

# 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

map, filter, and reduce  
explained with emoji 🤔

```
map([🐮, 🌽, 🐔, 🌽], cook)  
=> [🍔, 🍟, 🍗, 🍿]
```

```
filter([🍔, 🍟, 🍗, 🍿], isVegetarian)  
=> [🍟, 🍿]
```



```
reduce([🍔, 🍟, 🍗, 🍿], eat)  
=> 🤩
```



# Vlastnosti map, filter

---

Na zamyslenie:

- $\text{filter } p (\text{map } f \text{ } xs)$        $= ??? (\text{filter } (p.f) \text{ } xs)$       
- $\text{filter } p (\text{map } f \text{ } xs)$        $= \text{map } f (\text{filter } (p.f) \text{ } xs)$       
- $\text{filter } p . \text{map } f$        $= \text{map } f . \text{filter } (p.f)$

Dôkaz:

definícia map filter pomocou list-comprehension nám to objasní

$\text{filter } p (\text{map } f \text{ } xs)$

$= \text{filter } p [ f \text{ } x \mid x \leftarrow xs ]$

$= [ y \mid y \leftarrow [ f \text{ } x \mid x \leftarrow xs ], p \text{ } y ]$

$= \dots \text{ tu treba rozmýšľať } \dots$

$= [ f \text{ } x \mid x \leftarrow xs, p (f \text{ } x) ]$

$= \text{map } f [ x \mid x \leftarrow xs, p (f \text{ } x) ]$

$= \text{map } f (\text{filter } (p.f))$



# Quíz - prémia

nájdite pravdivé a zdôvodnite

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

- $\text{map } f . \text{take } n = \text{take } n . \text{map } f$

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

- $\text{map } f . \text{filter } p = \text{map } \text{fst} . \text{filter } \text{snd} . \text{map } (\text{fork } (f,p))$

where  $\text{fork} :: (a \rightarrow b, a \rightarrow c) \rightarrow a \rightarrow (b,c)$

$\text{fork } (f,g) \ x = (f \ x, g \ x)$

- $\text{filter } (p . g) = \text{map } (\text{inverzna\_g}) . \text{filter } p . \text{map } g$   
ak  $\text{inverzna\_g} . g = \text{id}$

- $\text{reverse} . \text{concat} = \text{concat} . \text{reverse} . \text{map } \text{reverse}$

- $\text{filter } p . \text{concat} = \text{concat} . \text{map } (\text{filter } p)$



# 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)
```



# 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]]
kso :: [t] -> Int -> [[t]]
vbo :: (Eq t) => [t] -> Int -> [[t]]
vso :: [t] -> Int -> [[t]]
```

?

```
n!
(n nad k)
((n+k-1) nad k)
n.(n-1). ... .(n-k+1)
```



# Definované typy - QuickCheck

---

Ak definujeme vlastnú dátovú štruktúru, ako využiť quickCheck ?

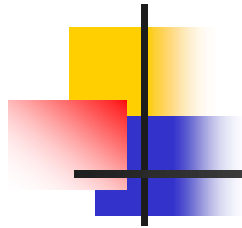
```
data BVS t = Nil | Node (BVS t) t (BVS t) deriving(Show, Eq)
```

- 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), od slajdu 11
- [https://github.com/Programovanie4/Prednasky/blob/master/13/13\\_java.pdf](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 konšuktora, potom zavolanie konšuktora 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 zdefinoval:

"?: " (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'), ...





