Programowanie funkcyjne
(wykład 2.)

Roman Dębski

Instytut Informatyki, AGH

7 października 2022

Plan wykładu

Plan wykładu

Definicja funkcji: funkcje wielu zmiennych

Nazewnictwo zmiennych w Haskellu

Nazwy zmiennych **nie mogą zaczynać się wielką literą** (te są zarezerwowane dla nazw typów).

```
mPi :: Fractional t => t
mPi = 3.141592653589793 -- stała/wartość
```

```
ghci> 2 + mPi
5.141592653589793
```

```
addT :: Num a => (a, a) -> a
addT (x,y) = x + y
```

```
ghci> addT (1,2)
3
```

```
addC :: Num a => a -> a -> a
addC x y = x + y
```

```
ghci> addC 1 2
3
```

```
curry :: ((a, b) -> c) -> a -> b -> c
uncurry :: (a -> b -> c) -> (a, b) -> c
```

```
ghci> (curry addT) 1 2 -- 3
ghci> (uncurry addC) (1,2) -- 3
```

Uwaga

```
f :: a -> b -> c -> d = a -> (b -> (c -> d)) -- prawostronna łączność
f 1 2.2 'c' = ((f 1) 2.2) 'c' -- lewostronna łączność
```


Currying*, partially applied functions**, sections (przykłady)

```
addC :: Num a =>
        a -> a -> a -> a
addC x y z = x + y + z
```

```
(^) :: (Num a, Integral b) =>
       a -> b -> a
flip :: (a -> b -> c) -> b -> a -> c
```

```
add1To_ :: Num a => a -> a -> a
-- partially applied function
add1To_ = addC 1

add1and2To_ :: Num a => a -> a
add1and2To_ = add1To_ 2
-- = addC 1 2

add1and2And3 :: Num a => a
add1and2And3 = add1and2To_ 3
-- = addC 1 2 3
```

```
ghci> (^) 2 5 -- = 2 ^ 5 = 32

ghci> twoToPower_ = (2^) -- = (^) 2
ghci> twoToPower_ 5 -- = 32

ghci> _ToPower5 = (^5) -- = flip (^) 5
ghci> _ToPower5 2 -- = 32
```

Uwaga

```
ghci> subtr5From = (-5) -- Num a => a
ghci> subtr5From = flip (-) 5 -- c -> c
ghci> subtr5From 6 -- = 1, Num c=>c->c
-- Prelude.subtract :: Num a => a->a->a
```

* from *Haskell Brooks Curry*

** partially applied functions vs. partial functions

Plan wykładu

Non-strictness* vs. laziness

Strictness (semantyka denotacyjna)

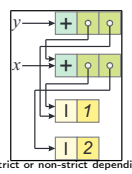
f is *strict* iff $f \perp = \perp$

otherwise *non-strict*, $\perp$ - *'bottom'*

```
ghci> f x = 42
ghci> f (1/0) -- 42 (f _|_ = 42)
ghci> f non_terminating_expr -- 42
```

Laziness (semantyka operacyjna)

Leniwe wartościowanie/obliczanie (lazy evaluation) - realizacja/implementacja 'non-strictness' wykorzystująca *'thunks'***




```
ghci> let x = 1 + 2 :: Int
ghci> :sp x -- x = _
ghci> let y = x + 1
ghci> :sp y -- y = _
ghci> seq x () -- ()
ghci> :sp x -- x = 3
ghci> :sp y -- y = _
```

*the language is strict or non-strict depending on whether functions are strict or non-strict by default

**thunks - unevaluated values with a recipe that explains how to evaluate them

Leniwe wartościowanie/obliczanie (przykłady)

```
ghci> import Data.Tuple
ghci> let x = 1 + 2 :: Int
ghci> let z = swap (x,x+1)
ghci> :sp z -- z = _
ghci> seq z () -- ()
ghci> :sp z -- z = (_,_)
ghci> seq x () -- ()
ghci> :sp z -- z = (_,3)
ghci> seq (fst z) () -- ()
ghci> :sp z -- z = (4,3)
```

```
ghci> let ys = [1..5] :: [Int]
ghci> :sp ys -- ys = _
ghci> seq (length ys) () -- ()
ghci> :sp ys -- ys = [1,2,3,4,5]

ghci> let xs = [1..] :: [Int]
ghci> :sp xs -- xs = _
ghci> head xs -- 1
ghci> :sp xs -- xs = 1 : _
ghci> xs !! 2 -- 3
ghci> :sp xs -- xs = 1 : 2 : 3 : _
```

Ciąg Fibonacciego

```
ghci> let fibs = 0 : 1 : zipWith (+) fibs (tail fibs) :: [Int]
ghci> :sp fibs -- fibs = _

ghci> take 6 fibs -- [0,1,1,2,3,5]
ghci> :sp fibs -- fibs = 0 : 1 : 1 : 2 : 3 : 5 : _
```

