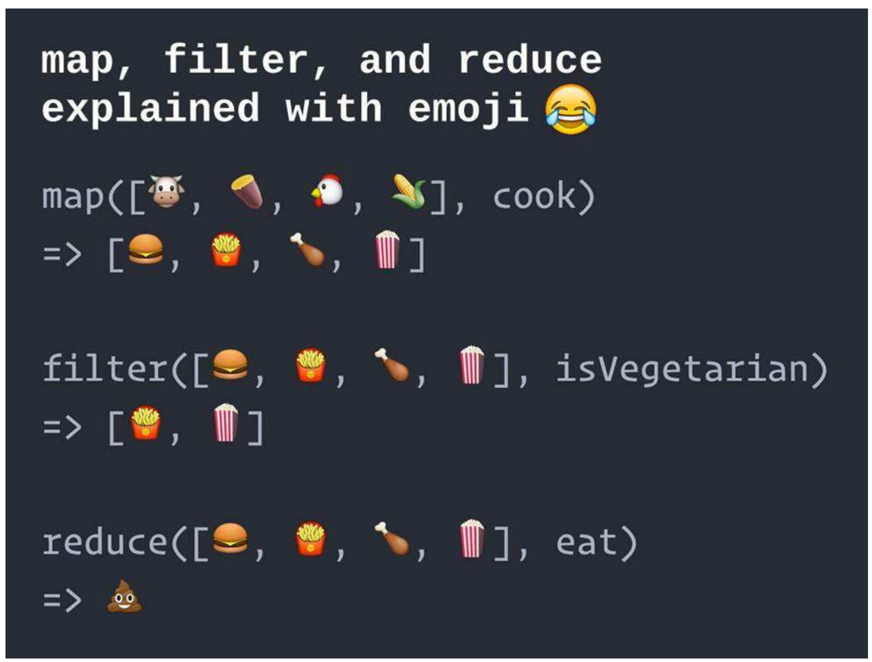
Haskell
A Purely Functional Language
featuring static typing, higher-order functions, polymorphism, type classes and monadic effects

Funkcie a funkcionály

dokončenie minulej prednášky

Peter Borovanský



Kvíz - platí/neplatí ?

(neseriózny prístup ale intuíciu treba tiež trénovať)

- length [m..n] == n-m+1

  "?: " quickCheck ((\(n,m) -> length [m..n] == n-m+1))

  *** Failed! Falsifiable (after 3 tests and 1 shrink):

  "?: " quickCheck ((\(n,m) -> **m <= n ==>** length [m..n] == n-m+1))

  +++ OK, passed 100 tests.

- length (xs ++ ys) == length xs + length ys

  "?: " quickCheck((\xs->\ys->(length (xs++ys)==length xs + length ys)))

  +++ OK, passed 100 tests.

- length (reverse xs) == length xs

  quickCheck((\xs -> (length (reverse xs) == length xs)))

  +++ OK, passed 100 tests.

- (xs, ys) == unzip (zip xs ys)

  quickCheck((\xs -> \ys -> ((xs, ys) == unzip (zip xs ys))))

  *** Failed! Falsifiable (after 3 tests and 1 shrink):

  quickCheck((\xs -> \ys -> (length xs == length ys ==>
  (xs, ys) == unzip (zip xs ys))))

QuickCheck.hs

Počet cifier ešte raz

funkcionálny štýl

```
pocetCifier :: Integer -> Int
pocetCifier n = length $ show n
                    pocetCifier = length . show

pocetCifier' :: Integer -> Int
pocetCifier' n =  fromIntegral $ ceiling $ (logBase 10 (fromIntegral n))
                    pocetCifier' =  fromIntegral . ceiling . (logBase 10) . fromIntegral

pocetCifier'' :: Integer -> Int
pocetCifier'' n =  length $ takeWhile (/=0) $ iterate (`div` 10) n
                    pocetCifier'' =  length . takeWhile (/=0) . iterate (`div` 10)

hypoteza1 = quickCheck(\n -> (n > 0) ==> pocetCifier n == pocetCifier'' n)
hypoteza2 = quickCheck(\n -> (n > 0) ==> pocetCifier n == pocetCifier' n)
hypoteza2' = quickCheck(\n -> (n > 1) ==> pocetCifier n == pocetCifier' n)
hypoteza2'' = quickCheck(\n -> (n > 10) ==> pocetCifier n == pocetCifier' n)
-- platí/neplatí ?
```

cifry.hs

Funkcia/predikát argumentom

- zober zo zoznamu tie prvky, ktoré spĺňajú 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]

alternatívna definícia:

```
filter p []     = []
filter p (x:xs) = if p x then x:(filter p xs) else filter p xs
```

vlastnosti (zväčša úplne zrejmé ?):

- filter True xs = xs ... [x | x <- xs, True] = [x | x <- xs] = xs
- filter False xs = [] ... [x | x <- xs, False] = [ ]
- filter p1 (filter p2 xs) = filter (p1 && p2) xs
- (filter p1 xs) ++ (filter p2 xs) = filter (p1 || p2) xs

```
filter p []         = []
filter p (x:xs)     = if p x then x:(filter p xs) else filter p xs
```

Dôkaz

filter p1 (filter p2 xs) = filter (p1 && p2) xs

Indukcia vzhľadom na parameter xs

- []

  L.S. = filter p1 (filter p2 []) = filter p1 [] = [] = filter (p1 && p2) [] = P.S.

