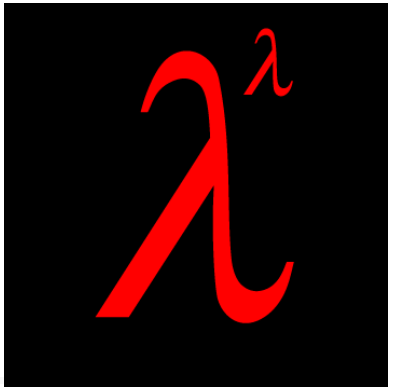
Funkcionálne programovanie 2



Peter Borovanský, KAI, I-18,
borovan(a)ii.fmph.uniba.sk

- lineárne patterny a pattern-matching
- množinová notácia (list comprehension)
- funkcionály (*funkcie vyšších rádov*)
  - map, filter, foldl, foldr, ...

Cvičenie:

- funkcionálny štýl (množinová notácia a map, ...)

Porovnávanie so vzorom
(pattern matching)

V hlavičke klauzule, či vo *where*/*let* výraze sa môže vystytnúť vzor typu: premenné sa nesmú opakovať (lineárny pattern/term)

- konštruktorový vzor, n-tica

```
reverse []     = []
reverse (a:x)  = reverse x ++ [a]
```

- n+k - vzor

```
ack 0 n          = n+1
ack (m+1) 0      = ack m 1
ack (m+1) (n+1) = ack m (ack (m+1) n)
```

- wildcards (anonymné premenné)

```
head (x:_)  = x
tail (_:xs) = xs
```

- @-vzor (aliasing)

```
zopakuj_prvy_prvok s@(x:xs) = x:s
```

@-aliasing
(záležitosť efektívnosti)

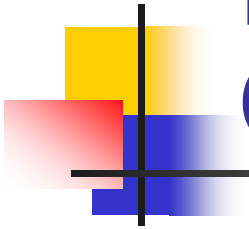
- definujte test, či zoznam [Int] je usporiadaným zoznamom:

```
-- prvé riešenie (ďalšie alternatívy, viď cvičenie):
usporiadany                         :: [Int] -> Bool
usporiadany []                      = True
usporiadany [_]                     = True
usporiadany (x:y:ys)        | x < y   = usporiadany (y:ys)
                            | otherwise= False
```

- @ alias použijeme vtedy, ak chceme mať prístup (hodnotu v premennej) k celému výrazu (xs), aj k jeho častiam (y:ys), bez toho, aby sme ho najprv deštruovali a následne hneď konštruovali (čo je neefektívne):

```
-- v tomto príklade xs = (y:ys)
usporiadany'''                      :: [Int] -> Bool
usporiadany''' []                   = True
usporiadany''' [_]                  = True
usporiadany''' (x:xs@(y:ys))        = x < y && usporiadany''' xs
```

List comprehension
(množinová notácia)



- pri písaní programov používame efektívnu konštrukciu, ktorá pripomína matematický množinový zápis.
- z programátorského hľadiska táto konštrukcia v sebe skrýva cyklus/rekurziu na jednej či viacerých úrovniach.

Príklad:

- zoznam druhých mocnín čísel z intervalu 1..100:

  [n*n | n <- [1..100]] { n*n | n ∈ { 1, …, 100 }}
- zoznam druhých mocnín párnych čísel z intervalu 1..100:

  [n*n | n <- [1..100], even n] { n*n | n ∈ { 1, …, 100} & 2|n}
- zoznam párnych čísel zoznamu:

  selectEven xs = [x | x<-xs , even x] { x | x ∈ xs & even x }

  Main> selectEven [1..10]
  [2,4,6,8,10]

List comprehension
(množinová notácia)

Syntax

[výraz **|** (generátor alebo test)* **]**

<generátor> ::= <pattern> **<-** <výraz typu zoznam (množina)>
<test> ::= <booleovský výraz>

- zoznam vlastných deliteľov čísla

```
factors n       = [ i | i <- [1..n-1], n `mod` i == 0 ]
```

Main> factors 24
[1,2,3,4,6,8,12,24]

- pythagorejské trojuholníky s obvodom <= n

```
pyth n = [ ( a, b, c ) |  a <- [1..n],
                          b <- [1..n],        -- určite aj efektívnejšie ...
                          c <- [1..n],
                          a + b + c <= n,
                          a^2 + b^2 == c^2 ]
```

