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

Funkcie a funkcionály

a ich

referečná transparentosť

Peter Borovanský

Všetko, čo by ste chceli vedieť o Haskelli, ale báli ste sa spýtať...

Funkcia

a čo s ňou

- **aplikovať** – na argument/y
  - Haskell: f 5, goo 17 21, fib (n-2) -- zátvorky píšeme kvôli zložitému argumentu, (fib n)-2
  - inde: f(5), goo(17,21) fib(n-2)

  Celá pravda o Haskelli:

  že **a b c d = (((a b) c) d)**...lebo operátor aplikácie funkcie na argument je *ľavo-asociatívny*, teda, ak zabudnem zátvorky, tak ich iniciatívne chápe grupované doľava

- **abstrahovať** / abstrakcia
- jazyk je funkcionálny, ak viete vytvoriť funkciu (λ-abstrakciu) počas behu program
  - Haskell:(+) = \x -> \y -> x+y
  - Python: `lambda x, y: x + y`
  - JS: (a, b) => a + b
  - Java: (a, b) -> a + b
  - Kotlin: a:Int, b:Int -> a + b
  - Swift: a, b in a+b
  - C, Pascal, … ☹
- syntax doesn't matter

Všetko, čo by ste chceli vedieť o Haskelli, ale báli ste sa spýtať...

Funkcia

a čo s ňou

- má typ
  - Python, JS, ...: ☹
  - **Haskell**:
    (+) :: Int -> Int -> Int, čo **nie je** to isté ako :: (Int, Int) -> Int
  - **Curry**ing (+ 4) :: Int -> Int, (+ 4 7) :: Int = 11
  - Celá pravda o Haskelli:
  - **Int -> Int -> Int -> Char = Int -> (Int -> (Int -> Char))** … lebo operátor funkčného typu **->** je *pravo asociatívny*, teda ak zabudnem zátvorky, tak ich chápe doprava. Explicitne, (Int->Int) -> (Int -> Int)
  - **Int -> Int -> String != (Int, Int) -> String**… lebo prvé je funkcia, ktorá vráti funkciu, ktorá vráti String. Vďaka *currying* ju volám takto: f 4 5, čo je (f 4) 5. Druhé je funkcia, ktorá čaká dvojicu. Musím ju volať takto: g (4,5), a vyzerám skôr Javista, a nie Haskellista
  - Príklad:

```
f :: Int -> (Int -> Int)
f a b = 10*a+b
f 5 7 = 57
f 5 :: Int -> Int
f 5 = \b -> 10*5+b
f 5 b = 10*5+b
f 5 y = 10*5+y
```

```
g :: (Int -> Int) -> Int
g h = h 7
g (+11) = 18
g (\x->x+11) = 18
g (^2) = 49
g (\x->x^2) = 49
g (*5) = 35
```

Všetko, čo by ste chceli vedieť o Haskelli, ale báli ste sa spýtať...

Funkcia

a čo s ňou

- komponovať – f . g - z matematiky $(f \circ g)(x) = f(g(x))$
- je asociatívna, nie je komutatívna, identita id = \x->x je neutrálny prvok

Haskell:

```
(.) :: (b->c) -> (a->b) -> (a->c)
(.) = \f -> \g -> \x -> f (g x)
f . g = \x -> f (g x)
(f . g) x = f (g x)

composeMany :: [a->a] -> (a->a)
composeMany [] = id
composeMany (f:fs) = f . composeMany fs

-- ak poznáte reduce.py ≈ foldr
composeMany xs = foldr (.) id xs
```

Bezargumentový snobizmus

```
composeMany = foldr (.) id
```

pramení tiež z matematiky, lebo matematik tiež radšej napíše f = g miesto **∀x: f(x) = g(x)**

Python, JS, ...:

