

Funkcie a funkcionály na ceste k Wholemeal (functional) programming

Peter Borovanský
I-18

<http://dai.fmph.uniba.sk/courses/FPRO/>



Čo je wholemeal (celozrnné)

Geraint Jones: Wholemeal programming means to think big:

- work with an entire list, rather than a sequence of elements
- develop a solution space, rather than an individual solution
- imagine a graph, rather than a single path.

Wholemeal programming je štýl rozmýšľania, programovania

... privedie vás k šľachtickým manierom vo funkcionálnom svete



Celozrnný programátor musí

poznať funkcie a najzákladnejšie funkcionály

- map/filter

- $\text{map } f \text{ } xs = \text{map } f [x_1, \dots, x_n] = [f x_1, \dots, f x_n] = [f x \mid x \leftarrow xs]$
- $\text{filter } f \text{ } xs = \text{filter } p [x_1, \dots, x_n] = [x \mid x \leftarrow xs, p x]$

- foldr/foldl

- $\text{foldr } f \text{ } z [x_1, \dots, x_n] = (f x_1 (f x_2 \dots (f x_n z) \dots))$
- $\text{foldl } f \text{ } z [x_1, \dots, x_n] = (\dots((f z x_1) x_2) \dots x_n)$

- scanr/scanl

- $\text{scanr } f \text{ } z [x_1, \dots, x_n] = \text{reverse } [z, (f x_n z), \dots, (f x_2 \dots (f x_n z) \dots), (f x_1 (f x_2 \dots (f x_n z) \dots))]$
- $\text{scanl } f \text{ } z [x_1, \dots, x_n] = [z, (f z x_1), ((f z x_1) x_2), \dots, (\dots((f z x_1) x_2) \dots x_n)]$

- iterate

- $\text{iterate } f \text{ } x = [x, (f x), ((f x) x), \dots, f^n x, \dots]$

- concat, ... a t.d'.



Extrémny príklad celozrnného

```
rozdelParneNeparne :: [Integer] -> ([Integer],[Integer])
rozdelParneNeparne [] = ([],[])
rozdelParneNeparne (x:xs) = (xp, x:xn) where (xp, xn) = rozdelNeparneParne xs
```

```
rozdelNeparneParne :: [Integer] -> ([Integer],[Integer])
rozdelNeparneParne [] = ([],[])
rozdelNeparneParne (x:xs) = (x:xp, xn) where (xp, xn) = rozdelParneNeparne xs
```

```
rozdielSuctu :: [Integer] -> Integer
rozdielSuctu xs = sum parneMiesta - sum neparneMiesta
  where (parneMiesta, neparneMiesta) = rozdelParneNeparne xs
```

Celozrnné riešenie:

```
rozdielSuctu = negate . foldr (-) 0
alebo len -foldr(-)0
```



Niektoré vaše riešenia

-- PeterK

rozdielSuctu xs = f xs 0

f [] s = s

f (x:xs) s = g xs (s-x)

g [] s = s

g (x:xs) s = f xs (s+x)

-- PeterP

rozdielSuctu [] = 0

rozdielSuctu [x] = -x

rozdielSuctu (x:xs) = -x - rozdielSuctu(xs)

-- DusanM

rozdielSuctu [] = 0

rozdielSuctu xs = (spocitaj xs 1) - (spocitaj xs 0)

spocitaj [] 0 = 0

spocitaj (x:[]) 0 = x

spocitaj (x:[]) 1 = 0

spocitaj (x:xs) n | n == 0 = (x + (spocitaj xs 1))
| otherwise = (spocitaj xs 0)

-- FilipJ

rozdielSuctu [] = 0

rozdielSuctu [x] = (0 - x)

rozdielSuctu (x:y:ys) =
(y - x) +
(rozdielSuctu ys)



Krok-po-kroku

- Krok 1 - zbierame párne a nepárne prvky do zoznamov

`rozdielSuctu" xs = (sum p) - (sum n)`

where `(p,n) = foldr (\x -> \a,b -> (b,x:a)) ([],[]) xs`

- Krok 2 - prečo nepočítať súčet už hneď

`rozdielSuctu''' xs = p - n`

where `(p,n) = foldr (\x -> \a,b -> (b,a+x)) (0,0) xs`

- Krok 3 – ušetrený where, zistíme, čo je uncurry

`rozdielSuctu'''' xs = uncurry (-) $ foldr (\x -> \a,b -> (b,a+x)) (0,0) xs`

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

`uncurry f (a,b) = f a b`

- Krok 4 – ušetrený explicitný argument

`rozdielSuctu''''' = uncurry (-) . foldr (\x -> \a,b -> (b,a+x)) (0,0)`



Celozrnné krok-po-kroku

(a na jednoduchých príkladoch)

Čo robí táto funkcia ?