# 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



# 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]
```

Pre nami definované typy  
XXX musíme definovať  
inštanciu triedy Arbitrary XXX

```
"?: " generate (arbitrary::Gen Minca)
```

```
"?: " (generate arbitrary)::IO Minca
```

```
falosnaMinca :: Gen Minca
```

```
falosnaMinca = frequency [(1,return Hlava), (2,return Panna)]
```

```
-- "?: " generate falosnaMinca
```

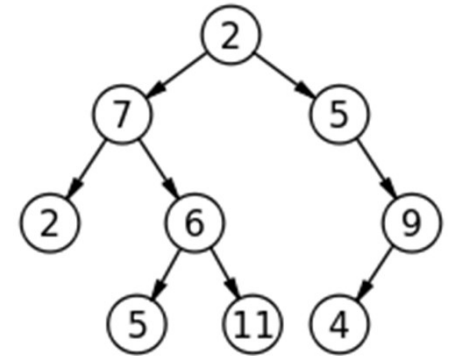


# Generátory - zoznam

```
arbitraryListMax8Len :: Arbitrary a => Gen [a]    -- náhodný zoznam len <= 8
arbitraryListMax8Len =
    do {                                           "?: " generate (arbitraryListMax8Len::Gen [Int])
        [-21,12,17,16,4,-20]
        k <- choose (0, 8)::(Gen Int);
        sequence [ arbitrary | _ <- [1..k] ] }

arbitraryList :: Arbitrary a => Gen [a]
arbitraryList =
    mysized ( \n -> do {                           "?: " generate (arbitraryList::Gen [Int])
        [-9,7,14,24,18,28,-4,0,22,12,-14]
        k <- choose (0, n) ;
        sequence [ arbitrary | _ <- [1..k] ] }
    )
mysized :: (Int -> Gen a) -> Gen a                "?: " generate
mysized f = f 50                                (mysized (\n -> choose(n,n)))
50
```

# 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 )    "?: " generate (arbitrary :: Gen (Tree Int))
                                   Leaf (-18)
    , (1, liftM3 Node arbitrary arbitrary arbitrary)
  ]
```

```
strom :: Gen (Tree Int)    "?: " generate strom
strom = frequency [
    (1, liftM Leaf arbitrary )
    , (10, liftM3 Node arbitrary arbitrary arbitrary)
  ]
Node (Node (Leaf (-2)) 3 (Leaf (-6))) 23 (Leaf 22)
```



# 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 <
```

-- nájsť 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
```



# 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))
```

```
    &&
```

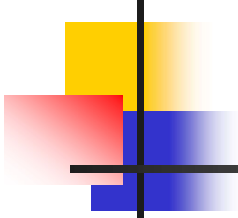
```
    (all (>value) (flat right))
```

```
    &&
```

```
    isBVS left
```

```
    &&
```

```
    isBVS right
```



# 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 ??

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
```