- (x:xs)

  L.S. = filter p1 (filter p2 (x:xs)) = ... definícia

  filter p1 (if p2 x then x:(filter p2 xs) else filter p2 xs) = ... filter dnu cez if

  if p2 x then filter p1 (x:(filter p2 xs)) else filter p1 (filter p2 xs) = ... indukcia

  if p2 x then filter p1 (x:(filter p2 xs)) else filter (p1 && p2) xs = ... definícia

  if p2 x then

          if p1 x then x:(filter p1 (filter p2 xs)) else filter p1 (filter p2 xs)

  else filter (p1 && p2) xs = ... 2 x indukcia

  if p2 x then

          if p1 x then x:(filter (p1 && p2) xs) else filter (p1 && p2) xs

  else filter (p1 && p2) xs =

QuickCheck.hs

filter p [] = []
filter p (x:xs) = if p x then x:(filter p xs) else filter p xs

Dôkaz

filter p1 (filter p2 xs) = filter (p1 && p2) xs

if p2 x then

if p1 x then x:(filter (p1 && p2) xs) else filter (p1 && p2) xs

else filter (p1 && p2) xs = ... požívame vlastnosť if-then-else

if A then	if A && B then C
if B then C	else D
else D	
else D	

if (p1 && p2) x then x:(filter (p1 && p2) xs) else filter (p1 && p2) xs = ... def.

filter (p1 && p2) (x:xs) = P.S.

č.b.t.d.

QuickCheck a funkcie

Funkcie sú hodnoty ako každé iné

Ako vie QuickCheck pracovať s funkciami ?

- je skladanie funkcií komutatívne ?

```
"?: " import Text.Show.Functions    ☹
"?: " quickCheck(
    (\x -> \f -> \g -> (f.g) x == (g.f) x)::Int->(Int->Int)->(Int->Int)->Bool)
*** Failed! Falsifiable (after 2 tests):
```

- je skladanie funkcií asociatívne ?

```
"?: " quickCheck(
    (\x -> \f -> \g -> \h -> (f.(g.h)) x == ((f.g).h) x)    ☺
          ::Int->(Int->Int)->(Int->Int)->(Int->Int)->Bool)
+++ OK, passed 100 tests.
```

Opäť to NIE je DÔKAZ, len 100 pokusov.

QuickCheck.hs

QuickCheck a predikáty

Predikát je len funkcia s výsledným typom Bool

- filter p1 (filter p2 xs) = filter (p1 && p2) xs ☹

?: " quickCheck ((\xs -> \p1 -> \p2 ->

filter p1 (filter p2 xs) == filter (p1 && p2) xs)

:: [Int] -> (Int->Bool) -> (Int->Bool) -> Bool)

<interactive>:113:91: Couldn't match expected type 'Bool' ---

NEPLATÍ LEBO ANI TYPY NESEDIA, && je definovaný na Bool, a nie na funkciách Int->Bool

- filter p1 (filter p2 xs) = filter (\x-> p1 x && p2 x) xs ☺

  +++ OK, passed 100 tests.

Opäť to NIE je DÔKAZ (ten už bol), len 100 pokusov.

- (filter p1 xs) ++ (filter p2 xs) = filter (\x -> p1 x || p2 x) xs

"?: " quickCheck ((\xs -> \p1 -> \p2 ->

(filter p1 xs) ++ (filter p2 xs) == filter (\x -> p1 x || p2 x) xs)

:: [Int] -> (Int->Bool) -> (Int->Bool) -> Bool)

*** Failed! Falsifiable (after 3 tests):

[0] <function> <function>

QuickCheck.hs

QuickCheck a predikáty

Predikát je len funkcia s výsledným typom Bool

- filter p1 (filter p2 xs) = filter (p1 && p2) xs

?: " quickCheck ((\xs -> \p1 -> \p2 ->

filter p1 (filter p2 xs) == filter (p1 && p2) xs)

:: [Int] -> (Int->Bool) -> (Int->Bool) -> Bool)

<interactive>:113:91: Couldn't match expected type 'Bool' ---

NEPLATÍ LEBO ANI TYPY NESEDIA, && je definovaný na Bool, a nie na funkciách Int->Bool

- filter p1 (filter p2 xs) = filter (\x-> p1 x && p2 x) xs

  +++ OK, passed 100 tests.

Kontrapríklad

```
ghci> filter (\x -> 10 `div` x < 10) (filter (\x -> x > 1) [0, 1, 2])
[2]
ghci> filter (\x-> (10 `div` x < 10) && x > 1) [0, 1, 2]
*** Exception: divide by zero
ghci>
```

QuickCheck.hs

Rekapitulácia

videli sme tzv. **Property Based Testing** pomocou **QuickCheck**:

- najznámejšie dva funkcionály: map, filter – ktoré poznáte aj z Pythonu
- quickCheck náhodne generujúci testy/kontrapríklady pre typy
  - základné typy: Int, Bool, String…
  - zoznamy: [Int], [t]
  - funkcie: Int->Int, a->b, …
- množstvo 'ekvivalentných' tvrdení, niektoré boli neekvivalentné...

Property Based Testing (PBT):

- rôzne implementácie QuickCheck v jazykoch:
  - Scala (Scala Check), F# (FsCheck), Clojure (test.check), Python (Hypothesis)
- musí implementovať:
  - generovanie dát pre základné typy, parametrické typy, funkčné typy, ...
  - generovanie dát pre používateľom definované typy
  - zjednodušovanie kontrapríkladu (shrinking)

https://en.wikipedia.org/wiki/QuickCheck

Vlastnosti map

- map id xs = xs ☑ map id = id
- map (f.g) xs = map f (map g xs) ☑ map f . map g = map (f.g)
- ~~head (map f xs) = f (head xs)~~ ☑ ~~head . map f = f . head~~
- ~~tail (map f xs) = map f (tail xs)~~ ☑ ~~tail . map f = map f . tail~~
- map f (xs++ys) = map f xs++map f ys ☑
- length (map f xs) = length xs ☑ length . map f = length
- map f (reverse xs) = reverse (map f xs) ☑ map f.reverse=reverse.map f
- ~~sort (map f xs) = map f (sort xs)~~ ☑ ~~sort . map f = map f . sort~~
- map f (concat xss) = concat (map (map f) xss) ☑

map f . concat = concat . map (map f)

```
concat               :: [[a]] -> [a]
concat []            = []
concat (xs:xss)      = xs ++ concat xss
concat [[1], [2,3], [4,5,6],[]] = [1,2,3,4,5,6]
```

