Programowanie funkcyjne

HASKELL

Listy

[]
[1,2,3]
["ab","bc","cd"]

.. wyliczenie

[1..10] ozn. [1,2,3,4,5,6,7,8,9,10]
[1.0,1.25..2.0] ozn. [1.0,1.25,1.5,1.75,2.0]
[1,4..15] ozn. [1,4,7,10,13]
[10,9..1] ozn. [10,9,8,7,6,5,4,3,2,1]
['a'..'e'] ozn. ['a', 'b', 'c', 'd', 'e']

Listy nieskończone

Jeżeli ostatni element listy nie zostanie podany, Haskell utworzy listę o "nieskończonej" długości. Jest to możliwe dzięki leniwemu wartościowaniu. Wyznaczony zostanie tylko ten element listy, który będzie w danej chwili potrzebny.

[1..] ozn. [1, 2, 3, 4, 5, 6, ...
[1, 4 ..] ozn. [1, 4, 7, 10, 13, ...
take 3 [1 ..] ozn. [1,2,3]

Definiowanie list (List comprehensions)

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

```
*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]
```

Przykłady

```
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>
```

Przykłady

```
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>
```

Konstruktor list

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

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

```
Prelude> 3 : [4, 5]
[3,4,5]
Prelude> True : []
[True]
Prelude> "ab" : ["cd", "efg"]
["ab", "cd", "efg"]
Prelude> 1 : 2 : 3 : []
[1,2,3]
```

Konstruktor list

[1, 2, 3, 4, 5]
1 : [2, 3, 4, 5]
1 : 2 : [3, 4, 5]
1 : 2 : 3 : [4, 5]
1 : 2 : 3 : 4 : [5]
1 : 2 : 3 : 4 : 5 : []

```
Prelude> 1:[2,3]
[1,2,3]
Prelude> 1:[]
[1]
Prelude> 'a' : [1,2,3]

<interactive>:40:8:
    No instance for (Num Char) arising from the literal ‘1’
    In the expression: 1
    In the second argument of ‘(:)’, namely ‘[1, 2, 3]’
    In the expression: 'a' : [1, 2, 3]
Prelude> 'a':['b', 'c']
"abc"
```

Operator indeksowania

!!

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

Prelude> "abcd"  !!  2
'c'
```

Operator konkatenacji

++

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

Prelude>  "abcd"++"ef"
"abcdef"
```

Podstawowe operacje na listach

head

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

```
Prelude> head [1,2,3,4]
1
Prelude> head "Haskell"
'H'
```

---

tail

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

```
Prelude> tail [1,2,3,4]
[2,3,4]
Prelude> tail "Haskell"
"askell"
```

init

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

```
Prelude> init [1,2,3,4]
[1,2,3]
Prelude> init "haskell"
"haskel"
```

---

last

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

```
Prelude> last [1,2,3,4]
4
Prelude> last "haskell"
'l'
```

length

```
length :: [a] -> Integer
length [] = 0
length (x:xs) = 1 + length xs
```

```
Prelude> length [6,7,8,4,32,0]
6
Prelude> length ['a','c','v']
3
Prelude> length ["ala","ola"]
2
```

---

sum

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

```
Prelude> sum [1,2,3,4]
10
Prelude> sum [1..4]
10
Prelude> sum [1..100]
5050
```

take

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

```
Prelude> take 2 [1,2,3,4,5]
[1,2]
Prelude> take 3 ['a','b','c','d']
"abc"
Prelude> take 3 "abcd"
"abc"
```

---

drop

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

```
Prelude> drop 2 [1,2,3,4,5]
[3,4,5]
Prelude> drop (-1) [1,2,3]
[1,2,3]
Prelude> drop 0 [1,2,3]
[1,2,3]
Prelude> drop 3 ['a','b','c']
""
Prelude> drop 3 ['a','b','c','d']
"d"
```

elem

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

```
Prelude> elem 2 [3,5,2,3,1]
True
Prelude> elem 2 [1,3,5,7]
False
Prelude> elem 'a' "ala"
True
```

---

reverse

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

```
Prelude> reverse [1,2,3,4,5]
[5,4,3,2,1]
Prelude> reverse "lleksaH"
"Haskell"
```

Najmniejszy element listy

```
min :: Int -> Int -> Int
min x y | x <= y  = x
        | otherwise = y

mnm :: [Int] -> Int
mnm [] = error "empty list"
mnm [x] = x
mnm (x:xs) = min x (mnm xs)
```

Średnia elementów listy

```
srednia :: [Int] -> Float
srednia [] = error ”lista pusta”
srednia xs = fromInteger (sum xs) / fromInteger (length xs)
```

fromInteger - konweruje INTs do Floats

```
Prelude> :t (/)
(/) :: Fractional a => a -> a -> a
```

Sortowanie elementów listy

removeFst – usuwa pierwsze wystąpienie liczby *m* w liście
srtInts – sortuje elementy listy liczbowej rosnąco

```
removeFst :: Eq a => a -> [a] -> [a]
removeFst x [] = []
removeFst x (y:ys) | x == y = ys
                   | otherwise = y : (removeFst x ys)

srtInts :: [Int] -> [Int]
srtInts [] = []
srtInts xs = m : (srtInts (removeFst m xs)) where m = mnm xs
```

Quicksort

```
quicksort [] =[]
quiksort (x:xs) = quicksort(filter(<x)xs) ++
                  [x]++
                  quicksort(filter(>=x)xs)
```

- Wynikiem sortowania ciągu pustego jest ciąg pusty
- (x:xs) ciąg niepusty składa się z głowy x i ogona xs
- (filter(<x)xs) z ciągu xs wybierz elementy mniejsze od x
- (filter(>=x)xs) z ciągu xs wybierz elementy większe lub równe x
- ++ połącz ciągi
- Kolejność obliczeń nie jest określona

Funkcje wyższego rzędu

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

map

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : (map f xs)
```

```
Prelude> map sqrt [1,4,9]
[1.0,2.0,3.0]
Prelude> map reverse ["as","ola","las"]
["sa","alo","sal"]
Prelude> map fst [('a',3),('s',9)]
"as"
Prelude> map sum [[1,1],[2,2],[3,3]]
[2,4,6]
```

Funkcje wyższego rzędu

filter

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

```
Prelude> filter (>3) [2,-1,5,0,8,6]
[5,8,6]
Prelude> filter even [1,2,3,4,5,6]
[2,4,6]
```

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!