foldable
- Data.Function
- Data.Function
- Data.Function.Classes
- Data.Function.Compose
- Data.Function.Const
- Data.Function.Identity
- Data.Function.Product
- Data.Function.Sum
- Data.IORef
- Data.Function.Classes
- Data.Function.Compose
- Data.Function.Const
- Data.Function.Identity
- Data.Function.Product
- Data.Function.Sum
- Data.IORef
- Data.Function.Classes
- Data.Function.Compose
- Data.Function.Const
- Data.Function.Identity
- Data.Function.Product
- Data.Function.Sum
- Data.IORef

## Moduły cd.

import nazwa\_modulu

Przykład:

```
import Data.List           (wszystkie)
import Data.List (nub, sort)  (tylko nub i sort)
import Data.List hiding nub  (wszystkie z wyjątkiem nub)
```

:m + nazwa\_modulu

Przykład:

```
Prelude> :m + Data.List
Prelude Data.List> nub [1,2,3,4,1,2,1,2]
[1,2,3,4]
Prelude Data.List> (\xs -> length (nub xs)) [1,3,4,1,3,1,3]
3
```

## Data.List

### intersperse

```
Prelude Data.List> intersperse ' ' "PPD"
"P.P.D"
Prelude Data.List> intersperse '0' "PPD"
"P0P0D"
```

### intercalate

```
Prelude Data.List> intercalate "-" ["Podstawy", "programowania", "deklaratywnego"]
"Podstawy_programowania_deklaratywnego"
Prelude Data.List> intercalate [0,0] [[1,1], [2,2,2], [3,3,3,3]]
[1,1,0,0,2,2,2,0,0,3,3,3,3]
```

## Data.List

### transpose

```
Prelude Data.List> transpose [[1,2,3],[4,5,6],[7,8,9]]
[[1,4,7],[2,5,8],[3,6,9]]
Prelude Data.List> transpose ["abc", "def"]
["ad", "be", "cf"]
```

### concat, concatMap

```
Prelude Data.List> replicate 4 [2,3]
[[2,3],[2,3],[2,3],[2,3]]
Prelude Data.List> concat (replicate 4 [2,3])
[2,3,2,3,2,3,2,3]
Prelude Data.List> concatMap (replicate 4) [2,3]
[2,2,2,2,3,3,3,3]
```

## Data.List

and

```
Prelude Data.List> and $ map (>4) [5,6,7,8]
True
Prelude Data.List> and $ map (==4) [4,4,4,3,4]
False
```

or

```
Prelude Data.List> or $ map (==4) [1,2,3,4,5,6]
True
Prelude Data.List> or $ map (>5) [1,2,3,4]
False
```

## Data.List

any

```
Prelude Data.List> any (==4) [2,3,5,6,1,4]
True
Prelude Data.List> all (>4) [6,9,10]
True
```

all

```
Prelude Data.List> all (>4) [1,2,3]
False
Prelude Data.List> any ('elem' ['A'..'Z']) "HASKELL"
True
Prelude Data.List> all ('elem' ['A'..'Z']) "HASKELL"
True
Prelude Data.List> all ('elem' ['A'..'Z']) "HASKELLghci"
False
Prelude Data.List> any ('elem' ['A'..'Z']) "HASKELLghci"
True
```

## Data.List

iterate

```
Prelude Data.List> take 10 $ iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512]
Prelude Data.List> take 3 $ iterate (++ " Has ") "kell "
["kell ", "kell Has ", "kell Has Has "]
Prelude Data.List> take 4 $ iterate (++ " kell ") "Has "
["Has ", "Has kell ", "Has kell kell ", "Has kell kell kell "]
```

splitAt

```
Prelude Data.List> splitAt 3 "Haskell"
("Has", "kell")
Prelude Data.List> splitAt 50 "Haskell"
("Haskell", "")
Prelude Data.List> splitAt (-3) "Haskell"
("", "Haskell")
Prelude Data.List> let (a,b) = splitAt 3 "Haskell " in b ++ a
"skell Ha"
```