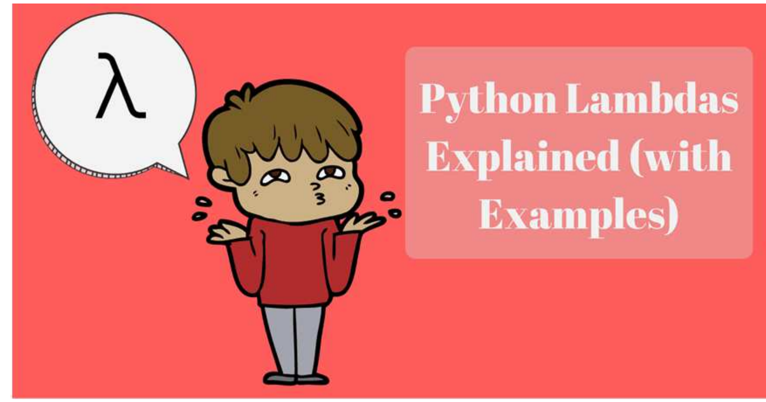
```
def compose(f, g):
    return lambda x: f(g(x))


def composeMany(*fs):
    return reduce(compose, fs)

print(composeMany(
        lambda x: x+1,
        lambda x: x+2,
        lambda x: x*3
        )(10))
```

Python Kvíz

pre aplikovancov




```
print(map(lambda x: x*x, [1,2,3,4,5]))
```

<map object at 0x037

```
print(list(map(lambda x: x*x, [1,2,3,4,5])))
```

[1, 4, 9, 16, 25]

```
print(list(filter(lambda y:y>10,map(lambda x: x*x, [1,2,3,4,5]))))
```

[16, 25]

```
from functools import reduce
print(reduce((lambda x, y: x * y), [1, 2, 3, 4]))
```

24

```
print(reduce((lambda x, y: x + y), [1, 2, 3, 4]))
```

10

```
print(reduce((lambda x, y: x - y), [1, 2, 3, 4]))
```

-8

```
def compose(f, g):
        return lambda x: f(g(x))
print(compose( lambda x: x+1, lambda x: x*3 )(10))
```

31

```
def composeMany(*fs):
        return reduce(compose, fs)
print(composeMany(lambda x:x+1, lambda x:x+2, lambda x:x*3)(10))
```

33

lambdas.hs

All Things Pythonic

Guido van Rossum: The fate of reduce() in Python 3000, (r.2005)

- Python aquired lambdas, reduce(), filter() and map() thanks a Lisp hacker
- despite of the PR value, I think these features should be cut from Python 3
- **Update: lambda, filter and map will stay (the latter two with small changes, returning iterators instead of lists). Only <u>reduce</u> will be removed from the 3.0 standard library. You can import it from functools.**

https://www.artima.com/weblogs/viewpost.jsp?thread=98196

Lambda & kompozícia

prvočíselná etuda

6 spôsobov ako naprogramovať prvočíselný test, ilustrácia možností jazyka

```
-- upravené riešenie s HaskellThonu, delitele len po sqrt
jePrvocislo n = null (delitele n)
sqrti n        = floor (sqrt $ fromIntegral n)
delitele n     = [d | d <- [2..sqrti n], n `mod` d == 0]

-- explicitne viditeľné lambda abstrakcie
jePrvocislo' =  \n -> null (delitele' n)
sqrti'       =  \n -> floor (sqrt $ fromIntegral n)
delitele'    =  \n -> [d | d <- [2..sqrti' n], n `mod` d == 0]

-- 'funkcionálny' snobizmus, zamlčané explicitné argumenty, poučívanie kompozície fcií
jePrvocislo'' =  null . delitele''
sqrti''       = floor . sqrt . fromIntegral
delitele''    = \n -> [d | d <- [2..sqrti'' n], n `mod` d == 0]
```