Vlastnosti map, filter
(superpozícia map a filter)

Na zamyslenie:

- filter p (map f xs) = ??? (filter (p.f) xs)
- filter p (map f xs) = map f (filter (p.f) xs)
- filter p . map f = map f . filter (p.f)

Dôkaz:

definícia map filter pomocou list-comprehension nám to objasní

```
filter p (map f xs)     = filter p [ f x | x<-xs]
                        = [y | y <- [ f x | x<-xs], p y]
                        = ... tu treba rozmýšľať ...
                        = [f x |  x<-xs, p (f x)]
                        = map f [x |  x<-xs, p (f x)]
                        = map f (filter (p.f))
```

Quíz - prémia

nájdite pravdivé a zdôvodnite

Pred tým, ako sa snažíte o dôkaz, vyskúšajte si, či QCH nenájde kontrapríklad

- map f . take n = take n . map f

```
quickCheck(\f -> \n -> \xs -> (map f . take n) xs == (take n . map f) xs)
+++ OK, passed 100 tests.
 platia, vid QuickCheck.hs
```

- map f . filter p = map fst . filter snd . map (fork (f,p))

  where fork :: (a->b, a->c) -> a -> (b,c)

  fork (f,g) x = (f x, g x)

- filter (p . g) = map (inverzna_g) . filter p . map g

  ak inverzna_g . g = id

- reverse . concat = concat . reverse . map reverse

- filter p . concat = concat . map (filter p)

QuickSort s QuickCheck
(na cvičeniach)

```
import Test.QuickCheck
import Data.List (sort)
qsort :: Ord a => [a] -> [a]              -- Ord a – vieme triediť len porovnateľné typy
qsort []       = []                       -- analógia interface Comparable<a>
qsort (p:xs) = qsort (filter  (< p) xs) ++ [p] ++ qsort (filter (>= p) xs)
```

~~quickCheck(\xs -> length (qsort xs) == length xs)~~

```
quickCheck((\xs -> length (qsort xs) == length xs)::[Int]->Bool)
quickCheck((\xs -> qsort xs == sort xs)::[Int]->Bool)
quickCheck((\xs -> qsort(qsort xs) == qsort xs)::[Int]->Bool)

isSorted :: Ord a => [a] -> Bool
isSorted xs = sort xs == xs
isSorted' :: Ord a => [a] -> Bool
isSorted' [] = True
isSorted' xs = and $ zipWith (<=) (init xs) (tail xs)
quickCheck((\xs -> isSorted (qsort xs))::[Int]->Bool)
quickCheck((\xs -> isSorted' (qsort xs))::[Int]->Bool)
```

QuickSort.hs

Kombinatorika
(podobné nájdete v Prémii QC & Kombinatorika)

```
module Kombinatorika where
import Test.QuickCheck
import Data.List

fact n = product [1..n]
comb n k = (fact n) `div` ((fact k) * (fact (n-k)))

-- permutácie
perms :: [t] -> [[t]]
perms [] = [[]]
perms (x:xs) = [ insertInto x i ys | ys <- perms xs, i <- [0..length ys] ]
                 where insertInto x i xs = (take i xs) ++ (x:drop i xs)
qchPERM = quickCheck(\n -> (n > 0 && n < 10) ==> length (perms [1..n]) == fact n)
```

```
kbo  ::              [t] -> Int -> [[t]]          n!
kso  ::              [t] -> Int -> [[t]]     ?    (n nad k)
vbo  :: (Eq t) =>    [t] -> Int -> [[t]]          ((n+k-1) nad k)
vso  ::              [t] -> Int -> [[t]]          n.(n-1). … .(n-k+1)
```

Kombinatorika.hs

Definované typy - QuickCheck

Ak definujeme vlastnú dátovú štruktúru, ako využiť quickCheck ?

data BVS t = Nil | Node (BVS t) t (BVS t) deriving(Show, Eq)

- dva konštruktory **Nil** a **Node** _ _ _
- deriving popisuje patričnosť do triedy class - (resp. implements interface)
  - Show – automaticky vygenerovaná funkcia show :: BVS t ->String
  - Eq – automaticky vygenerované funkcie ==,/= :: BVS t -> BVS t -> Bool

Ako definovať funkciu, ktorá vracia náhodný strom, napr. BVS Int ?

Existuje nejaká náhodná funkcia, napr. nextInt :: Int ?

Nie je to v rozpore s Referenčnou transparentnosťou ?

Java a Reflexivita
(malá odbočka – prémia Random class)

Skúsme si tú istú otázku preformulovať v Jave, ktorú poznáme

- Napíšte funkciu, ktorá vytvorí náhodnú inštanciu ľubovoľnej triedy

  Object gener(String className)

- Nechceme mať náhodný generátor pre každú triedu, lebo pre nami definované triedy by sme ho museli písať sami...