Main> pyth 25
[(3,4,5),(4,3,5),(6,8,10),(8,6,10)]

Main> :type pyth
pyth :: (Num a, Enum a, Ord a) => a -> [(a,a,a)]

List comprehension (matice)

- malá násobilka:

  nasobilka = [(i, j, i*j) | i <- [1..10], j <- [1..10]]

  [(1,1,1),(1,2,2),(1,3,3), …] :: [(Int,Int,Int)]

  nasobilka' = [[(i,j,i*j) | j <- [1..10]] | i <- [1..10]]

  [[(1,1,1),(1,2,2),(1,3,3),…],
  [(2,1,2),(2,2,4),…….],
  [(3,1,3),…],
  …
  ] :: [[(Int,Int,Int)]]

type Riadok = [Int] – type definuje typové synonymum

type Matica = [Riadok]

- i-ty riadok jednotkovej matice

  riadok i n = [if i==j then 1 else 0 | j <- [1..n]]

  [1,0,0,0]

- jednotková matica

  jednotka n = [riadok i n | i <- [1..n]]

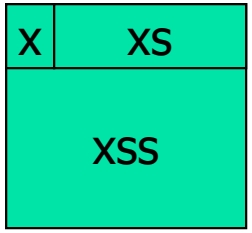
  [[1,0,0,0]
  [0,1,0,0]
  [0,0,1,0]
  [0,0,0,1]]

List comprehension (matice)

- sčítanie dvoch matíc – vivat Pascal ☺

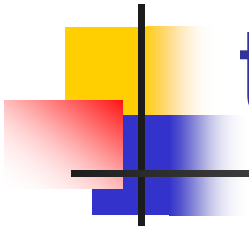
```
scitajMatice        :: Matica -> Matica -> Matica
scitajMatice m n    =
        [ [(m!!i)!!j + (n!!i)!!j | j <- [0..length(m!!0)-1] ]
                                 | i <- [0..length m-1] ]
```

- transponuj maticu pozdĺž hlavnej diagonály




```
transpose                   ::  Matica  ->  Matica
transpose []                =  []
transpose ([] : xss)        =  transpose xss
transpose ((x:xs) : xss)    =  (x : [h | (h:t) <- xss]) :
                                    transpose (xs : [t | (h:t) <- xss])
```

m1 = [[1,2,3],[4,5,6],[7,8,9]]
m2 = [[1,0,0],[0,1,0],[0,0,1]]
m3 = [[1,1,1],[1,1,1],[1,1,1]]
scitajMatice m2 m3 = [[2,1,1],[1,2,1],[1,1,2]]
transpose m1 = [[1,4,7],[2,5,8],[3,6,9]]



transpose v Pythone

```
def head(xs):
  return xs[0]

def tail(xs):
  return xs[1:]

def transpose(xss):
  if xss == []:
     return []
  else:
     if head(xss) == []:
        return transpose(tail(xss))
     else:
        return [([head(head(xss))] + [ head(row) for row in tail(xss)])]
               +
               transpose([tail(head(xss))]+[ tail(row) for row in tail(xss)])

print(transpose([[1,2,3],[4,5,6],[7,8,9]]))
```

List comprehension
(permutácie-kombinácie)

- vytvorte zoznam všetkých 2^n n-prvkových kombinácií {0,1}

```
pre n=2, kombinácie 0 a 1 sú: [[0,0],[1,0],[1,1],[0,1]]
kombinacie 0            = [[]]
kombinacie n            = [ 0:k | k <- kombinacie (n-1)] ++
                          [ 1:k | k <- kombinacie (n-1)]
```

- vytvorte permutácie prvkov zoznamu

```
perms []        = [[]]
perms x         = [ a:y | a <- x, y <- perms (diff x [a]) ]
-- rozdieľ zoznamov x y (tie, čo patria do x a nepatria do y), alebo x\\y
diff x y        = [ z | z <- x, notElem z y]
```

```
Main> :type perms
perms :: Eq a => [a] -> [[a]]
Main> :type diff
diff :: Eq a => [a] -> [a] -> [a]
```

