Programowanie funkcyjne
(wykład 3.)

Roman Dębski

Instytut Informatyki, AGH

2 listopada 2021

Plan wykładu

1. Funkcje wyższego rzędu
2. Wzorzec 'Collection Pipeline'

Plan wykładu

1. Funkcje wyższego rzędu
2. Wzorzec 'Collection Pipeline'

Reguła "DRY"

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

sumSqr :: Num a => [a] -> a
sumSqr []     = 0
sumSqr (x:xs) = x^2 + sumSqr xs
```

```
sumCub :: Num a => [a] -> a
sumCub []     = 0
sumCub (x:xs) = x^3 + sumCub xs

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

Baczność! :)

Pisanie kodu podobnego do powyższego grozi ... [a tego wolelibyśmy uniknąć]

Funkcje wyższego rzędu, funkcje jako argumenty

Funkcja wyższego rzędu
to funkcja, która zwraca i/lub przyjmuje jako argument(y) inne funkcje

Funkcje jako argumenty

```
sumWith :: Num a => (a -> a) -> [a] -> a
sumWith _ [] = 0
sumWith f (x:xs) = f x + sumWith f xs

sum, sumSqr, sumCub, sumAbs :: [Integer] -> Integer

sum    = sumWith (\e -> e)     -- sumWith (id)
sumSqr = sumWith (\e -> e ^ 2) -- sumWith (^2)
sumCub = sumWith (\e -> e ^ 3) -- sumWith (^3)
sumAbs = sumWith (\e -> abs e) -- sumWith (abs)
```

Funkcje anonimowe, wyrażenia λ (λ-wyrażenia)

```
ghci> (\x y -> x + y) 1 2
3
```


Funkcje jako wyniki

$f'(x_0, h) \approx \frac{f(x_0+h)-f(x_0-h)}{2h}$

```
df :: (Double -> Double) -> (Double -> Double)
df f = \x -> (f (x + h) - f (x - h)) / (2 * h)
  where h = 1e-8

absErr :: Num a => (t -> a) -> (t -> a) -> t -> a
absErr fExact fApprox x=abs (fExact x - fApprox x)
```

```
ghci> xs = [0,0.2..1.0]
ghci> zip xs (map (absErr (\x -> 2 * x) (df (\x -> x^2))) xs)
[(0.0,                0.0),
 (0.2,                2.105423613230073e-10),
 (0.4,                4.210847226460146e-10),
 (0.6000000000000001, 4.919487928134458e-9),
 (0.8000000000000002, 1.0260060578914931e-8),
 (1.0000000000000002, 6.603827262807726e-9)]
```

Z iloma argumentami wywołana jest funkcja *absErr*?

```
absErr (\x -> 2 * x) (df (\x -> x^2)) :: Double -> Double
```


Domknięcie funkcji (closure)

Domknięcie (funkcji), closure
funkcja + (jej) środowisko

$f'(x_0, h) \approx \frac{f(x_0+h)-f(x_0-h)}{2h}$

```
df :: (Double -> Double) -> Double -> (Double -> Double)
df f h = \x -> (f (x + h) - f (x - h)) / (2 * h)
```

```
ghci> dfSqrXDx = df (\x -> x^2) 1e-8
ghci> dfSqrXDx 1
1.9999999933961732
```


Operatory (.) (złożenie funkcji) i ($) (obliczenie wartości/'aplikacji' funkcji)

Złożenie w matematyce
$(f \circ g)(x) = f(g(x))$

... i w Haskellu

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

```
ghci> f x = 2 * x
ghci> g x = x ^ 2

ghci> f_o_g x = (f . g) x -- 2 * x^2
ghci> g_o_f = g . f       -- 4 * x^2
ghci> f_o_g 3 -- 18
ghci> g_o_f 3 -- 36
```

Uwaga: $ priorytet 0, prawostronna łączność

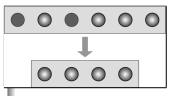
```
($) :: (a -> b) -> a -> b
f $ x = f x
```

```
ghci> f = \x y -> x^2 + y^2
ghci> f 1 2 + 3   -- 8
ghci> f 1 $ 2 + 3 -- 26
```

Uwaga
istnieje też operator $! (f $! x) – wersja 'strict' operatora $

Plan wykładu

Funkcje wyższego rzędu: *filter*



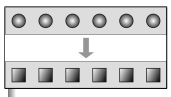
```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