- Reflexivita (Java Reflection Model)
- https://github.com/Programovanie4/Prednasky/blob/master/13/13_java.pdf
- java primitívne typy (int, char, double, ...), String...
- polia (int[], ...)
- triedy s default konštruktorom (Stvorec(), ...)
- triedy s konštruktorom s parametrami – rekurzívne pre každý parameter konštruktora, potom zavolanie konštruktora s náhodnými parametrami
- generické triedy

QuickCheck – Generátor
(pre základné typy)

- trieda Arbitrary t definuje generátor Gen t pre hodnoty typu t:

class **Arbitrary** a where

    arbitrary :: Gen t

a volá sa pomocou funkcie generate :: Gen t -> IO t

IO je tzv. IO monáda, je to built-in hack pre vstupno-výstupné, side-effects

Pre preddefinované typy to už niekto zadefinoval:

```
"?: " (generate arbitrary) :: IO Int            23, 45, 12, 49,  12, …
"?: " generate arbitrary :: IO Char             't'"'w', '\199', …
"?: " generate arbitrary :: IO (Char, Int)      ('6',0), ('<',-7)
"?: " generate arbitrary :: IO [Int]            [-29,-17,10], [-10,9]
"?: " generate arbitrary :: IO Double           -5.5026813
"?: " generate arbitrary :: IO Bool             True, False, False
"?: " do { fst <- generate arbitrary::IO Int;
           snd <- generate arbitrary::IO Char;
           return (fst, snd) }                  (-6, 'r'), (15, 'a'), …
```

Generatory.hs

QuickCheck – Generátor
(pre funkčné typy)

```
"?: " generate arbitrary :: IO (Int->Int)                          <function>
"?: " do {f<-generate arbitrary :: IO (Integer->Integer); return (f 7)}    9, 11

"?: " do {
        f<-generate arbitrary :: IO (Integer->Integer);
        g<-generate arbitrary :: IO (Integer->Integer);
        x<-generate arbitrary :: IO Integer;
        return (((f.g) x) == ((g.f) x)) }             False, False, False, True

"?: " do {
        f<-generate arbitrary :: IO (Integer->Integer);
        g<-generate arbitrary :: IO (Integer->Integer);
        h<-generate arbitrary :: IO (Integer->Integer);
        x<-generate arbitrary :: IO Integer;
        return ((((f.g).h) x) == (((f.g).h) x)) }     True, True, True, True
```

Generatory.hs

Generátory
(pre definované typy)

```
kocka :: Gen Int
kocka = choose(1,6)                  -- "?: " generate kocka
                                     -- "?: " generate (choose(1,10))

yesno :: Gen Bool
yesno = choose(True, False)          -- "?: " generate yesno
                                 -- "?: " generate (choose(True, False))

data  Minca = Hlava | Panna deriving (Show)
instance Arbitrary Minca where
   arbitrary = oneof [return Hlava, return Panna]
                               "?: " generate (arbitrary::Gen Minca)
                               "?: " (generate arbitrary)::IO Minca

falosnaMinca :: Gen Minca
falosnaMinca = frequency [(1,return Hlava), (2,return Panna)]
                                     -- "?: " generate falosnaMinca
```

Pre nami definované typy XXX musíme definovať inštanciu triedy Arbitrary XXX

Generatory.hs

Generátory - zoznam

```
arbitraryListMax8Len :: Arbitrary a => Gen [a]    -- náhodný zoznam len <= 8
arbitraryListMax8Len =
    do {
      k <- choose (0, 8)::(Gen Int);
      sequence [ arbitrary | _ <- [1..k] ] }
```

"?: " generate (arbitraryListMax8Len::Gen [Int])
[-21,12,17,16,4,-20]

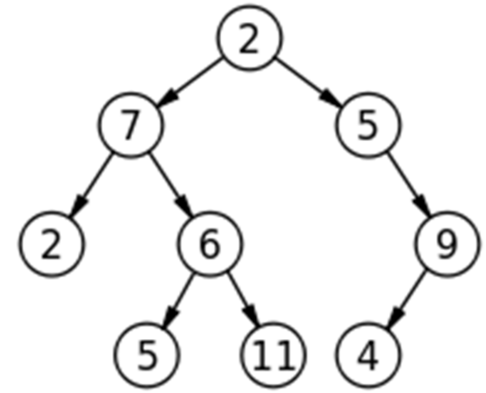
```
arbitraryList :: Arbitrary a => Gen [a]
arbitraryList =
  mysized ( \n -> do {
                  k <- choose (0, n) ;
                  sequence [ arbitrary | _ <- [1..k] ] }
          )
mysized :: (Int -> Gen a) -> Gen a
mysized f = f 50
```

"?: " generate (arbitraryList::Gen [Int])
[-9,7,14,24,18,28,-4,0,22,12,-14]

"?: " generate
(mysized (\n -> choose(n,n)))
50

Generatory.hs

Generátory - strom



```
data Tree t = Leaf t | Node (Tree t) t (Tree t)
      deriving (Show, Ord, Eq)

instance Arbitrary a => Arbitrary (Tree a) where
  arbitrary = frequency
              [
                (1, liftM Leaf arbitrary )
              , (1, liftM3 Node arbitrary arbitrary arbitrary)
              ]
```

"?: " generate arbitrary :: IO (Tree Int)
Node (Leaf (-7)) (-5) (Leaf 9)
"?: " generate (arbitrary :: Gen (Tree Int))
Leaf (-18)

```
strom :: Gen (Tree Int)
strom = frequency [
                (1, liftM Leaf arbitrary )
              , (10, liftM3 Node arbitrary arbitrary arbitrary)
              ]
```

"?: " generate strom
Node (Node (Leaf (-2)) 3 (Leaf (-6))) 23 (Leaf 22)

Generatory.hs

BVS – binárny vyhľadávací

```
data BVS t = Nil | Node (BVS t) t (BVS t)  deriving(Show, Ord, Eq)

-- je binárny vyhľadávací strom
isBVS                :: (Ord t) => BVS t -> Bool          -- t vieme porovnávať <

-- nájdi v binárnom vyhľadávacom strome
find                 :: (Ord t) => t -> (BVS t) -> Bool -- analógia Comparable<t>
find _ Nil                             = False
find x (Node left value right)         | x == value = True
                                       | x < value   = find x right
                                       | x > value   = find x left

flat                          :: BVS t -> [t]
flat Nil                      = []
flat (Node left value right)  = flat left ++ [value] ++ flat right
```