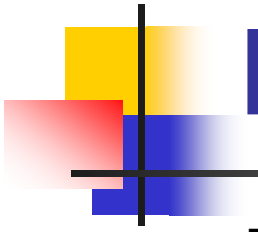
```
Main> perms [1,2,3]
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

List comprehension (quicksort)

- quicksort

```
qs              :: [Int] -> [Int]
qs []           = []
qs (a:as)       = qs [x | x <- as, x <= a]
                        ++
                        [a]
                        ++
                  qs [x | x <- as, x > a]
```

Main> qs [4,2,3,4,6,4,5,3,2,1,2,8]
[1,2,2,2,3,3,4,4,4,5,6,8]



Polymorfizmus

čítajte Prelude.hs

- obsahuje veľa použiteľných definícií
- tak programujú majstri ...

Parametrický polymorfizmus znamená, že jedným predpisom definujeme typ (funkcie, dát) pre nekonečne veľa inštancií typového parametra.

- zreťazenie dvoch zoznamov

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

- vyber prvých n prvkov zo zoznamu

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

QuickSort

Niekedy však potrebujeme vymedziť vlastnosti typového parametra tak, aby spĺňal isté predpoklady, napr. aby dosadený typ a bolo možné:

- porovnávať na rovnosť (==),
- nerovnosť (<, >),
- či mal definované isté funkcie (napr. show)...

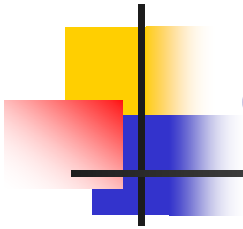
Príklad:

predpoklady kladené na typ a

```
quickSort          :: (Ord a) => [a] -> [a]

quickSort []       = []
quickSort (b:bs)   = quickSort [x | x <- bs, x <= b]
                            ++
                            [b]
                            ++
                     quickSort [x | x <- bs, x > b]
```

- spĺňal isté predpoklady sa chápe tak, aby patril do istej triedy implementujúcej požadované funkcie (napr. ==, /=, …)

Enumerovany dátový typ
(jednoduchá verzia union-type/record-case)



Enum typ zodpovedá lineárnemu usporiadaniu, Int(eger), Bool, Char, Float

data Color = Red | Green | Blue

data Bool = True | False

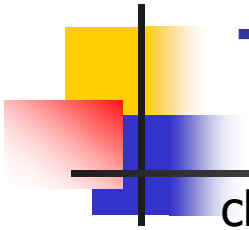
data Znamka = A | B | C | D | E | Fx
deriving (Eq, Show, Ord)

deriving(…) zamená, že požadované funkcie vymenovaných tried sú definované implicitne, syntakticky

Pozná funkcie succ, pred, succ E == Fx, a pred Fx == E

Triedy:

- Eq - umožní porovnávať na == (/=) hodnoty tohoto typu
- Show - umožní vypísať hodnotu tohoto typu
- Ord - umožní porovnávať <, > hodnoty tohoto typu



Typová trieda Eq

```
class Eq a where                    -- trieda (Eq a) definuje
    (==) :: a->a->Bool              -- rovnosť pre a,
    (/=)  :: a->a->Bool             -- nerovnosť pre a
                                    -- interface

instance Eq Znamka where            -- štandardná implementácia
    A == A          = True          -- ak použijem
    B == B          = True          -- data Znamka =...deriving (Eq)

    C == C          = True
    ...
    Fx == Fx        = True
    _  == _         = False         -- implementation
```

Typová trieda Ord

```
Ord (je podtrieda Eq):                  -- triedy majú svoju hierarchiu
                                        -- ak chceme niečo porovnávať na
                                        -- <, <=, … musíme vedieť ==
class Eq a => Ord a where               -- Ord a pozná == a /=
    (<)  :: a->a->Bool                  -- a okrem toho aj <, >, …
    (>)  :: a->a->Bool
    (<=) :: a->a->Bool
    (>=) :: a->a->Bool
    min  :: a->a->Bool
    max  :: a->a->Bool

instance Ord Znamka where               -- defaultná implementácia
    A < B           = True              -- data Znamka =…deriving (Ord)
    B < C           = True
    …
```

Binárny strom - (trieda Show)

Konštruktory vždy s veľkým začiatočným písmenom