`foo` :: `[Integer] -> Integer`

`foo []` = 0

`foo (x:xs)` | `odd x` = $(3 \cdot x + 1) + \text{foo } xs$

| `otherwise` = `foo xs`

Sčíta $3x+1$ pre každý prvok x vstupného zoznamu, ale len tie nepárne...

`foo' xs = sum [3*x+1 | x <- xs, odd x]` – toto je výrazný progres v čitateľnosti

`foo'' xs = sum (map (\x -> 3*x+1) (filter (odd) xs))` -- to isté len s filter/map

`foo''' xs = sum $ map (\x -> 3*x+1) $ filter (odd) xs` -- poznajúc operátor \$

`foo'''' = sum . map (\x -> 3*x+1) . filter (odd)` -- poznajúc kompozíciu .

`foo''''' = sum . map ((+1).(*3)) . filter (odd)` -- 2xpoznajúc kompozíciu

`foo'''''' = foldr (+) 0 . map ((+1).(*3)) . filter (odd)` -- extrémna verzia bez sum



Celozrnné krok-po-kroku

(a na príkladoch)

Čo robí táto funkcia ?

```
goo :: [Integer] -> Integer
```

```
goo [] = 1
```

```
goo (x:xs) | even x = (x-2) * goo xs  
           | otherwise = goo xs
```

Vynásobí všetky párne prvky vstupného zoznamu zmenšené o 2

```
goo' xs = product [ x-2 | x <- xs, even x] -- výrazný progres v čitateľnosti
```

```
goo'' = product . map (subtract 2) . filter (even)
```

```
goo''' = foldl (*) 1 . map (subtract 2) . filter (even) -- extrémna verzia bez product
```


Colatz

(a na príkladoch)

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod{2} \\ 3n + 1 & \text{if } n \equiv 1 \pmod{2} \end{cases}$$

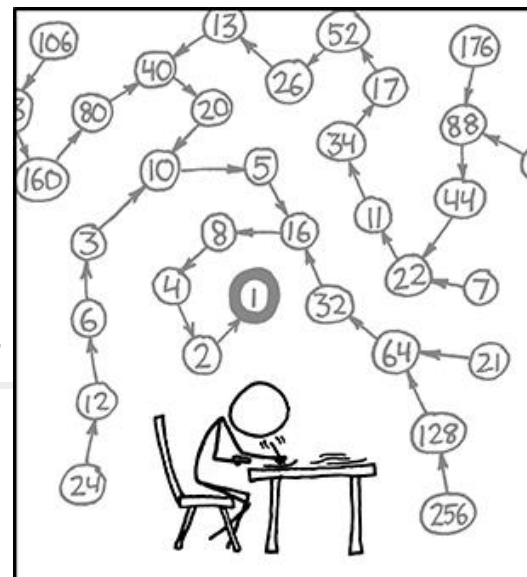
Čo robí táto funkcia ?

hoo :: Integer -> [Integer]

hoo 1 = []

hoo n | even n = n : hoo (n `div` 2)

| otherwise = n : hoo (3 * n + 1)



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

To sú prvky tzv. Colatzovej postupnosti

hoo' = takeWhile (/=1) . iterate (\x -> if even x then x `div` 2 else 3 * x + 1)

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

iterate f x = [x, f x, f f x, f f f x, ... , fⁿx, ...]