Tree.hs

BVS - isBVS

Príšerne neefektívne riešenie, prepíšte lepšie:

```
isBVS     :: (Ord t) => BVS t -> Bool
isBVS Nil = True
isBVS (Node left value right) =
      (all (<value) (flat left))
      &&
      (all (>value) (flat right))
      &&
      isBVS left
      &&
      isBVS right
```

Tree.hs

BVS - testy

```
qch1 = verbose((\x -> \tree -> find x tree)::Int->(BVS Int)->Bool)
qch2 = quickCheck((\x -> \tree -> ((find x tree) == (elem x (flat tree))))
                                             ::Int->BVS Int->Bool)
{--
"?: " qch2
*** Failed! Falsifiable (after 3 tests):
1 ; Node Nil (-2) (Node Nil 1 Nil)
--}
qch3 = quickCheck((\x -> \tree -> (isBVS tree) ==>
           ((find x tree) == (elem x (flat tree)))):::Int->BVS Int->Property)
{--
*** Failed! Falsifiable (after 2 tests):
0 ; Node (Node Nil (-1) (Node Nil 0 Nil)) 1 Nil
--}
```

KDE je chyba v definícii BVS ??

Tree.hs

Don't write tests!

Generate them from properties

BVS – tajnička

```
find  :: (Ord t) => t -> (BVS t) -> Bool
find _ Nil = False
find x (Node left value right)  | x == value = True
                                | x < value = find x right
                                | x > value = find x left

                                | x < value = find x left
                                | x > value = find x right
```

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.

first

- **solve a more general problem**,

then

- extract the interesting bits and pieces by transforming the general program into more specialized ones

Wholemeal programming je štýl rozmýšlania, programovania

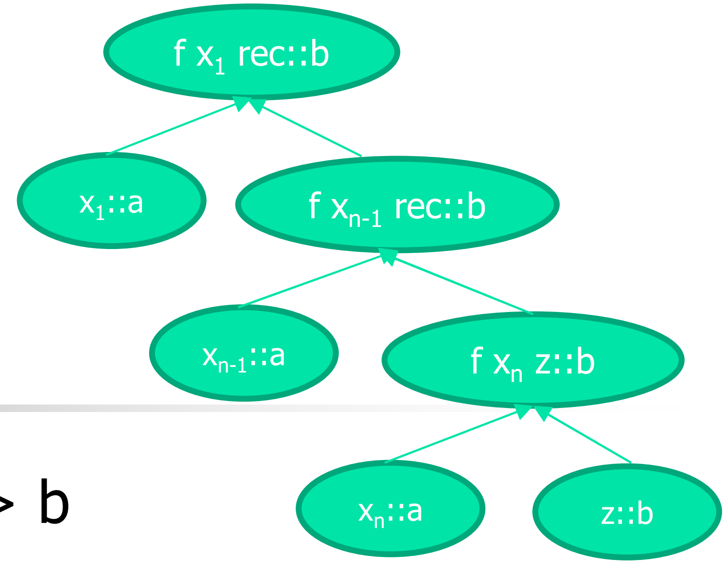
... privedie vás k *šlachtickým* manierom vo funkcionálnom svete

Celozrnný programátor musí

poznať funkcie a najzákladnejšie funkcionály

- map/filter
  - map f xs = map f [x_1, …, x_n] = [f x_1, …, f x_n] = [f x | x <- xs]
  - filter f xs = filter p [x_1, …, x_n] = [x | x <- xs, p x]
- foldr/foldl
  - foldr f z [x_1, …, x_n] = (f x_1 (f x_2 … (f x_n z)..))
  - foldl f z [x_1, …, x_n] = (..((f z x_1) x_2) … x_n)
- scanr/scanl
  - scanr f z [x_1, …, x_n] = reverse [z, (f x_n z), …, (f x_2…(f x_n z)..), (f x_1 (f x_2…(f x_n z)..))]
  - scanl f z [x_1, …, x_n] = [z, (f z x_1), ((f z x_1) x_2), …, (..((f z x_1) x_2) … x_n)]
  - scanr1 f [x_1, …, x_n] = reverse [x_n, (f x_{n-1} x_n), …, (f x_1 (f x_2 …(f x_{n-1} x_n)..))]
  - scanl1 f [x_1, …, x_n] = [x_1, (f x_1 x_2), ((f x_1 x_2) x_3), …, (..((f x_1 x_2) x_3) … x_n)]
- iterate
  - iterate f x = [x, (f x), ((f x) x), …, f^n x, …]
- concat, … a t.ď.

Haskell – foldr



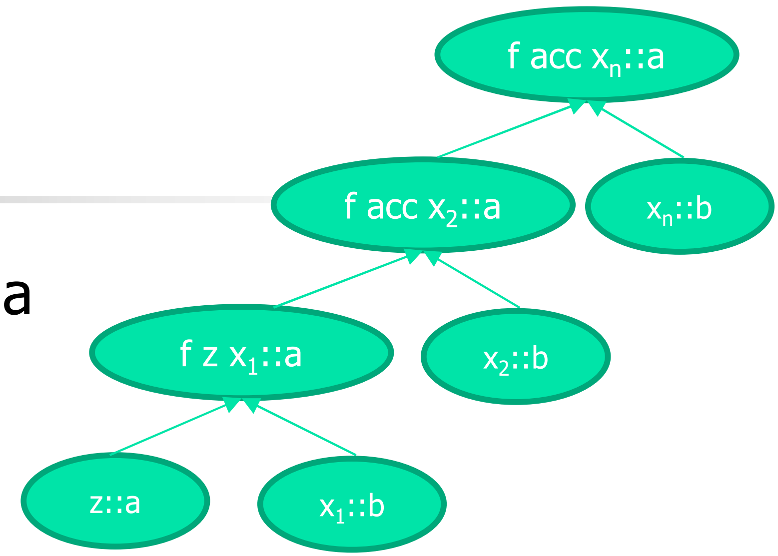

```
foldr           :: (a -> b -> b) -> b -> [a] -> b

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))
```

```
      :                    f
     / \                  / \
    a   :    foldr f z   a   f
       / \   -------->      / \
      b   :                b   f
         / \                  / \
        c  []                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
```

```
  :                      f
 / \                    / \
a   :    foldl f z     f   c
   / \    ------>     / \
  b   :              f   b
     / \            / \
    c  []          z   a
```