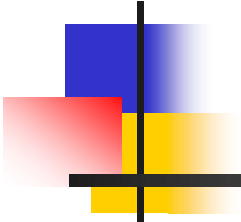
```
    | x < value = find x right
```

```
    | 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 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

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

- scanr/scanl

- $\text{scanr } f \ z [x_1, \dots, x_n] = \text{reverse } [z, (f \ x_n \ z), \dots, (f \ x_2 \dots (f \ x_n \ z) \dots), (f \ x_1 (f \ x_2 \dots (f \ x_n \ z) \dots))]$
- $\text{scanl } f \ z [x_1, \dots, x_n] = [z, (f \ z \ x_1), ((f \ z \ x_1) \ x_2), \dots, (\dots((f \ z \ x_1) \ x_2) \dots x_n)]$
- $\text{scanr1 } f [x_1, \dots, x_n] = \text{reverse } [x_n, (f \ x_{n-1} \ x_n), \dots, (f \ x_1 (f \ x_2 \dots (f \ x_{n-1} \ x_n) \dots))]$
- $\text{scanl1 } f [x_1, \dots, x_n] = [x_1, (f \ x_1 \ x_2), ((f \ x_1 \ x_2) \ x_3), \dots, (\dots((f \ x_1 \ x_2) \ x_3) \dots x_n)]$

- iterate

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

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

# Haskell – foldr

`foldr`  $:: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$

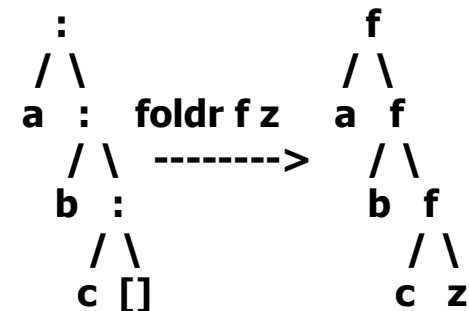
`foldr f z []` = `z`

`foldr f z (x:xs)` = `f x (foldr f z xs)`

`a : b : c : []`  $\rightarrow$  `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

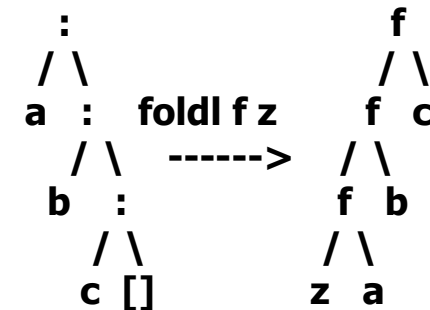
---

`foldl`  $:: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$

`foldl f z []` = `z`

`foldl f z (x:xs)` = `foldl f (f z x) xs`

`a : b : c : []`  $\rightarrow f (f (f z a) b) c$



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

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



# 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]`

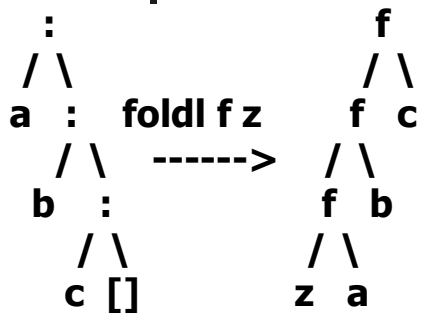
- `foldr (-) 0 [1..100] =`

$$(1-(2-(3-(4-\dots-(100-0)))))) = 1-2 + 3-4 + 5-6 + \dots + (99-100) = -50$$

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

$$(\dots(((0-1)-2)-3) \dots - 100) = -5050$$

# Kvíz

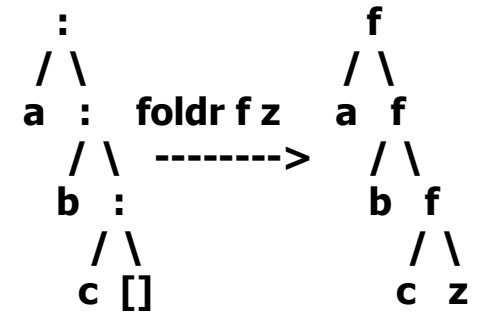


$\text{foldr } (:) [] \text{ xs} = \text{xs}$

$\text{foldr } (:) \text{ ys xs} = \text{xs} ++ \text{ys}$

$\text{foldr } ? ? \text{ xs} = \text{reverse xs}$

$\text{foldr } ((:) . h) [] = ???$



<http://foldl.com/>



Pre tých, čo zvládli kvíz, odmena !

kliknite si podľa vašej politickej orientácie

<http://foldr.com/>





# Funkcia je hodnotou

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

- čo je foldr (.) id  $[(+1), (+2), (*3)]$  ??

akého je typu

foldr (.) id  $[(+1), (+2), (*3)]$  100

foldl (.) id  $[(+1), (+2), (*3)]$  100

$[a \rightarrow a]$

303

???

lebo skladanie fcií je asociatívne:

- $((f \cdot g) \cdot h) x = (f \cdot g) (h x) = f (g (h x)) = f ((g \cdot h) x) = (f \cdot (g \cdot h)) x$
- funkcie nevieme porovnávať, napr.  $\text{head } [(+1), (+2), (*3)] == \text{id}$
- funkcie vieme permutovať,  $\text{length } \$ \text{permutations } [(+1), (+2), (*3), (^2)]$





# 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/vyvráťte: `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



# take pomocou foldr/fold

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`

# 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
```