primes.hs

Clojure

prvočíselná etuda

```
-- closure: sqrt a delitele sú vnorené funkcie, ktoré vidia do prostredia materskej fcie
jePrvocislo''' n = null delitele where
                    sqrtn = floor (sqrt $ fromIntegral n)
                    delitele = [d | d <- [2..sqrtn], n `mod` d == 0]

-- let … in … vždy nahradí where, ale naopak to neplatí
jePrvocislo'''' n = let sqrtn = floor (sqrt $ fromIntegral n)
                        delitele = [d | d <- [2..sqrtn], n `mod` d == 0]
                    in null delitele

-- konečne riešenie správne aj matematicky … ☺
jePrvocislo''''' n = n>1 && null delitele where
                    sqrtn = floor (sqrt $ fromIntegral n)
                    delitele = [d | d <- [2..sqrtn], n `mod` d == 0]
```

```
main = do putStrLn $ show $ filter jePrvocislo [1..100]
          putStrLn $ (show . filter jePrvocislo'') [1..100]
          putStrLn $ (show . filter jePrvocislo''') [1..100]
          putStrLn $ (show . filter jePrvocislo'''') [1..100]
          putStrLn $ (show . filter jePrvocislo''''') [1..100]
```

```
*Main> main
[1,2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]
[1,2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]
[1,2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]
[1,2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]
```

primes.hs

Iteracia funkcie f^n

```
iterujf       :: Int -> ((Float -> Float) -> (Float -> Float) )
iterujf       = iteruj

iteruj        :: Int -> ((t -> t) -> (t -> t) )
iteruj 0      f  = (\x -> x)                              -- identita
iteruj n      f  = (\x -> f (iteruj (n-1) f x))

iteruj'       :: Int -> ((t -> t) -> (t -> t) )
iteruj' 0     f  = (\x -> x)                              -- identita
iteruj' n     f  = (\x -> iteruj' (n-1) f (f x))

iteruj''      :: Int -> ((t -> t) -> (t -> t) )
iteruj‟’ 0 f x = x
iteruj‟’ n f x = f (iteruj'' (n-1) f x)

iteruj'''     :: Int -> ((t -> t) -> (t -> t) )
iteruj"’ 0 f  = id
iteruj"’ n f  = f  . (iteruj''' (n-1) f)
```

Iteruj.hs

Iteracia funkcie f^n

```
iteruj'''       :: Int -> ((t -> t) -> (t -> t) )
iteruj"’ 0 f  = id
iteruj"’ n f  = f  . (iteruj''' (n-1) f)

iteruj''''      :: Int -> ((t -> t) -> (t -> t) )
iteruj'"’ 0 f  = id
iteruj'"’ n f  = (iteruj'''' (n-1) f) . f

iteruj_foldr              :: Int -> ((t -> t) -> (t -> t) )
iteruj_foldr n f          = foldr (.) id (replicate n f)

iteruj_foldl              :: Int -> ((t -> t) -> (t -> t) )
iteruj_foldl n f          = foldl (.) id (take n (cycle [f]))

iteruj_using_iterate               :: Int -> ((t -> t) -> (t -> t) )
iteruj_using_iterate n f x         = iterate f x !! n

iteruj_funkciu :: Int -> ((t -> t) -> (t->t))
iteruj_funkciu n f  = iteruj_f !! n  where iteruj_f = id:[f . g | g <- iteruj_f]
```

Iteruj.hs

Všetko, čo by ste chceli vediet o Haskelli, ale báli ste sa spýtať

Otázka z interview

a ako na ňu

Haskell:

```
dvojica a b = pair
    where pair f = f a b
dvojica :: s->t->((s->t->v)->v)
-- inak
dvojica a b = \f -> f a b
dvojica a b f = f a b
prvy (dvojica a b) = a
druhy (dvojica a b) = b
prvy p = p (\x-> \y -> x)
druhy p = p (\x-> \y -> y)
print $ prvy (dvojica 4 5)
print $ druhy (dvojica 4 5)
prvy p = p true
   where true a b = a
druhy p = p false
    where false a b = b
```

Python, JS, ...:

```
def dvojica(a, b):
    def pair(f):
        return f(a, b)
    return pair
# inak
def dvojica(a,b):
    return lambda f: f(a,b)

def head(p):
    return p(lambda a,b :a)

def tail(p):
    return p(lambda a,b:b)

print(head(dvojica(4,5)))
print(tail(dvojica(4,5)))
```

pair.hs, pair.py

Často sa *obrábabie* zoznamov prezentuje ako funkcionálne programovanie, pritom je to len nevyhnutný úvod k lepšiemu...