27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

Celozrnné krok-po-kroku

(a na príkladoch)

Čo robí táto funkcia ?

```
moo :: Integer -> Integer
```

```
moo 1 = 0
```

```
moo n | even n      = n + moo (n `div` 2)
```

```
      | otherwise   = moo (3 * n + 1)
```

súčet párnych prvkov Colatzovej postupnosti, teda `sum . filter (even) . hoo`

```
2+(
```

-- nezapočítali sme dvojicu (1, s+2)

```
  snd $
```

-- z poslednej dvojice zober druhú zložku

```
    last $
```

-- zober poslednú dvojicu

```
      takeWhile ((/=1).fst) $
```

-- kým prvá zložka dvojice <> 1

```
        iterate (\(x,s) -> if even x then (x `div` 2,x+s) else (3 * x + 1,s) )
```

(n,0)

```
)
```

```
moo" n = snd $ last $ takeWhile ((/=1).fst) $
```

-- z jemne zoprimalizované

```
  iterate (\(x,s) -> if even x then (x `div` 2,x+s) else (3 * x + 1,s) ) (n,2)
```

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod{2} \\ 3n + 1 & \text{if } n \equiv 1 \pmod{2}. \end{cases}$$

[Intro.hs](#)



Cifry

(niektoré vaše riešenia)

module Cifry where

`cifry 12345 == [1,2,3,4,5]`

`cifryR 12345 == [5,4,3,2,1]`

-- PeterK, ViktorN, PeterP, DusanM (veľmi podobne)

`cifry n = reverse (cifryR n)`

`cifryR 0 = []`

`cifryR n = (n `mod` 10):(cifryR (n `div` 10))`



Cifry

(Viktor z minulého roku po Erazme na CWI, Amsterdam)

module Cifry where

cifry 12345 == [1,2,3,4,5]

cifryR 12345 == [5,4,3,2,1]

cifry :: Integer -> [Integer]

cifry n = map(`mod` 10) \$ reverse \$

takeWhile (> 0) \$ iterate (`div` 10) n

iterate (`div` 10) 12345 == [12345,1234,123,12,1,0,0,0,0,0,0,0,0,0...]

[1,12,123,1234,12345]

[1, 2, 3, 4, 5]

cifry' = map(`mod` 10) . reverse . takeWhile (> 0) . iterate (`div` 10)

cifryR n = map(`mod` 10) \$ takeWhile (> 0) \$ iterate (`div` 10) n

cifryR' = map(`mod` 10) . takeWhile (> 0) . iterate (`div` 10)

Maximálny súčet súvislej podpostupnosti

0	1	2	3	4	5	6	7
-1	2	1	-3	2	3	-3	1

to

	0	1	2	3	4	5	6	7
0	-1	1	2	-1	-1	2	-1	0
1	x	2	3	0	2	5	2	3
2	x	x	1	-2	0	3	0	1
3	x	x	x	-3	-1	2	-1	0
4	x	x	x	x	2	5	2	3
5	x	x	x	x	x	3	0	1
6	x	x	x	x	x	x	-3	-2
7	x	x	x	x	x	x	x	1

from

$x_{\text{from}} + \dots + x_{\text{to}}$

Predošlý stĺpec

xs

následujúci stĺpec

`map(+x) xs ++ [x]`

nešikovné:

stĺpec otočíme

následujúci stĺpec

`x: map(+x) xs`

Stĺpce tejto tabuľky vyrábame postupne

`[[-1], [1, 2], [2, 3, 1], [-1, 0, -2, -3], [-1, 2, 0, -1, 2], [2, 5, 3, 2, 5, 2], [-1, 2, 0, -1, 2, 0, -3]]`



Pamäťová zložitosť $O(n^2)$ či $O(n^3)$

Maximálny súčet

```
maxSudet' :: [Int] -> Int
```

```
maxSudet' [] = 0
```

```
maxSudet' xs =
```

```
    maximum (map (maximum) -- maximum trojuholníkovej matice
```

```
        (init (                -- posledný prvok - trojuholníková
```

```
            foldl (\xss -> \x -> (x:(map (+x) (head xss))): xss) [[]] xs)))
```

```
maxSudet'' xs = maximum $ map (maximum) $
```

```
    init $ foldl (\xss -> \x -> (x:(map (+x) (head xss))): xss) [[]] xs
```

```
maxSudet''' = maximum . map (maximum) .
```

```
    init . foldl (\xss -> \x -> (x:(map (+x) (head xss))):xss) [[]]
```

```
maxSudet' [(-1), 2, 1, (-3), 2, 3, 1] == 6
```

```
maxSudet'' [(-1), 2, 1, (-3), 2, 3, 1] == 6
```

