Programowanie funkcyjne

HASKELL

Definiowanie własnych typów

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

konstruktor typu

konstruktory (danych)

Konstruktory bezargumentowe

```
data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
           deriving (Show)
nameOfDay :: Day -> String
nameOfDay d = case d of
              Mon -> "Poniedzialek"
              Tue -> "Wtorek"
              Wed -> "Sroda"
              Thu -> "Czwartek"
              Fri -> "Piatek"
              Sat -> "Sobota"
              Sun -> "Niedziela"
```

Konstruktory bezargumentowe

```
data Color = Red | Blue | Green | Black | White
             deriving (Show)
kolor_kwiatu:: [Char]->Color

kolor_kwiatu x
    | x=="mak" = Red
    | x=="rumianek" = White
    | x=="chaber" = Blue
```

```
*Main> :t kolor_kwiatu
kolor_kwiatu :: [Char] -> Color
*Main> kolor_kwiatu "mak"
Red
*Main> kolor_kwiatu "chaber"
Blue
*Main>
```

Definiowanie własnych typów

```
data BookInfo = Book Int String [String]
                deriving (Show)
```

BookInfo <-- *nazwa typu*

Book <-- *nazwa konstruktora (funkcja)*

Int, String [String] <-- *kolejne argumenty* Book

```
*Main> myInfo
Book 980011982 "Real Word Haskell" ["B.OSullivan","J.Goerzen"]
*Main> :t myInfo
myInfo :: BookInfo
*Main> let book1 = Book 19877 "Prolog" ["E.Gatnar", "K.Stapor"]
*Main> :t book1
book1 :: BookInfo
*Main> :info BookInfo
data BookInfo = Book Int String [String]    -- Defined at typy.hs:9:1
instance Show BookInfo -- Defined at typy.hs:10:29
*Main> : Book
data BookInfo = Book Int String [String]
    -- Defined at typy.hs:9:17
*Main> :t Book
Book :: Int -> String -> [String] -> BookInfo
```

Typy parametryzowane

```
data PairType  a = Pair a a
p = Pair 2 5
fstPair :: PairType a -> a
fstPair (Pair x _) = x
```

```
*Main> :t fstPair
fstPair :: PairType a -> a
*Main> :t p
p :: PairType Integer
*Main> :t Pair 2 5
Pair 2 5 :: Num a => PairType a
*Main> fstPair p
2
```

Typy parametryzowane

```
data PairType a b = Pair a b
q = Pair 1 'a'
sndPair :: PairType a b -> b
sndPair (Pair _ y) = y
```

```
*Main> :t sndPair
sndPair :: PairTypea a b -> b
*Main> :t q
q :: PairTypea Integer Char
*Main> sndPair q
'a'
```

Synonimy typów

Synonimy typów umożliwiają nadanie własnej nazwy dla dowolnego typu. Wykorzystanie synonimów typów jest możliwe tylko w zewnętrznym pliku.

```
type Nazwisko = String
type Imie = String
type BookRecord = (Int,BookInfo)
```

Synonimy typów mogą być parametryzowane, np.

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

Synonimy typów

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

```
*Main> :t odleglosc
odleglosc :: Punkt -> Punkt -> Double
*Main> odleglosc (0,0) (1,1)
1.4142135623730951
*Main> odleglosc (-1,03) (1,5)
2.8284271247461903
```

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>
```

Typy rekurencyjne

Liczba naturalna to "zero" lub jej następnik

```
data Nat = Zero | Succ Nat
           deriving (Show)
n = Zero
n1 = Succ Zero
n2 = Succ (Succ Zero)
```

```
*Main> :t n
n :: Nat
*Main> n
Zero
*Main> n1
Succ Zero
*Main> 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
```

```
*Main> add n n1
Succ Zero
*Main> add n1 n2
Succ (Succ (Succ Zero))
*Main> add n2 n1
Succ (Succ (Succ Zero))
*Main> nat2int Zero
0
*Main> nat2int (Succ Zero)
1
*Main> nat2int (add n n2)
2
*Main> nat2int (add n1 n2)
3
```

Typy rekurencyjne – listy

Lista jest pusta, albo składa się z głowy i ogona (listy)

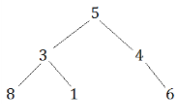
```
data List a = Empty | Cons a (List a)
            deriving (Show)
l :: List Int
l = Cons 12 (Cons 8 (Cons 10 Empty))

len :: List a -> Int
len Empty = 0
len (Cons _ xs) = 1 + len xs
```

```
*Main> :t l
l :: List Int
*Main> l
Cons 12 (Cons 8 (Cons 10 Empty))
*Main> :t len
len :: List a -> Int
*Main> len l
3
```

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))
```




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

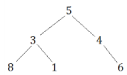
```
*Main> :t t
t :: Tree Int
*Main> t
Node 5 (Node 3 (Node 8 Empty Empty) (Node 1 Empty Empty)) (Node 4 Empty (Node 6 Empty Empty))
*Main> :t depth
depth :: Tree a -> Int
*Main> depth t
3
*Main>
```

Przechodzenie po drzewie

preorder – najpierw odwiedzony zostaje wierzchołek, a następnie odwiedzone zostaną jego 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
```

```
*Main> preorder t
[5,3,8,1,4,6]
```

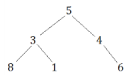


Przechodzenie po drzewie

inorder – wierzchołek zostaje odwiedzony po odwiedzeniu lewego i przed odwiedzeniem jego prawego poddrzewa

```
data Tree a = Empty | Node a (Tree a) (Tree a)
inorder :: Tree a -> [a]
inorder Empty      = []
inorder (Node a l r) = inorder l ++ [a] ++ inorder r
```

```
*Main> inorder t
[8,3,1,5,4,6]
```

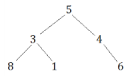


Przechodzenie po drzewie

postorder – wierzchołek zostaje odwiedzony po odwiedzeniu jego lewego i prawego poddrzewa

```
data Tree a = Empty | Node a (Tree a) (Tree a)
postorder :: Tree a -> [a]
postorder Empty = []
postorder (Node a l r) = postorder l ++ postorder r ++ [a]
```

```
*Main> postorder t
[8,1,3,6,4,5]
```



Drzewa nieskończone

```
tr = Node 5
       (Node 1 Empty Empty)
       tr
```

```
*Main> :t tr
tr :: Tree Integer
*Main> tr
Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1
mpty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (N
de 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 E
pty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (No
e 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Em
ty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Nod
 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Emp
y Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node
 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty
Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5
Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty E
pty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (
ode 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Em
ty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (N
de 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Emp
y) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (No
e 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empty) (Node 5 (Node 1 Empty Empt
```

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!