```
data TreeInt      = Vrchol TreeInt TreeInt | List Int
                                        -- strom s hodnotami typu Int v listoch
konštanta         Vrchol (List 5) (Vrchol (List 6) (List 9)) :: TreeInt

class Show a where                      -- analógia __str__ z pythonu
    show :: a->String                   -- analógia toString() z javy

instance Show TreeInt where             -- vlastná implementácia show pre TreeInt
    show (List i)              = show i
    show (Vrchol left right) = "(" ++ show left ++ "," ++ show right ++ ")"
```

Main> Vrchol (List 5) (Vrchol (List 6) (List 9))
(5,(6,9))

- Príklad: maximálna hodnota v listoch binárneho stromu typu TreeInt

```
maxTree                 :: TreeInt -> Int
maxTree (List x)        = x
maxTree (Vrchol l r)    = max (maxTree l) (maxTree r)
```

Polymorfický binárny strom


```
data BTree a                = Branch (BTree a) (BTree a) | Leaf a
                              deriving (Show)
konštanta                   -- strom s hodnotami typu a v listoch stromu
Main> Branch (Leaf 5) (Branch (Leaf 6) (Leaf 9))        -- defaultný show
Branch (Leaf 5) (Branch (Leaf 6) (Leaf 9)) :: BTree Int
```

ak nepoužijeme deriving(show) ani neinštancujeme Show pre BTree, dostaneme:

```
Main> Branch (Leaf 5) (Branch (Leaf 6) (Leaf 9))
ERROR - Cannot find "show" function for:
```

- Príklad rekurzívnej funkcie prechádzajúcej BTree a:
  spoštenie stromu do zoznamu hodnôt v listoch(rekurzia na strome):

```
flat                        :: BTree a -> [a]
flat (Leaf x)               = [x]
flat (Branch left right)    = (flat left) ++ (flat right)
```

Binárny vyhľadávací strom

```
-- binárny strom s hodnotami typu t vo vnútorných vrcholoch
data BVS t          = Nod (BVS t) t (BVS t) | Nil        -- konštruktory Nod, Nil
                      deriving (Show)

konštanta:          Nod (Nod Nil 1 Nil) 3 (Nod Nil 5 Nil) :: BVS Int

--spĺňa x::BVS t podmienku na binárny vyhľadávací strom ?
jeBVS                           :: BVS Int -> Bool
jeBVS Nil                       = True              -- neefektívne ale priamočiare...
jeBVS (Nod left value right)=(maxBVS left)<=value && value<=(minBVS right)&&
                                jeBVS left && jeBVS right

--zisti maximálnu/minimálnu hodnotu v nejakom strome
maxBVS :: BVS Int -> Int
maxBVS Nil                      = minBound::Int      --  -∞ = 2147483648
maxBVS (Nod left value right) = max (maxBVS left) (max value (maxBVS right))
```

BVS – find a insert

```
--vyhľadávanie v binárnom vyhľadávacom strome:
findBVS  :: (Ord t) => t -> BVS t -> Bool
findBVS  _ Nil                                  = False
findBVS  x (Nod left value right)    | x == value    = True
                                     | x < value     = findBVS x left
                                     | otherwise     = findBVS x right

--vsunutie prvku do binárneho vyhľadávacieho stromu:
insertBVS :: (Ord t) => t -> BVS t -> BVS t
insertBVS x Nil     = Nod Nil x Nil
insertBVS x bvs@(Nod left value right)
                    | x == value     = bvs
                    | x < value      = Nod (insertBVS x left) value right
                    | otherwise      = Nod left value (insertBVS x right)

--odstránenie bude na domácu úlohu...
```

Funkcia (predikát) argumentom

učíme sa z Prelude.hs:

- zober zo zoznamu tie prvky, ktoré spĺňajú bool-podmienku (test)
  Booleovská podmienka príde ako argument funkcie a má typ (a -> Bool):

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

> filter even [1..10]
[2,4,6,8,10]

- ako by to napísal lenivý haskellista:
  tripivot (x:xs) = (filter (<x) xs, filter (==x) xs, filter (>x) xs)

Funkcia (predikát) argumentom

učíme sa z Prelude.hs:

- ber zo zoznamu prvky, kým platí podmienka (test):

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

> takeWhile (>0) [1,2,-1,3,4]
[1,2]

- vyhoď tie počiatočné prvky zoznamu, pre ktoré platí podmienka:

```
dropWhile                       :: (a -> Bool) -> [a] -> [a]
dropWhile p []                  = []
dropWhile p xs@(x:xs')  | p x = dropWhile p xs'
                        | otherwise = xs
```

> dropWhile (>0) [1,2,-1,3,4]
[-1,3,4]

Príklad
(porozdeľuj)

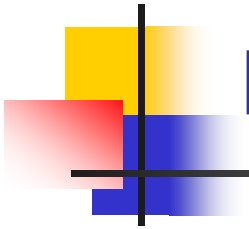
porozdeluj :: (a -> Bool) -> [a] -> [[a]] rozdelí zoznam na podzoznamy, v ktorých súvisle platí podmienka daná 1. argumentom

porozdeluj (>0) [1,2,0,3,4,5,-1,6,7] = [[1,2],[3,4,5],[6,7]]

porozdeluj (\x -> x `mod` 3 > 0) [1..10] = [[1,2],[4,5],[7,8],[10]].

```
porozdeluj p [] = []
porozdeluj p xs =
      takeWhile p xs :
          porozdeluj p
              (dropWhile (\x -> (not (p x)))
                  (dropWhile p xs))
```

```
Main> porozdeluj (>0) [1,2,0,0,3,4,-1,5]
[[1,2],[3,4],[5]]
```



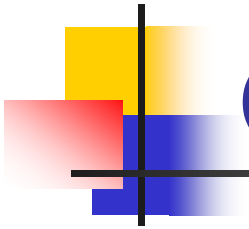
map

- funktor, ktorý aplikuje funkciu (1.argument) na všetky prvy zoznamu

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

- Príklady:

```
map (+1) [1,2,3,4,5]                    = [2,3,4,5,6]
map odd  [1,2,3,4,5]                    = [True,False,True,False,True]
and (map odd  [1,2,3,4,5])              = False

map head [ [1,0,0], [2,1,0], [3,0,1] ]  = [1, 2, 3]
map tail [ [1,0,0], [2,1,0], [3,0,1] ]  = [ [0,0], [1,0], [0,1] ]
map (0:) [[1],[2],[3]]                  = [[0,1],[0,2],[0,3]]
```

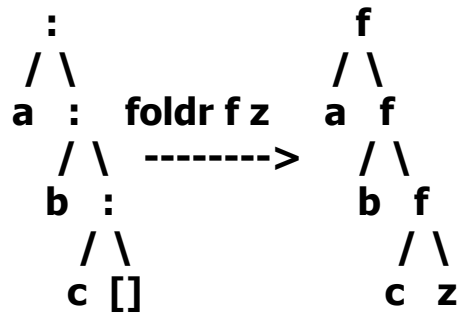


Transponuj maticu
(ešte raz)

```
transponuj                  :: Matica -> Matica

transponuj []               = []
transponuj  ([]:xss)        = transponuj xss
transponuj ((x:xs):xss)     = (x:(map head xss)):
                                    (transponuj (xs:(map tail xss)))

transpose                   :: [[a]] -> [[a]]
transpose []                = []
transpose ([] : xss)        = transpose xss
transpose ((x:xs) : xss)    = (x : [h | (h:t) <- xss]) :
                                    transpose (xs : [t | (h:t) <- xss])
```

Haskell – foldr

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

a : b : c : [] -> f a (f b (f c z))
```



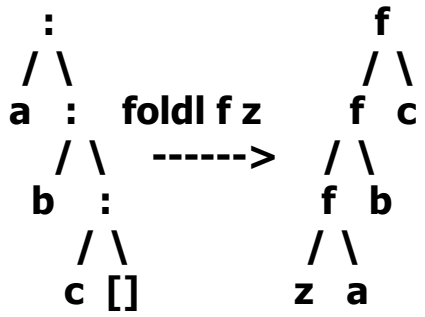

```
Main> foldr (+) 0 [1..100]
5050
```

```
Main> foldr (\x y->10*y+x) 0 [1,2,3,4]
4321
```

```
-- g je vnorená lokálna funkcia

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

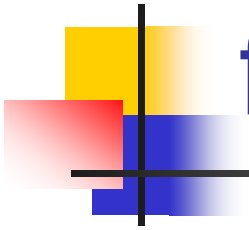
Haskell – foldl

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

a : b : c : [] -> f (f (f z a) b) c
```




```
Main> foldl (+) 0 [1..100]
5050

Main> foldl (\x y->10*x+y) 0 [1,2,3,4]
1234
```

foldr a foldl ešte raz



- foldl (+) 0 [1,2,3,4]
  foldr (+) 0 [1,2,3,4]