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

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

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

Kvíz

```
  :                         f
 / \                       / \
a   :    foldl f z        f   c
   / \   ------>         / \
  b   :                 f   b
     / \               / \
    c  []             z   a
```

foldr (:) [] xs =   xs

foldr (:) ys xs =   xs++ys

```
  :                       f
 / \                     / \
a   :    foldr f z      a   f
   / \   -------->         / \
  b   :                   b   f
     / \                     / \
    c  []                   c   z
```

foldr ? ? xs = reverse xs

foldr ((:) . h) [] = ???

Pre tých, čo zvládli kvíz, odmena !

kliknite si podľa vašej politickej orientácie

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 — [a->a]
  foldr (.) id [(+1),(+2),(*3)] 100 — 303
  foldl (.) id [(+1),(+2),(*3)] 100 — ???

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

- funkcie nevieme porovnávať, napr. head [(+1),(+2),(*3)] == id

- funkcie vieme permutovať, length $ permutations [(+1),(+2),(*3),(^2)]

fold.hs

Maximálna permutácia funkcií

- zoznam funkcií aplikujeme na zoznam argumentov

```
apply          :: [a -> b] -> [a] -> [b]
apply fs args = [ f a | f <- fs, a <- args]
```

```
apply [(+1),(+2),(*3)] [100, 200]
[101,201,102,202,300,600]
```

Dokážte/vyvraťte: map f . apply fs = apply (map (f.) fs)

- čo počíta tento výraz

```
maximum $
   apply
     (map (foldr (.) id) (permutations [(+1),(^2),(*3),(+2),(/3)]))
     [100]
```

31827

- ((+1).(+2).(*3).(^2).(/3)) 100

3336.333333333334

- ((/3).(^2).(*3).(+2).(+1)) 100

31827.0

fold.hs

take pomocou foldr/foldl

Výsledkom foldr ?f? ?z? xs je funkcia, do ktorej keď dosadíme n, vráti take n:

… preto aj ?z? musí byť funkcia, do ktorej keď dosadíme n, vráti take n []:

```
take'        :: Int -> [a] -> [a]
take' n xs  =  (foldr pomfcia (\_ -> []) xs) n where
                    pomfcia x h = \n -> if n == 0 then []
                                                   else x:(h (n-1))
                    alebo
                    pomfcia x h n =  if n == 0 then [] else x:(h (n-1))
                    alebo
take''' n xs = foldr (\a -> \h -> \n -> case n of
                                               0 -> []
                                               n -> a:(h (n-1)) )
                     (\_ -> [])
                    xs
                    n
```

foldoviny.hs

Extrémny príklad celozrnného

2 vzájomne rekurzívne fcie

```
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

RozdielSuctu.hs

Krok-po-kroku

(len pre tých, čo to nepochopili ešte)

- **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)
```

RozdielSuctu.hs

Celozrnné krok-po-kroku
(a na jednoduchých príkladoch)

Čo robí táto funkcia ?

```
foo                          :: [Integer] -> Integer
foo []                        = 0
foo (x:xs)    | odd x         = (3*x + 1) + 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
```

Intro.hs

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

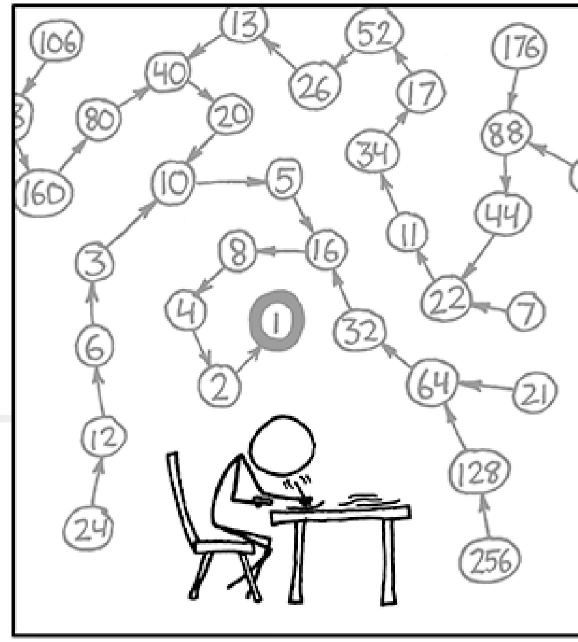
```
goo'  xs  = product [ x-2 | x <- xs, even x] -- výrazný progres v čitateľnosti

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

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

Intro.hs

Collatz

(a na príkladoch)

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



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.

Čo robí táto funkcia ?

```
collatz    :: Integer -> [Integer]
collatz 1  = []
collatz n | even n          = n : collatz (n `div` 2)
          | otherwise       = n : collatz (3 * n + 1)
```

To sú prvky tzv. Colatzovej postupnosti

```
collatz'= 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, … , f^nx, …]

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**

Intro.hs

Celozrnné krok-po-kroku

(a na príkladoch)

Čo robí táto funkcia ?