Zoznamová zoznamka

Haskell:

xs = [1,2,3,4,5] [1..5]

length xs

xs!!i

neexistuje-immutable list

head xs

tail xs

last xs

init xs

take n xs

drop n xs

take m (drop n xs)

xs++xs

reverse xs

Python:

[1,2,3,4,5]		
len xs		
xs[i]	.. indexy 0..length xs-1	
xs[i]=…		
xs[0]		1
xs[1:]		[2,3,4,5]
xs[len(xs)-1]		5
xs[:len(xs)-1]		[1,2,3,4]
xs[:n]		
xs[n:]		
xs[n:n+m]		
xs+xs		[1,2,3,4,5,1,2,3,4,5]
xs.reverse()		returns void

import Data.List

http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-List.html

Data.List

Copyright	(c) The University of Glasg
License	BSD-style (see the file libra
Maintainer	libraries@haskell.org
Stability	stable
Portability	portable
Safe Haskell	Trustworthy
Language	Haskell2010

Operations on lists.

Contents

Basic functions

```
(++) :: [a] -> [a] -> [a]    infixr 5    # Source
```

Append two lists, i.e.,

```
[x1, ..., xm] ++ [y1, ..., yn] == [x1, ..., xm, y1, ..., yn]
[x1, ..., xm] ++ [y1, ...] == [x1, ..., xm, y1, ...]
```

If the first list is not finite, the result is the first list.

```
head :: [a] -> a    # Source
```

Extract the first element of a list, which must be non-empty.

```
last :: [a] -> a    # Source
```

Extract the last element of a list, which must be finite and non-empty.

```
tail :: [a] -> [a]    # Source
```

Extract the elements after the head of a list, which must be non-empty.

```
init :: [a] -> [a]    # Source
```

Return all the elements of a list except the last one. The list must be non-empty.

Možno vás to prekvapí

- ale zoznam – list – [a]

> - **je immutable/nemenný asi ako String v Jave**
> - raz, keď ho vytvoríte, nikdy ho už nezmeníte, len môžete vytvoriť nový, trochu iný
> - inak povedané
>
>   indexovanie xs!!i existuje, ale niečo ako nahradenie xs[i] = v neexistuje
>
> Chce to objaviť inú dátovú štruktúru, a sú: Data.Array, Data.Set, ...
> - ale tie sú tiež immutable/nemenné
> - …lebo filozófia
>
> - prvý pocit mutable-dátovej štruktúry poskytnú až monády, state monad

> - **je homogénny v type**
>
> - v zozname typu [a] sú len hodnoty typu a, a žiadne iné
> - žiaden Any, Object, mother-of-all-types ... neexistuje, našťastie

List-comprehension

Každý poriadny kurz FP začína funkcionálmi map a filter:

...ale my sme trénovali list-comprehension:

```
[ f x | x <- xs, p x]        [ f(x) for x in xs if p(x)]
```

map, filter sú deriváti list-comprehension

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

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

Všetko, čo ste chceli zmeniť, a nikdy sa vám to nepodarilo

- zoznam ("pole") xs vieme indexovať indexami i <- [0..length xs-1]

```
        xs!!i                                   -- getter
```

- neexistuje setter xs[i] = value

```
set  ::  [t] -> Int -> t -> [t]
set xs i value | i < 0              = xs    -- out of range
               | i >= length xs     = xs    -- out of range
               | otherwise          = (if i == 0 then value else y):set ys (i-1) value
                                      where (y:ys) = xs
               | otherwise          = let (y:ys) = xs in
                                      (if i == 0 then value else y):set ys (i-1) value
set''  ::  [t] -> Int -> t -> [t]
set'' xs i value | i < 0            = xs    -- out of range
                 | i >= length xs   = xs    -- out of range
                 | otherwise        = [xs!!j | j <- [0.. i-1] ] ++ [value] ++
                                      [xs!!j | j <- [i+1..length xs-1] ]
```

zoznam.hs

Zoznamová rekurzia

```
-- vyber prvých n prvkov zo zoznamu
take                  :: Int -> [a] -> [a]
take 0 _              = []
take _ []             = []
take n (x:xs)         = x : (take (n-1) xs)

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