  10

- foldr max (-999) [1,2,3,4]
  foldl max (-999) [1,2,3,4]

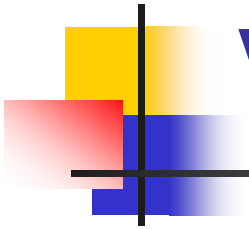
  4

- foldr (_ -> \y ->(y+1)) 0 [3,2,1,2,4]
  foldl (\x -> _ ->(x+1)) 0 [3,2,1,2,4]

  5

- rozpoj :: [(a,b)] -> ([a],[b])
  rozpoj = foldr (\(x,y) -> \(xs,ys) -> (x:xs, y:ys)) ([], [])

  rozpoj [(1,11),(2,22),(3,33)]
  ([1,2,3],[11,22,33])



Vypočítajte

- foldr max (-999) [1,2,3,4]
  foldl max (-999) [1,2,3,4]

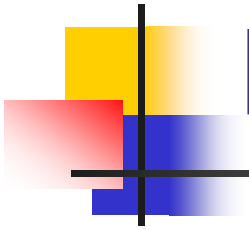
- foldr (_ -> \y ->(y+1)) 0 [3,2,1,2,4]
  foldl (\x -> _ ->(x+1)) 0 [3,2,1,2,4]

- foldr (-) 0 [1..100] =

(1-(2-(3-(4-...-(100-0))))) = 1-2 + 3-4 + 5-6 + ... + (99-100) = -50

- foldl (-) 0 [1..100] =

  (...(((0-1)-2)-3) ... - 100) = -5050



Funkcia je hodnotou

- [a->a] je zoznam funkcií typu a->a
  napríklad: [(+1),(+2),(*3)] je [\x->x+1,\x->x+2,\x->x*3]

- čo je foldr (.) id [(+1),(+2),(*3)] ??
  akého je typu
  foldr (.) id [(+1),(+2),(*3)] 100
  foldl (.) id [(+1),(+2),(*3)] 100

[a->a]

303

???

lebo skladanie fcií je asociatívne:

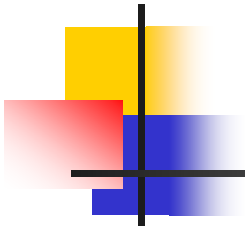
- ((f . g) . h) x = (f . g) (h x) = f (g (h x)) = f ((g . h) x) = (f . (g . h)) x

- mapf :: [a->b] -> [a] -> [b]
  mapf [] _ = []
  mapf _ [] = []
  mapf (f:fs) (x:xs) = (f x):mapf fs xs

-- rovnako dlhé zoznamy

[11,22,33]

mapf [(+1),(+2),(+3)] [10,20,30]



Kvíz

```
foldr (:) [] xs =     xs

foldr (:) ys xs =     xs++ys
```

```
foldr ? ? xs = reverse xs
```

Priemerný prvok

Ak chceme vypočítať aritmetický priemer (a-priemer) prvkov zoznamu, matice, ... potrebujeme poznať ich súčet a počet. Ako to urobíme na jeden prechod štruktúrou pomocou foldr/foldl ? ...počítame dvojicu hodnôt, súčet a počet:

- priemerný prvok zoznamu

```
priemer  xs = sum/count where (sum, count) = sumCount xs
sumCount xs = foldr (\x -> \(sum, count) -> (sum+x, count+1)) (0, 0) xs
```

- priemerný prvok matice

  je a-priemer a-priemerov riadkov matice a-priemerom hodnôt matice ?

```
sumCount'          :: [[Float]] -> (Float,Float)
sumCount' xs =
    foldr (\x -> \(sum, count)->scitaj (sumCount x) (sum, count)) (0, 0) xs
    where scitaj (a,b) (c,d) = (a+c, b+d)

priemer' :: [[Float]] -> Float
priemer' = uncurry (/) . sumCount'
```

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