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

Intro.hs

```
moo       :: Integer -> Integer
moo 1     = 0
moo n     | even n              = n + moo (n `div` 2)
          | otherwise           = moo (3 * n + 1)
```

súčet párnych prvkov Collatzovej 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)
```

Cifry
(niektoré vaše riešenia)

```
module Cifry where
cifry 12345 == [1,2,3,4,5]
cifryR 12345 == [5,4,3,2,1]

cifry n = reverse (cifryR n)
cifryR 0 = []
cifryR n = (n `mod` 10):(cifryR (n `div` 10))
```

Cifry

```
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)
```

Intro.hs

Kritérium delietel’nosti 11

- rodné číslo 786115 3333 (ženské, *15.nov1978)
- 7861153333 `mod` 11 == 0
- 11 | 7861153333 iff 11 | 7+6+1+3+3 –(8+1+5+3+3) = 0

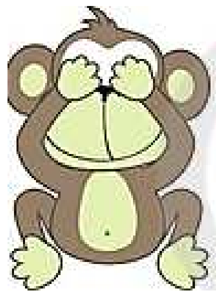
- naše rodné čísla sú delitelné 11, ľahká kontrola

- čísla kariet majú tiež kontrolu, Luhnnov algo, DÚ1

- čo bankové účty
- 7000155733 / 8180 – soc.poisťovňa
- cifry násobíme váhami 6,3,7,9,10,5,8,4,2,1, sčítame, výsledok delitelný 11
- 11 | 7*6+0*3+0*7+0*9+1*10+5*5+5*8+7*4+3*2+3*1
- (sum $ zipWith (*) [7,0,0,0,1,5,5,7,3,3] [6,3,7,9,10,5,8,4,2,1]) `mod` 11
- (sum $ zipWith (*) [2,7,0,1,1,3,2,4,4,3] [6,3,7,9,10,5,8,4,2,1]) `mod` 11

Binárne číslo $\{1\}^+\{0\}^*$

- $\underbrace{111...111}_{m}\underbrace{00.....0000}_{n} = (2^m-1) * 2^n$



```
null $                                   True
   dropWhile (==1) $                     []
      dropWhile(==0) $                   [1, 1]
         map (`mod` 2) $                 [0, 0, 1, 1]
            takeWhile (>0) $             [12, 6, 3, 1]
               iterate (`div` 2)         12 = [12,6,3,1,0,0,0...]
```

suma + i*cenaPiva = $(2^m-1) * 2^n$

suma \`mod\` cenaPiva = $((2^m-1) * 2^n)$ \`mod\` cenaPiva

suma \`mod\` cenaPiva = ((2^m-1) \`mod\` cenaPiva

*

2^n \`mod\` cenaPiva

) \`mod\` cenaPiva

mod 11	2^n	2^{m-1}	mod 11
1	1	0	0
2	2	1	1
4	4	3	3
8	8	7	7
5	16	15	4
10	32	31	9
9	64	63	8
7	128	127	6
3	256	255	2
6	512	511	5
1	1024	1023	0

Kombinácie s opakovaním
kso

```
repeat [] = [ [], [], [], [], [], ...          ::[[t]]
[[]] : repeat [] = [ [[]], [], [], [], [],... ::[[[t]]]
```

```
kso        :: [t] -> Int -> [[t]]
kso xs k   = (foldr f ([[]] : repeat []) xs) !! k
               f x = scanl1 $ (++) . map (x :)
               f x y = (scanl1 $ (++) . map (x :)) y
               f x y = scanl1 ((++) . map (x :)) y
               f x y = scanl1 (\acc -> \ws -> ((++) . map (x :)) acc ws) y
               f x y = scanl1 (\acc -> \ws -> ((++)  (map (x :) acc) ws)) y
               f x y = scanl1 (\acc -> \ws -> ((map (x :) acc) ++ ws)) y
               f :: t -> [[[t]]] -> [[[t]]]
               f x y = scanl1 g y
                    where g          :: [[t]] -> [[t]] -> [[t]]
                          g acc ws =(map (x :) acc) ++ ws
```

https://rosettacode.org/wiki/Combinations_with_repetitions#Haskell https://rosettacode.org/wiki/Combinations_with_repetitions#Haskell

Kombinácie s opakovaním

kso

```
repeat [] = [ [], [], [], [], [], ...          ::[[t]]
[[]] : repeat [] = [ [[]], [], [], [], [],... ::[[[t]]]
```

```
kso          :: [t] -> Int -> [[t]]
kso xs k  = (foldr f ([[]] : repeat []) xs) !! k
                f x y = scanl1 g y                        f :: t -> [[[t]]] -> [[[t]]]
                  where                                   g :: [[t]] -> [[t]] -> [[t]]
                     g acc ws =(map (x :) acc) ++ ws
```

f 4 ([[]] : repeat []) = [[[]], [[4]], [[4,4]], [[4,4,4]], [[4,4,4,4]], [[4,4,4,4,4]], [[4,4,4,4,4,4]],

f 3 $ f 4 ([[]] : repeat []) = [[[]], [[3],[4]], [[3,3],[3,4],[4,4]],[[3,3,3],[3,3,4],[3,4,4],[4,4,4]],

f 2 $ f 3 $ f 4 ([[]] : repeat []) = [[[]], [[2],[3],[4]],[[2,2],[2,3],[2,4],[3,3],[3,4],[4,4]],
[[2,2,2],[2,2,3],[2,2,4],[2,3,3],[2,3,4],[2,4,4],[3,3,3],[3,3,4],[3,4,4],[4,4,4]],…

f 1 $f 2 $ f 3 $ f 4 ([[]] : repeat []) = [[[]], [[1],[2],[3],[4]],
[[1,1],[1,2],[1,3],[1,4],[2,2],[2,3],[2,4],[3,3],[3,4],[4,4]],
[[1,1,1],[1,1,2],[1,1,3],[1,1,4],[1,2,2],[1,2,3],[1,2,4],[1,3,3],[1,3,4],[1,4,4],
[2,2,2],[2,2,3],[2,2,4],[2,3,3],[2,3,4],[2,4,4],[3,3,3],[3,3,4],[3,4,4],[4,4,4]],
[[1,1,1,1],[1,1,1,2],[1,1,1,3],[1,1,1,4],[1,1,2,2],[1,1,2,3],[1,1,2,4],[1,1,3,3],[1,1,3,4],[1,1,4,4],[1,2,2,2],[1,2,2,3],[1,2,2,4],[1,2,3,3],[1,2,3,4],[1,2,4,4],[1,3,3,3],[1,3,3,4],[1,3,4,4],[1,4,4,4],[
2],[2,2,2,3],[2,2,2,4],[2,2,3,3],[2,2,3,4],[2,2,4,4],[2,3,3,3],[2,3,3,4],[2,3,4,4],[2,4,4,4],[3,3,3,3],[3,3,3,4],[3,3,4,4],[3,4,4,4],[4,4,4,4]]]


Have a break. Have a KitKat
Time for a Break

Maximálny súčet
súvislej podpostupnosti

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

to

from

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

$x_{from} + \ldots + x_{to}$

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]]

Intro.hs

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

Maximálny súčet

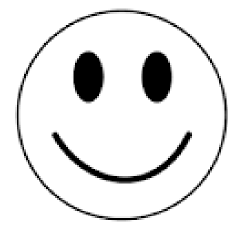
```
maxSucet' :: [Int] -> Int
maxSucet' [] = 0
maxSucet' xs =
     maximum (map (maximum) – maximum  trojuholníkovej matice
           (init (                                 -- posledný prvok - trojuholníková
              foldl (\xss -> \x -> (x:(map (+x) (head xss))): xss)   [[]]   xs)))