```
ghci> filter (\x -> x < 5 && x > 1) [1..10] -- [2,3,4]
ghci> filter (<5) [1..10] -- [1,2,3,4]
ghci> filter even [1..10] -- [2,4,6,8,10]

ghci> filter (\s -> length s == 2) ["a", "aa", "aaa", "b", "bb"]
["aa","bb"]

ghci> filter (\(x,y) -> x > y) [(1,2), (2,2), (2,1), (2,2), (3,2)]
[(2,1),(3,2)]
```

filter and list comprehension

```
filter p xs = [x | x <- xs, p x]
```


Funkcje wyższego rzędu: *map*



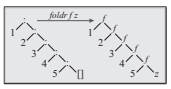
```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

```
ghci> map (\e -> e^2) [1..5] -- [1,4,9,16,25]
ghci> map (1+) [1..5] -- [2,3,4,5,6]
ghci> map ($ 2) [(^2), (^3), (^4), (^5)] -- [4,8,16,32]

ghci> map (\e -> (e, length e)) ["My","name","is","Inigo","Montoya"]
[("My",2),("name",4),("is",2),("Inigo",5),("Montoya",7)]
```

map and list comprehension

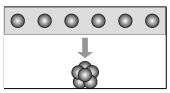
```
map f xs = [f x | x <- xs]
```


Funkcje wyższego rzędu: *foldr* [reduce]



Częsty schemat rekursji*

```
f []     = z
f (x:xs) = x `op` (f xs)
-- sum: z = 0, op = +
-- prod: z = 1, op = *
```

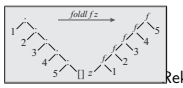


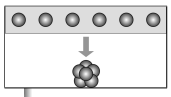
```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

```
ghci> foldr (+) 0 [1..5] -- 15
ghci> foldr (*) 1 [1..5] -- 120
ghci> foldr (||) False [False, False, True, False] -- True
ghci> foldr (&&) True [False, False, True, False] -- False
ghci> foldr (\_ n -> 1 + n) 0 [1..10] -- 10
ghci> foldr (\x xs -> xs ++ [x]) [] [1..5] -- [5,4,3,2,1]
```

* *foldr* ∼ (:) -> f, [] -> z

Funkcje wyższego rzędu: *foldl* [reduce]





Rekursja z akumulatorem" [z ∼ acc]

```
f :: Num t => t -> [t] -> t
f z []     = z -- z, op jak dla foldr
f z (x:xs) = f (x `op` z) xs
```

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

Funkcje pokrewne

```
foldl1 :: Foldable t => (a -> a -> a) -> t a -> a
foldr1 :: Foldable t => (a -> a -> a) -> t a -> a
foldl' :: Foldable t => (b -> a -> b) -> b -> t a -> b
```


Funkcje: *zip, unzip, zipWith*

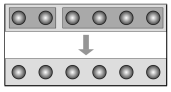
```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _ = []
zipWith _ _ [] = []
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
```

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

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

```
ghci> zip [1,2,3] ['a','b'] -- [(1,'a'),(2,'b')]
ghci> unzip [(1,'a'),(2,'b')] -- ([1,2],"ab")
ghci> unzip (zip [1,2,3] ['a','b']) -- ([1,2],"ab")
ghci> zipWith (+) [1..5] [5,4..1] -- [6,6,6,6,6]
```


Funkcja *concat* [flatten]



```
concat :: [[a]] -> [a]
concat = foldr (++) []
```

```
ghci> concat [[1,2],[3,4]]
[1,2,3,4]

ghci> (concat . concat) [ [[1,2], [3,4]] , [[5,6], [7,8]] ]
[1,2,3,4,5,6,7,8]
```


Wzorzec *'Collection pipeline'*

"*Collection pipelines* are a programming pattern where you organise some computation as a sequence of operations which compose by taking a collection as output of one operation and feeding it into the next. (Common operations are filter, map, and reduce). This pattern is common in functional programming, and also in object-oriented languages which have lambdas".

— Martin Fowler

```
capitalize :: [Char] -> [Char] -- Data.Char, Data.List needed
capitalize [] = []
capitalize (x:xs) = toUpper x : (map toLower xs)

formatStr s = foldr1 (\w s -> w ++ " " ++ s) .
              map capitalize .
              filter (\x -> length x > 1) $
              words s
```

```
ghci> formatStr "tomasz t bogdan anna Jerzy j maria"
"Tomasz Bogdan Anna Jerzy Maria"
```

Bibliografia

- Simon Thompson, Haskell: The Craft of Functional Programming, Addison-Wesley Professional, 2011
- Graham Hutton, Programming in Haskell, Cambridge University Press, 2007
- `https://wiki.haskell.org`
- `http://learnyouahaskell.com`