Plan wykładu

1. Definicja funkcji (uzupełnienie)
2. Leniwe obliczanie/wartościowanie
3. Rekursja i przetwarzanie list

Lista: podstawowe operacje [uwaga: niektóre z nich zgłaszają RT error, jeśli lista jest pusta]

Definicja listy

```
data [] a = [] | a : [a]
```

```
ghci>(:) 1 ((:) 2 []))=1:(2:[]))=[1,2]
-- [], (:) - konstruktory
```

```
ghci>xs = 1:2:3:4:[]--[1,2,3,4]
ghci>length xs -- 4
ghci>reverse xs -- [4,3,2,1]

ghci>head xs -- 1
ghci>tail xs -- [2,3,4]
ghci>last xs -- 4
ghci>init xs -- [1,2,3]

ghci>0 : xs -- [0,1,2,3,4]
ghci>xs ++ [5] -- [1,2,3,4,5]
ghci>xs !! 2 -- 3
ghci>[1,2] ++ [3,4] --[1,2,3,4]

ghci>take 2 xs -- [1,2]
ghci>drop 2 xs -- [3,4]
```

```
ghci>null xs -- False
ghci>any (>2) xs -- True
ghci>all (>0) xs -- True

ghci>zip xs ['a','b']--[(1,'a'),(2,'b')]
ghci>splitAt 2 xs --([1,2],[3,4])
ghci>sort [2,3,1,4] -- [1,2,3,4]
ghci>2 `elem` xs -- True

ghci>filter even xs -- [2,4]
ghci>map (*2) xs -- [2,4,6,8]
ghci>foldr (+) 0 xs -- 10

ghci>minimum xs -- 1
ghci>maximum xs -- 4
ghci>sum xs -- 10
```


List comprehensions*

Set-builder notation (set comprehension)

$xs = \{x^2 \mid x \in \{1...10\}, x^5 < 1025\}$

List comprehension

```
xs = [ x^2 | x <- [1..10], x^5 < 1025 ]
```

Generatory i ich zagnieżdżanie, guards

```
ghci> [(x,c) | x <- [1..3], c <- ['a','b']] -- x <- [1..3] - generator
[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]

ghci> [(x,y) | x <- [1..3], y <- [1..x]]
[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3)]

ghci> [(a,b,c) | a <- [1..10], b <- [a..10], c <- [b..10], a^2+b^2==c^2]
[(3,4,5),(6,8,10)]
```

Napisy jako listy znaków

```
ghci> lowers xs = length [x | x <- xs, isLower x]
ghci> lowers "Hello" -- 4
```

*połączenie *map* i *filter*

Rekursja

Rodzaje rekursji

końcowa/"niekońcowa"*, bezpośrednia/pośrednia, zagnieżdżona, liniowa/drzewiasta, korekursja, ...

Prosty schemat definicji rekurencyjnej [uwaga: rekursja vs. rekurencja]

```
recurFun x = if {- base case(s) ? -}
             then {- handle base case(s) ... -}
             else {- handle recurrence case(s)... recurFun (f x)... -}
```

```
fact :: Integer -> Integer
-- assert (n >= 0)
fact n = -- **
  -- the base case
  if n == 0 || n == 1 then 1
  else n * fact (n - 1)
```

```
fibb :: (Num a, Eq a) => a -> a
-- assert (n >= 0)
fibb n =
  -- the base case
  if n == 0 || n == 1 then n
  else fibb (n - 2) + fibb (n - 1)
```

```
notTerm :: Int -> Bool
notTerm x = not (notTerm x)
```

* uwaga: tail call optimisation in Haskell

** "Never hire a developer who computes the factorial using recursion":)

Przetwarzanie list, rekursja, dopasowanie wzorców (przykł.)

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

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

```
qSort :: Ord a => [a] -> [a]
qSort []     = []
qSort (x:xs) = qSort (leftPart xs) ++ [x] ++ qSort (rightPart xs)
  where
    leftPart  xs = [ y | y <- xs, y <= x ]
    rightPart xs = [ y | y <- xs, y > x  ]
```

Conditional evaluation with guards

```
fst2Eq :: Eq a => [a] -> Bool
fst2Eq (x : y : _) | x == y = True
fst2Eq _                    = False
```

```
prod' :: Num a => [a] -> a
prod' = loop 1 where -- point-free
 loop acc []     = acc --accumulator
 loop acc (x:xs) = loop (x * acc) xs
```

```
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}
```


Bibliografia

- Graham Hutton, Programming in Haskell, Cambridge University Press, 2007
- Simon Marlow, Parallel and Concurrent Programming in Haskell, O'Reilly Media, Inc, 2013