```
"?: " take 5 [1,3..100]
[1,3,5,7,9]
"?: " length (take 5 [1,3..100])
5
"?: " length $ take 5 [1,3..100]
5
```

Hypotéza (pre ľubovoľné n a xs) platí:

- length (take n xs) = n
- length $ take n xs = n   -- dolárová notácia
- (length . take n) xs = n   -- kompozícia funkcií z matematike

Dôkaz - length (take n xs) = n
(matematická indukcia)

Indukcia (vzhľadom na dĺžku/štruktúru xs):

- **xs = []**

  length (take n []) = 0

  0 = 0

  č.b.t.d.

- **xs = (y:ys)**

  length (take n (y:ys)) = n

  length (y:take (n-1) ys) = n

  1 + <u>length (take (n-1) ys)</u> = n

  indukčný predpoklad, |ys| < |xs|

  1 + <u>(n-1)</u> = n

  č.b.t.d.

Definície z predošlej strany:

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

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

QuickCheck

Elegantný nástroj na testovanie (!!! nie dôkaz !!!) hypotéz

"?: " import Test.QuickCheck

"?: " quickCheck (\(xs,n) -> length (take n xs) == n)

*** Failed! Falsifiable (after 2 tests and 1 shrink):

"?: " verboseCheck (\(xs,n) -> length (take n xs) == n)

Passed:

([],0)

Passed:

([()],1)

Failed:

([],-1)

*** Failed! Failed:

Neplatí to pre n záporne, lebo napr. take (-3) [1..100] = [],

resp. naša definícia nepokrýva prípad n < 0

!!! ALE MY SME TO AJ TAK "DOKÁZALI"... !!! ☹

QuickCheck.hs

QuickCheck

Podmienka: miesto písania

<u>if n >= 0 then</u> length (take n s) == n **<u>else True</u>**

Napíšeme pre-condition pomocou **==>**

"?: " verboseCheck (\(xs,n) -> **n>=0 ==>** length (take n xs) == n)

Passed:

([],0)

Failed:

([()],2)

Neplatí to pre ak length xs < n

"?: " quickCheck (\(xs,n) -> n>=0 && length xs >= n ==>

length (take n xs) == n)

*** Gave up! Passed only 35 tests.

Tvrdenie sme **overili** na niekoľkých prípadoch, ale to **nie je dôkaz**.

V dôkaze môžeme urobiť chybu (ako na slajde 2), QuickCheck slúži ako nástoj na hľadanie/odhaľovanie kontrapríkladov, kedy naše tvrdenie neplatí.

QuickCheck.hs

Don’t write tests!

Generate them from properties

QuickCheck

- miesto písania unit testov, quickcheck vám ich (nejaké) vygeneruje
- vy potom nepíšete testy, ale vlastnosti vašich programov.

O niečom podobnom dávno snívali/dúfali Hoare, Dijkstra, ...

- s rozdielom, že vlastnosti programov chceli dokázať,
- miesto hľadania kontrapríkladu.

Quickcheck:

- generuje náhodné vstupné hodnoty, pre základné aj definované typy
  - Int, Bool, ...
  - [Int], String, …
  - Int->Int, Int->Bool
- ak nájde kontrapríklad (už vieme, že to neplatí), snaží sa ho zminimalizovať/zjednodušiť, napr: length (take n xs)==n neplatí pre length (take 21 [5,-192,3981,-291,2220,-192,22,12,-192,-1]) == 21

Don't write tests!

Generate them from properties

QuickCheck

autori: Koen Claessen, John Hughes

Príklad Parretovho pravidla 20:80 - za 20% energie chytíte 80% problémov

Príklad (viac tu):

Collatz (viac tu) je funkcia f(n)= if n `mod` 2 == 0 then n/2 else 3n+1.