# 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** – ušetrný 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** – ušetrný 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` =  $(3 \cdot x + 1) + \text{foo } xs$

| `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

# Collatz

(a na príkladoch)

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

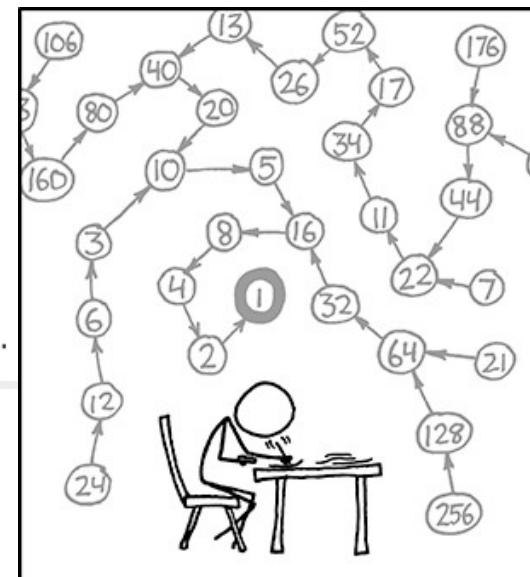
Čo robí táto funkcia ?

`collatz :: Integer -> [Integer]`

`collatz 1 = []`

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

`| otherwise = n : collatz (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

`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, ... , 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 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 zoprimlizované

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

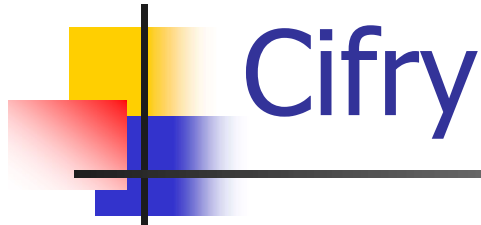
`cifry 12345 == [1,2,3,4,5]`

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

`cifry n = reverse (cifryR n)`

`cifryR 0 = []`

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



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 == [12345,1234,123,12,1,0,0,0,0,0,0,0,0,0,0...]

[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)

# Binárne číslo $\{1\}^+ \{0\}^*$

■  $\underbrace{111\dots111}_m \underbrace{00\dots000}_n = (2^m - 1) * 2^n$

null \$

dropWhile (==1) \$

dropWhile(==0) \$

map (`mod` 2) \$

takeWhile (>0) \$

iterate (`div` 2)



True

[]

[1, 1]

[0, 0, 1, 1]

[12, 6, 3, 1]

12 = [12, 6, 3, 1, 0, 0, 0...]



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

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

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



$2^n \text{ `mod` cenaPiva}$

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
```

# 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
```

```
    where
```

```
        g acc ws =(map (x :) acc) ++ ws
```

```
f :: t -> [[t]] -> [[t]]
```

```
g :: [[t]] -> [t] -> [[t]]
```

```
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],[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



# 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							
		0	1	2	3	4	5	6	7
from	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

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

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

```
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
```



# 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

---

```
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
```

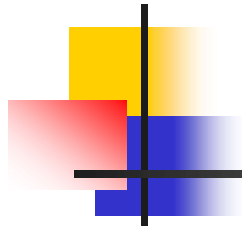
-- slušnejšie prepísané

```
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>

# Deliteľnosť 11

Rule for Divisibility by 11

10,813?

10,813

$1+8+3=12$

$0+1=1$

$12-1=11$

$11 \div 11$



- SK67 8360 5207 0042 0002 6991
- $6783605207004200026991 = 11 * 616691382454927275181$
- Rodné číslo (.cz, .sk) je deliteľné 11

oneStep :: Integer -> Integer

oneStep = \n -> abs \$

~~uncurry (-) \$ foldr (\c > \ (sp,sn) -> (c+sn, sp)) (0,0) \$~~ foldr (-) 0 \$

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

JaroF:

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.



# 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: (
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
```