maxSucet'' xs = maximum $ map (maximum) $
           init $ foldl (\xss -> \x -> (x:(map (+x) (head xss))): xss) [[]] xs

maxSucet''' = maximum . map (maximum) .
           init . foldl (\xss  -> \x -> (x:(map (+x) (head xss))):xss) [[]]

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

Intro.hs

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

Kadane Algo

```
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)
```



Pamäťová zložitosť O(n)

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

Kadane.hs

Maximálny súčet

```
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 vyskytujúcie sa slovo v reťazci

```
-- rozdeľ na slová podľa oddelovača, viac pozri Data.List.Split
splitOneOf          :: String -> String -> [String]
splitWords          = filter(/= "") . splitOneOf " .,;!@#$%^&*()"
```

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

```
chunks              :: [String] -> [[String]]
chunks []           = []
chunks xs@(w:_) = takeWhile (==w) xs: chunks (dropWhile (==w) xs)
```

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

```
type FreqTable      = [(Int,String)]
chunkLengths        :: [[String]] -> FreqTable
chunkLengths xs  = map (\chunk -> (length chunk, head chunk)) xs
```

"?: " 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! " ...

MostFrequent.hs

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

y++ (cartTemp x yss)

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
```

cartesianMarian.hs

Kartézsky – transformácie

```
-- iniciálne 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]
```

cartesianMarian.hs

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
```

cartesianStrachey.hs

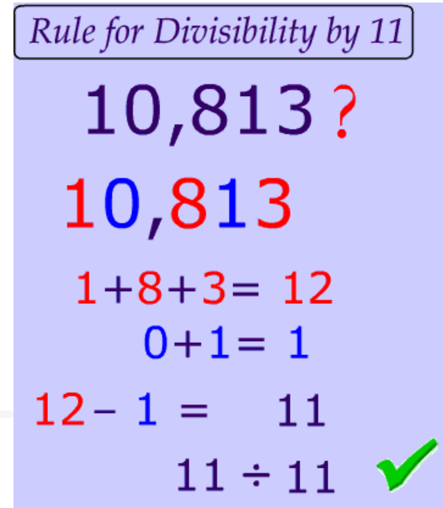
Kartézsky – transformácie

```
-- 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

cartesianStrachey.hs

Delitel’nosť 11



- SK67 8360 5207 0042 0002 6991
- 678360520700420002699 1=11*61669138245492727518 1

- Rodné číslo (.cz, .sk) je delitel’né 11

```
oneStep :: Integer -> Integer
oneStep = \n -> abs $
```
~~uncurry (-) $ foldr (\c -> \(sp,sn) -> (c+sn, sp)) (0,0) $~~

JaroF:
foldr (-) 0 $

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

allSteps :: Integer -> Bool
allSteps = \n -> 0 == (head $ dropWhile (>9) $ iterate oneStep n)

qch1   =  quickCheck(\n -> (n>0) ==> allSteps n == (n `mod` 11 == 0))
```
***Eleven>** qch1
+++ OK, passed 100 tests.

Eleven.hs

BiLandia
(Hejného metóda)

```
pocetMoznosti 0 = 0                                                    -- Martina
pocetMoznosti 1 = 1
pocetMoznosti n | n `mod` 2 == 0 = pocetMoznosti (n `div` 2) + pocetMoznosti (n-1)
                | otherwise = pocetMoznosti (n-1)

pocetMoznosti' 0 = 1                                                   -- Jarka
pocetMoznosti' 1 = 1
pocetMoznosti' x   | x `mod` 2 == 1 = pocetMoznosti' (x-1)
                   | otherwise = (pocetMoznosti' (x-2)) + (pocetMoznosti' (x `div` 2))
qch = quickCheck(\n -> (0<=n && n <= 1000) ==> pocetMoznosti n == pocetMoznosti’ n) -- failed: 0
qch1 = quickCheck(\n -> (0<n && n <= 1000) ==> pocetMoznosti n == pocetMoznosti' n)  -- passed
pocetMoznosti'' 0 = 1                                                  -- Samo
pocetMoznosti'' n = sum (map pocetMoznosti'' [0..(div n 2)])

pocetMoznosti''' 0 = 1                                                 -- and The Winner is: Jakub
pocetMoznosti''' n = sum [pocetMoznosti''' x | x <- [0..n `div` 2]]

qch2 = quickCheck(\n -> (0<n && n <= 1000) ==> pocetMoznosti n == pocetMoznosti'' n) -- passed
```

BiLandia.hs