```
f      :: Integer -> Integer
f n    | even(n) = n `div` 2
       | odd(n)  = 3*n + 1
collatz   :: Integer -> Bool
collatz   1 = True
collatz   n = collatz (f n)
```

```
"?: " quickCheck (\n -> n > 0 ==> collatz(n))
+++ OK, passed 100 tests.
"?: " quickCheckWith stdArgs{ maxSuccess = 100000 }
                  (\n -> n > 0 ==> collatz(n))
+++ OK, passed 100000 tests.
```

Paul Erdős: "Mathematics may not be ready for such problems." offered $500 for its solution.

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

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

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

<u>if p2 x then</u>

<u>if p1 x then</u> 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

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

Funkcia argumentom 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
map f xs        = [ f x | x <- 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]]
```

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

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:

filter p (map f xs) = filter p [f x | x<-xs]
= [y | y <- [f x | x<-xs], p y]
= [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

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

- Introduction to QuickCheck
  https://wiki.haskell.org/Introduction_to_QuickCheck2

- Introduction to QuickCheck by example: Number theory and Okasaki's red-black trees
  http://matt.might.net/articles/quick-quickcheck/

- K.Claessen, J.Hughes:QuickCheck:A Lightweight Tool for Random Testingof Haskell Programs
  https://www.eecs.northwestern.edu/~robby/courses/395-495-2009-fall/quick.pdf

- A QuickCheck Tutorial: Generators
  https://www.stackbuilders.com/news/a-quickcheck-tutorial-generators

Definované typy

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)

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
- 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

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'), …
```

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

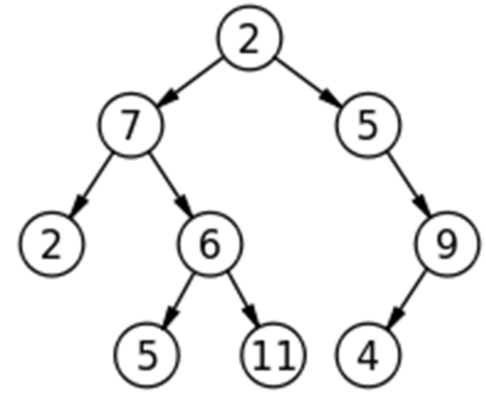
Generátory - zoznam

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

arbitraryList :: Arbitrary a => Gen [a]
arbitraryList =                              "?: " generate (arbitraryList::Gen [Int])
  mysized ( \n -> do {                       [-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
```

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 :: 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
```

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
```

```
-- 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)
```

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

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

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

http://foldl.com/

http://foldr.com/

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

- 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

Zákon fúzie – pre foldr

Fussion Law:

Nech g1, g2 sú binárne funkcie, z1, z2 konštanty
Ak pre funkciu f platí :

f z1 = z2 && f (g1 a b) = g2 a (f b)

potom platí

f . (foldr g1 z1 xs) = foldr g2 z2 xs

Príklad použitia Fussion Law:

$$(n*).\ \underbrace{\text{foldr } (+)\ 0}_{\text{sum}} = \text{foldr } ((+).(n*))\ 0$$

Dôkaz (pomocou Fussion Law): overíme predpoklady
čo je čo ?!:

f = (n*), z1 = z2 = 0, g1 = (+), g2 = (+). (n*)

treba overiť:

- (n*) 0 = 0 ☑
- Ľ.S.=(n*) (a+b) = (n*a + n*b) = (+).(n*) a ((n*) b) = P.S. ☑

Vlastnosti

Acid Rain (fold/build/deforestation theorem)

$$\underbrace{\underbrace{\text{foldr f z}}_{[x]\text{->}u}\ .\ \underbrace{\text{g (:) []}}_{t\text{->}[x]}}_{t\text{->}u} = \text{g f z}$$

Intuícia: Keď máme vytvoriť zoznam pomocou funkcie g zo zoznamových konštruktorov (:) [], na ktorý následne pustíme foldr, ktorý nahradí (:) za f a [] za z, namiesto toho môžeme konštruovať priamo výsledný zoznam pomocou g f z.

Otypujme si to (aspoň):

Ak z :: u, potom f :: x->u->u, foldr f z :: [x]->u.