## Data.List

takeWhile, dropWhile, span, break,  
sort, group, inits, tails, isPrefixOf, isSuffixOf,  
elem, notElem, partition, find, elemIndex, elemIndices,  
finfIndex, findIndices, zipWith3, zipWith4, lines, unlines,  
words, unwords, delete, \, \, union, intersect, insert,

## Data

Data.Bifoldable  
Data.Bifunctor  
Data.BitTraversable  
Data.Bits  
Data.Bool  
Data.Char  
Data.Coerce  
Data.Complex  
Data.Data  
Data.Dynamic  
Data.Either  
Data.Eq  
Data.Fixed  
Data.Foldable  
Data.Function  
Data.Functor



## Typ Complex

Typ **Complex** dostępny jest po załadowaniu biblioteki Complex, umożliwia wykonywanie obliczeń na liczbach zespolonych.

Na przykład, aby obliczyć  $\sqrt{8+6i}$

należy napisać:

```
> import Complex
> sqrt (8:+6i)
3.0 :+ 1.0
```

## Sumowanie

Definiujemy funkcję

$$\sum_{i=1}^k f(e_i) = f(e_1) + f(e_2) + \dots + f(e_k).$$

gdzie

$$a_i = a, \quad a_i \leq b \leq a_{i+1}, \quad a_i = \text{next}(a_{i-1}), \quad i = 1, \dots, k$$

następująco

$$\text{suma } f \text{ a next } b \quad \begin{cases} a > b & = 0 \\ \text{otherwise} & = (f \text{ a}) + \text{suma } f \text{ (next a) next b} \end{cases}$$

next x = x+1

```
*Main> f 6
36
*Main> suma f 1 next 6
91
```

## Sumowanie

Wykorzystanie funkcji *suma* do sumowania liczb od *a* do *b*

$$\sum_{i=a}^b i = a + (a+1) + \dots + b$$

```
suma_liczb a b = suma f a next b
  where f x = x
        next x = x+1
```

```
*Main> suma_liczb 1 10
55
*Main> suma_liczb 10 20
165
```

## Obliczanie całek oznaczonych metodą prostokątów

$$\int_a^b g(x) dx \approx \left[ g\left(a + \frac{dx}{2}\right) + g\left(a + dx + \frac{dx}{2}\right) + g\left(a + 2dx + \frac{dx}{2}\right) + \dots \right] \cdot dx$$

Zakładamy, że krok *dx* jest dany.

$$\text{W tym przypadku } f(x) = g\left(x + \frac{dx}{2}\right) \quad \text{oraz} \quad \text{next}(x) = x + dx.$$

```
całka g a b dx = dx * suma f a next b
  where f x = g (x+dx/2)
        next x = x + dx
```

## Obliczanie całek oznaczonych metodą prostokątów

$$\int_1^2 \sqrt{1+x^2} \, dx = 0.01$$

```
*Main> całka (\x->(1+x*x)**(1/3)) 1 2 0.01
1.4823761774960418
```

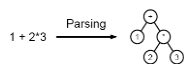
$$\int_{\pi/6}^{\pi/2} \sqrt{x} \sin x \, dx = 0.01$$

```
*Main> całka (\x->(sqrt x) * sin x) (pi/6) (pi/2) 0.01
0.9036112885201439
```

## Parsery

**Analizator składniowy** lub **parser** – program dokonujący **analizy składniowej** danych wejściowych w celu określenia ich struktury gramatycznej w związku z określoną gramatyką formalną.

Analizator składniowy umożliwia przetworzenie tekstu czytelnego dla człowieka w strukturę danych przydatną dla oprogramowania komputera.



