Programowanie funkcyjne

HASKELL
cz.3

Definiowanie list c.d.

```
*Main>  [x ^ 2 | x <- [1 .. 5]]
[1,4,9,16,25]
*Main>  [(x, y) | x <- [1, 2, 3], y <- [4, 5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
*Main>  [(x, y) | y <- [4, 5], x <- [1, 2, 3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
*Main>  [(x, y) | x <- [1 .. 3], y <- [x .. 3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
*Main>  [x | x <- [1 .. 10], even x]
[2,4,6,8,10]
```

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

```
*Main> firsts [(1,2),(6,7),(0,9)]
[1,6,0]
*Main> firsts [(1,"pf"),(6,[]),(2,"a")]
[1,6,2]
*Main> firsts [(1,"pf"),(6,[]),(2,'a')]

<interactive>:70:28:
    Couldn't match expected type ‘[Char]’ with actual type ‘Char’
    In the expression: 'a'
    In the expression: (2, 'a')
    In the first argument of ‘firsts’, namely
      ‘[(1, "pf"), (6, []), (2, 'a')]’
*Main>
```

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

```
*Main> factors 20
[1,2,4,5,10,20]
*Main> factors 17
[1,17]
*Main> factors 176
[1,2,4,8,11,16,22,44,88,176]
*Main>
```

Funkcje anonimowe

Funkcja anonimowa jest funkcją bez nazwy.

W Haskellu: **λ** zastępujemy przez ****

. zastępujemy przez **->**

Przykłady:

λx.fx **\x -> f**

λx.λy.fxy **\x -> \y -> f x y**

krócej: λxy.fxy **\x y -> f x y**

Funkcje anonimowe

Wszystkie funkcje są funkcjami jednorgumentowymi!

Przykład. Funkcja

dodaj:: Integer -> Integer -> Integer

dodaj = \x y -> x+y

jest funkcją jednego argumentu typu Integer zwracająca funkcję jednego argumentu typu Integer zwracającą wartość typu Integer

zwieksz1 = dodaj 1

```
*Main> :t dodaj
dodaj :: Integer -> Integer -> Integer
*Main> :t dodaj 5
dodaj 5 :: Integer -> Integer
*Main> :t dodaj 5 6
dodaj 5 6 :: Integer
*Main> zwieksz1 7
8
```

```
Prelude> (\x->x+x)3
6
Prelude> (\xy->x+y)3 4

<interactive>:24:7: Not in scope: ‘x’

<interactive>:24:9: Not in scope: ‘y’
Prelude> (\x y->x+y)3 4
7
Prelude> sum (map(\_ -> 1) "Haskell")
7
```

```
Prelude> (\x -> 2+x)4
6
Prelude> :t (\x -> 2+x)4
(\x -> 2+x)4 :: Num a => a
Prelude> (\f -> 2+f 4) sin
1.2431975046920718
Prelude> :t (\f -> 2+f 4) sin
(\f -> 2+f 4) sin :: Floating a => a
Prelude> map (\x -> 2*x^3-4*x^2+7*x-12) [1,-3,10,-12]
[-7,-123,1658,-4128]
Prelude> :t map (\x -> 2*x^3-4*x^2+7*x-12) [1,-3,10,-12]
map (\x -> 2*x^3-4*x^2+7*x-12) [1,-3,10,-12] :: Num b => [b]
```

Funkcje anonimowe

Czasami wygodniej używać lambda wyrażeń niż funkcji z daną nazwą.
Przykład:

dodj lista = map dj lista
where dj x=x+1

dodjeden lista = map (\x -> x+1) lista

Operator złożenia funkcji

(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)

```
Prelude> reverse "abcde"
"edcba"
Prelude> (reverse.reverse) "abcde"
"abcde"
Prelude> sum [1..10]
55
Prelude> (even.sum) [1..10]
False
```

Funkcja *curry*

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

```
Prelude> fst 'a' 1

<interactive>:35:5:
    Couldn't match expected type ‘(a0 -> t, b0)’
                with actual type ‘Char’
    Relevant bindings include it :: t (bound at <interactive>:35:1)
    In the first argument of ‘fst’, namely ‘'a'’
    In the expression: fst 'a' 1
    In an equation for ‘it’: it = fst 'a' 1
Prelude> fst ('a', 1)
'a'
Prelude> curry fst 'a' 1
'a'
```

Funkcja *uncurry*

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

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

Funkcja *flip*

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

```
Prelude> flip (-) 1 4
3
Prelude> div 3 4
0
Prelude> flip div 3 4
1
```

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!