Kadane Algo

tempMax

globalMax

0	1	2	3	4	5	6	7
-1	2	1	-3	2	3	-3	1
0	2	3	0	2	5	2	3
0	2	3	3	3	5	5	5

```
kadane :: [Int] -> Int -> Int -> Int -- list -> tempMax -> globalMax -> max
kadane [] _ globalMax = globalMax
kadane (x:xs) tempMax globalMax = kadane xs newTempMax newGlobalMax
  where
```

```
    newTempMax = max (tempMax + x) 0
```

```
    newGlobalMax = max globalMax newTempMax
```

```
kadane' :: [Int] -> Int
```

```
kadane' (x:xs) = snd $ foldr pom (0,0) xs
```

```
  where pom x (tempMax, globalMax) =
```

```
    let newTempMax = max (tempMax + x) 0
```

```
    in (newTempMax, max globalMax newTempMax)
```

```
kadane [(-1), 2, 1, (-3), 2, 3, 1] 0 0 == 6
```

```
kadane' [(-1), 2, 1, (-3), 2, 3, 1] == 6
```



Pamäťová zložitosť $O(n)$



Maximálny súčet

(riešenie PeterP.)

```
maxSucet s = maxSucet' s [] 0 [] 0
```

```
maxSucet' :: [Int] -> [Int] -> Int -> [Int] -> Int -> (Int, [Int])
```

```
maxSucet' [] curMaxS curMaxSum _ _ = (curMaxSum, curMaxS)
```

```
maxSucet' (x:xs) curMaxS curMaxSum indexS indexSum
```

```
    | newIndexSum < 0 = maxSucet' xs curMaxS curMaxSum [] 0
```

```
    | otherwise      =
```

```
        maxSucet' xs newMaxS newMaxSum newIndexS newIndexSum
```

```
where
```

```
    newIndexSum = indexSum + x
```

```
    newIndexS = indexS ++ [x]
```

```
    newMaxSum = max newIndexSum curMaxSum
```

```
    newMaxS =
```

```
        if newMaxSum == newIndexSum then newIndexS else curMaxS
```




Najčastejšie vyskytujúce slovo

Nájdi najčastejšie vyskytujúce sa slovo v reťazci

-- rozdel' na slová podľa oddelovača, viac pozri [Data.List.Split](#)

splitOneOf :: String -> String -> [String]

splitWords = filter(/= "") . splitOneOf " .,:;!@#\$\$%^&*()'"

chunks :: [String] -> [[String]]

chunks [] = []

chunks xs@(w:_) = takeWhile (==w) xs: chunks (dropWhile (==w) xs)

"?: " **splitWords hamlet**
["There","was","this","king",

type FreqTable = [(Int,String)]

chunkLengths :: [[String]] -> FreqTable

chunkLengths xs = map (\chunk -> (length chunk, head chunk)) xs

"?: " **chunks ["a", "a", "a", "b", "b", "c"]**
[["a","a","a"],["b","b"],["c"]]

"?: " **chunkLengths \$ chunks ["a", "a", "a", "b", "b", "c"]**
[(3,"a"),(2,"b"),(1,"c")]

[MostFrequent.hs](#)



Najčastejšie vyskytujúce slovo

```
mostFrequent :: String -> String
```

```
mostFrequent ws =
```

```
    snd $ last $ sort $ chunkLengths $ chunks $ sort $ splitWords $ map toLower ws
```

```
    "?: " sort [(3,"d"),(1,"b"), (2,"a")]
           [(1,"b"),(2,"a"),(3,"d")]
```

```
-- funkcionálna verzia
```

```
mostFrequent' =
```

```
    snd .last . sort . chunkLengths . chunks . sort . splitWords . map toLower
```

```
    "?: " mostFrequent' hamlet
    "the"
```

```
-- zátvorková verzia pre rodených Lispistov
```

```
mostFrequent" ws =
```

```
    snd (
      last (
        sort (
          chunkLengths (
            chunks (
              sort (
                splitWords (
                  map toLower ws
                )
              )
            )
          )
        )
      )
    )
```

Vstupný text:

```
hamlet = "There was this king sitting in his garden all alane " ++
  "When his brother in his ear poured a wee bit of henbane. " ++
  "He stole his brother's crown and his money and his widow. " ++
  "But the dead king walked and got his son and said Hey listen, kiddo! " ...
```



