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

Funkcie a funkcionály

referečná transparentosť

Peter Borovanský

I-18

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

Zoznamová rekurzia

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

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

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 ڮ

Dôkaz - length (take n xs) = n

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.

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

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

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

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)

Rekapitulácia

videli sme:

- najznámejšie dva funkcionály: map, filter – ktoré poznáte aj z Pythonu
- quickCheck náhodne generujúci testy/kontrapríklady, Int, [Int], Int->Int
- množstvo ekvivalentných tvrdení, niektoré neekvivalentné...

Ak definujeme vlastnú dátovú štruktúru, ako využiť quickCheck, príklad:

data BVS t = Nil | Node (BVS t) t (BVS t) deriving(Show, Ord, 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
  - Ord – automaticky vygenerované funkcie <,>,...,min,max :: BVS t->BVS t->Bool

QuickCheck - Generator

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

class Arbitrary a where

    arbitrary :: Gen a

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

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

Generátory

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

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

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

Generator.hs

Generátory - zoznam

```
arbitraryListMax8Len :: Arbitrary a => Gen [a]
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] ] )
```

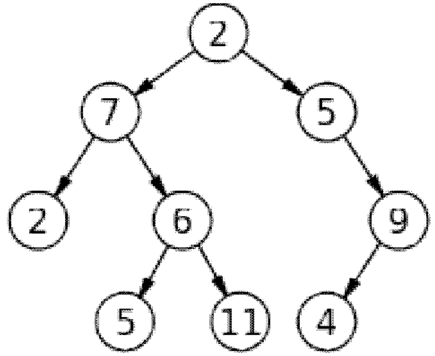
```
"?: " generate (arbitraryList::Gen [Int])
[-9,7,14,24,18,28,-4,0,22,12,-14]
```

```
mysized :: (Int -> Gen a) -> Gen a
mysized fg = fg 50
```

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

Generator.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)

Generator.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

-- nájdi v binárnom vyhľadávacom strome
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
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

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
```

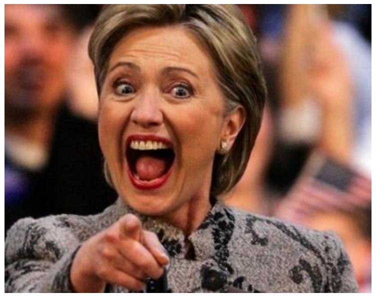
foldr ? ? xs = reverse xs

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)] ?? [a->a]
  akého je typu
  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 pom (\_ -> []) xs) n where
                    pom x h = \n -> if n == 0 then []
                                      else x:(h (n-1))
                    alebo
                    pom 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

$$f\ .\ \text{foldr}\ g1\ z1\ xs = \text{foldr}\ g2\ z2\ xs$$

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)

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

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 -> Integer) -> Integer -> 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

```
    :                    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
```

Zamyslime sa, ako z foldr urobíme foldl:

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 ...