Przykład parsingu wyrażeń arytmetycznych

## Typ reprezentujący parsery

**Parser** to funkcja przyjmująca napis i zwracająca:

- wartość  
type Parser a = String -> a
- wartość i nieskonsumowaną część napisu  
type Parser a = String -> (a, String)
- 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 _ = []
```

```
*Main> item ""
[]
*Main> item "anna"
[('a', "nna")]
*Main> failure ""
[]
*Main> failure "anna"
[]
```

### Podstawowe parsery

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

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

```
*Main> myreturn 1 "as"
[(1, "as")]
*Main> myreturn 2 "as"
[(2, "as")]
```

parser **p +++ q** zachowuje się jak parser p, jeżeli ten kończy się powodzeniem,  
a w przeciwnym razie jak parser q

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

```
*Main> (item +++ myreturn 'd') "as"
[('a', "s")]
*Main> (item +++ myreturn 'd') "abcd"
[('a', "bcd")]
*Main> (item +++ myreturn 'd') ""
[('d', "")]
*Main> (item +++ myreturn 'd') "a"
[('a', "")]
```

### Funkcja parse

funkcja **parse** aplikuje parser do napisu

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

```
*Main> myparse (myreturn 1) "aa"
[(1, "aa")]
*Main> myparse item ""
[]
*Main> myparse item "ala"
[('a', "la")]
*Main> myparse (myreturn 1) "ala"
[(1, "ala")]
*Main> myparse (myreturn 2) "ala"
[(2, "ala")]
```

### Wyrażenie do

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

**Oznacza:** 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)
```

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

### Podstawowe parsery

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)
```

### Podstawowe parsery

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

```
many :: Parser a -> Parser [a]
many1 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)
```

### Podstawowe parsery

```
Prelude> parse (many digit) "123abc"
```

```
[("123","abc")]
```

```
Prelude> parse (many digit) "abcdef"
```

```
[("","abcdef")]
```

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

### Wyrażenia arytmetyczne

Niech wyrażenie będzie zbudowane z cyfr, operacji dodawania (+) i mnożenia (\*) oraz nawiasów.

Operacje + i \* są prawostronnie łączne, \* ma wyższy priorytet niż +.

Gramatyka bezkontekstowa:

expr ::= term ('+' expr   e)	expr ::= term '+' expr   term
term ::= factor ('*' term   e)	term ::= factor '*' term   factor
factor ::= digit   '(' expr ')'	
digit ::= '0'   '1'   ...   '9'	

### Parser obliczający wartości wyrażeń arytmetycznych:

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

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

```
factor :: Parser Int
factor = do d <- digit
          return (read [d])
factor ::= digit | '(' expr ')'

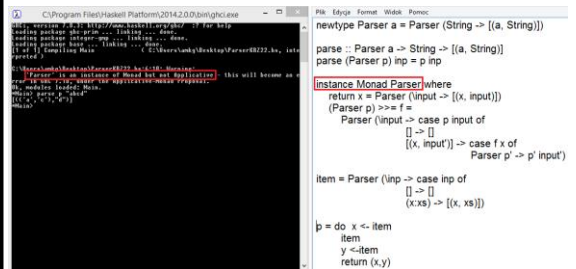
eval :: String -> Int
eval input = case parse expr input of
  [(n, [])] -> n
  [(_, out)] -> error ("nieskonsumowane " ++ out)
  [] -> error "błędne wejście"
```

```

*Main> eval "(3*2+2*6)*2"
36
*Main> eval "2+3"
5
*Main> eval "2+3*5"
17
*Main> eval "2*3+5"
11
*Main> eval "2*3-5"
*** Exception: nieskonsumowane -5
*Main> eval "(3+4)*(5+7)"
84
*Main> eval "(3*2+2*6)*2"
36

```

## Uwaga



```

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

instance Monad Parser where
  return x = Parser (\input -> [(x, input)])
  (Parser p) >>= f =
    Parser (\input -> case p input of
      [] -> []
      [(x, input')] -> case f x of
        Parser p' -> p' input')
  item = Parser (\inp -> case inp of
    [] -> []
    (x:xs) -> [(x, xs)])
  p = do x <- item
        item
        y <- item
        return (x,y)

```

## 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!
- <https://www.haskell.org/>