Kartézsky súčin

- fuj ☺ riešenie

cart xss = sequence xss

- tradičné, a priznajme, dobre čitateľné riešenie:

cp :: [[t]] -> [[t]]

cp [] = [[]]

cp (xs:xss) = [(x:ys) | x <- xs, ys <- cp xss]

- Marianové riešenie

pridáme jeden prvok do každej množiny `cartTemp 1 [[4,5],[6,7]] == [[1,4,5],[1,6,7]]`

-- verzia 1

cartTemp :: t -> [[t]] -> [[t]]

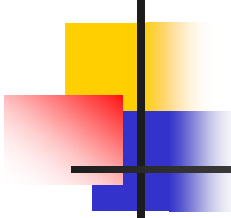
cartTemp element xss = foldr (\xs rekurgia -> (element:xs):rekurgia) [] xss

-- verzia 2

cartTemp element = foldr pom [] where

pom xs rek = (element:xs):rek

[cartesianMarian.hs](#)



riešenie – pokrač.

prvky jednej množiny kombinujeme s mnohými množinami

```
cartTemp2 [1, 2, 3] [[4, 5],[6,7],[8,9]] ==  
  [[1,4,5],[1,6,7],[1,8,9],[2,4,5],[2,6,7],[2,8,9],[3,4,5],[3,6,7],[3,8,9]]
```

```
cartTemp2' xs yss = concat [ cartTemp x yss | x<-xs]
```

```
cartTemp2      :: [t] -> [[t]] -> [[t]]      y++ (cartTemp x yss)  
cartTemp2 [] _ = []  
cartTemp2 xs yss = foldr (\x y -> (foldr (:) (cartTemp x yss) y)) [] xs
```

Kartézsky súčin množiny množín

```
--cart [ [1,2], [3,4], [5] ] = [[2,4,5],[2,3,5],[1,4,5],[1,3,5]]
```

```
cart      :: [[t]] -> [[t]]
```

```
cart xss = foldr (\x y -> cartTemp2 x y) [[]] xss
```



Kartézsky – transformácie

-- iníciaľne riešenie

```
cp_1 [] = [[]]
```

```
cp_1 (xs:xss) = [(x:ys) | x <- xs, ys <- cp_1 xss]
```

-- rozbité na vnútorný a vonkajší list-comprehension

```
cp_2 [] = [[]]
```

```
cp_2 (xs:xss) = concat [ [(x:ys) | ys <- cp_2 xss ] | x <- xs]
```

-- vnútorný list=comprehension prepíšeme cez map

```
cp_3 [] = [[]]
```

```
cp_3 (xs:xss) = concat [ map (x:) (cp_3 xss) | x <- xs]
```

-- zavedieme foldr

```
cp_4 xss = foldr pom [[]] xss where
```

```
    pom xs rek = concat [ map (x:) rek | x <- xs]
```



Kartézsky – transformácie

-- odstránime concat

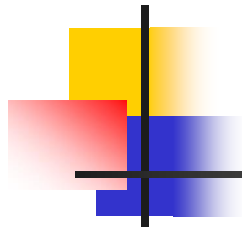
```
cp_5 xss = foldr pom [[]] xss where
    pom xs rek = foldr (\x -> \rek2 -> (map (x:) rek) ++ rek2) [] xs
```

-- slušnejšie prepísané

```
cp_6 xss = foldr pom [[]] xss where
    pom xs rek = foldr (pom2 rek) [] xs
    pom2 rek x rek2 = (map (x:) rek) ++ rek2
```

-- odstránime map

```
cp_7 xss = foldr pom [[]] xss where
    pom xs rek = foldr (pom2 rek) [] xs
    pom2 rek x rek2 = (foldr (pom3 x) [] rek) ++ rek2
    pom3 x y ys = (x:y):ys
```



Kartézsky – transformácie

```
-- odstránime append
cp_8 xss = foldr pom [[]] xss where
    pom xs rek = foldr (pom2 rek) [] xs
    pom2 rek x rek2 = foldr (:) rek2 (foldr (pom3 x) [] rek)
    pom3 x y ys = (x:y):ys
```

-- jediný problém, že to ide aj s tromi foldami

-- Strachey's functional pearl, forty years on

<https://spivey.oriel.ox.ac.uk/mike/firstpearl.pdf>