Ľavá strana: ([x]->u).(t->[x]) výsledkom je typ t->u

Pravá strana: g :: (x -> u -> u) -> u -> (t ->u) foldr f z . g (:) [] = g f z

length . map _ = length

map :: (a -> b) -> [a] -> [b]
map h = foldr ((:) . h) [] -- (:).h a as = (:)(h a as) = h a: as
= (\x -> \y -> foldr (x . h) y) (:) []
g

length :: [a] -> Int
length = foldr (\ -> \n -> n+1) 0
f z

length . map h = ... length

Ľ.S. = (foldr (\ _ -> \n -> n+1) 0) . (foldr ((:) . h) []) =
= podľa Acid Rain theorem (f = (\ _ -> \n -> n+1), z = 0, ale čo je g ? …
g x y = (foldr (x . h) y)
g f z = (foldr (f . h) z) = foldr ((\ _ ->\n -> n+1) . h) 0 =
foldr ((_ ->\n -> n+1)) 0 = length = P.S.
lebo (tento krok pomalšie):
((\ _->\n ->n+1) . h) x y = (\ _->\n->n+1) (h x) y = (\n->n+1) y = y+1

Iný príklad acid rain

```
g     :: (Integer ->u -> u) -> u -> Integer -> u
g h w 0 = w
g h w n = h n (g h w (n-1))
```

```
"?: " ((foldr (*) 1) . (g (:) [])) 100
93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000
"?: " g (*) 1 100
93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000
```

```
g'     :: Integer -> [t]
g' 0 = []
g' n =  n : (g' (n-1))

g''     :: Integer -> Integer
g'' 0 = 1
g'' n = n * (g'' (n-1))
```

acid_rain.hs

foldr a foldl pre pokročilejších

definujte foldl pomocou foldr, alebo naopak:

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

myfoldl f z xs = foldr (\x -> \y -> (f y x)) z (myReverse xs)
myfoldr f z xs = foldl (\x -> \y -> (f y x)) z (myReverse xs)

(flip f)

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

- odstránime myReverse
  myReverse xs = foldr (\x -> \y -> (y ++ [x])) [] xs
  myfoldl' f z xs = foldr (\x -> \y -> (f y x)) z
  (foldr (\x -> \y -> (y ++ [x])) [] xs)

- odstránime ++
  xs ++ ys = foldr (:) ys xs
  myfoldl'' f z xs = foldr (\x -> \y -> (f y x)) z
  (foldr (\x -> \y -> (foldr (:) [x] y)) [] xs)

hmmm..., teoreticky (možno) zaujímavé, prakticky nepoužiteľné ...

foldr a foldl posledný krát

Zamyslime sa, ako z foldr urobíme foldl:

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

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

induktívne predpokladajme, že rekurzívne volanie foldr nám vráti výsledok, t.j. hodnotu y, ktorá zodpovedá foldl:

- y = myfoldl f [b,c] = \z -> f (f z b) c

nech x je ďalší prvok zoznamu, t.j.

- x = a

ako musí vyzerať funkcia ?, ktorou fold-r-ujeme, aby sme dostali myfoldl f [a,b,c] = \z' -> f (f (f z' a) b) c = ? x y

- ? = (\x y z' -> y (f z' x))

dosaďme:

- (\z' -> (\z -> f (f z b) c) (f z' a)) =
- (\z' -> f (f (f z' a) b) c) =
- \z' -> f (f (f z' a) b) c

Pre tých, čo neveria, fakt posledný krát

? = (\x y z' -> y (f z' x))

- myfoldl''' f xs z = foldr (\x y z -> y (f z x)) id xs z
- myfoldl''' f [] = id
- myfoldl''' f [c] = (\x y z -> y (f z x)) c id = \z -> f z c
- myfoldl''' f [b,c] = (\x y z -> y (f z x)) b (\w -> f w c) =
  \z -> (\w -> f w c) (f z b) =
  \z -> f (f z b) c
- myfoldl''' f [a,b,c] = (\x y z -> y (f z x)) a (\w -> f (f w b) c) =
  \z -> (\w -> f (f w b) c) (f z a) =
  \z -> f (f (f z a) b) c
- myfoldl '''' f z xs = foldr (\x y z -> y (f x z)) id xs z

... doma skúste foldr pomocou foldl ...