

# 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)
```

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

Hypotéza (pre ľubovoľné n a xs) platí:

- $\text{length (take } n \text{ xs)} = n$
- $\text{length } \$ \text{ take } n \text{ xs} = n$
- $(\text{length} \cdot \text{take } n) \text{ xs} = n$

-- dolárová notácia

-- kompozícia funkcií z matematike .

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



# Dôkaz - $\text{length} (\text{take } n \text{ } xs) = n$

(matematická indukcia)

Indukcia (vzhľadom na dĺžku/štruktúru  $xs$ ):

-  **$xs = []$**

$$\text{length} (\text{take } n []) = 0$$

$$0 = 0$$

*č.b.t.d.*

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

$$\text{length} (\text{take } n (y:ys)) = n$$

$$\text{length} (y:\text{take } (n-1) ys) = n$$

$$1 + \text{length} (\text{take } (n-1) ys) = n$$

indukčný predpoklad,  $|ys| < |xs|$

$$1 + (n-1) = n$$

*č.b.t.d.*

Definície z predošlej strany:

$\text{take} \quad \quad \quad :: \text{Int} \rightarrow [a] \rightarrow [a]$

$\text{take } 0 \quad \quad = []$

$\text{take } \_ [] = []$

$\text{take } n (x:xs) = x : \text{take } (n-1) xs$

$\text{length} \quad \quad :: [a] \rightarrow \text{Int}$

$\text{length} [] = 0$

$\text{length } (x:xs) = 1 + \text{length } xs$



# 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

---

Podmienka: miesto písania

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

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 nástroj na hľadanie/odhaľovanie kontrapríkladov, kedy naše tvrdenie neplatí.

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  $f(n) = \text{if } n \bmod 2 == 0 \text{ then } n/2 \text{ else } 3n+1$ .

```
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.

# 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) ))) 😊





# Funkcia/predikát argumentom

- zober zo zoznamu tie prvky, ktoré spĺňajú podmienku (test)  
Booleovská podmienka príde ako argument funkcie a má typ  $(a \rightarrow \text{Bool})$ :

`filter`  $:: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

`filter p xs`  $= [x \mid x \leftarrow xs, p\ x]$

alternatívna definícia:

`filter p []`  $= []$

`filter p (x:xs)`  $= \text{if } p\ x \text{ then } x:(\text{filter } p\ xs) \text{ else } \text{filter } p\ xs$

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

vlastnosti (zväčša úplne zrejmé ?):

- `filter True xs`  $= xs$  ...  $[x \mid x \leftarrow xs, \text{True}] = [x \mid x \leftarrow xs] = xs$
- `filter False xs`  $= []$  ...  $[x \mid x \leftarrow xs, \text{False}] = []$
- `filter p1 (filter p2 xs)`  $= \text{filter } (p1 \ \&\& \ p2) \ xs$
- `(filter p1 xs) ++ (filter p2 xs)`  $= \text{filter } (p1 \ || \ p2) \ xs$

$$\begin{aligned}\text{filter } p [] &= [] \\ \text{filter } p (x:xs) &= \text{if } p \ x \text{ then } x:(\text{filter } p \ xs) \text{ else } \text{filter } p \ xs\end{aligned}$$

# Dôkaz

$\text{filter } p1 (\text{filter } p2 \ xs) = \text{filter } (p1 \ \&\& \ p2) \ xs$

Indukcia vzhľadom na parameter xs

- []  
L.S. =  $\text{filter } p1 (\text{filter } p2 []) = \text{filter } p1 [] = [] = \text{filter } (p1 \ \&\& \ p2) [] = \text{P.S.}$
- (x:xs)  
L.S. =  $\text{filter } p1 (\text{filter } p2 (x:xs)) = \dots \text{ definícia}$   
 $\text{filter } p1 (\text{if } p2 \ x \text{ then } x:(\text{filter } p2 \ xs) \text{ else } \text{filter } p2 \ xs) = \dots \text{ filter dnu cez if}$   
 $\text{if } p2 \ x \text{ then } \text{filter } p1 (x:(\text{filter } p2 \ xs)) \text{ else } \text{filter } p1 (\text{filter } p2 \ xs) = \dots \text{ indukcia}$   
 $\text{if } p2 \ x \text{ then } \text{filter } p1 (x:(\text{filter } p2 \ xs)) \text{ else } \text{filter } (p1 \ \&\& \ p2) \ xs = \dots \text{ definícia}$   
 $\text{if } p2 \ x \text{ then}$   
 $\quad \text{if } p1 \ x \text{ then } x:(\text{filter } p1 (\text{filter } p2 \ xs)) \text{ else } \text{filter } p1 (\text{filter } p2 \ xs)$   
 $\text{else } \text{filter } (p1 \ \&\& \ p2) \ xs = \dots \text{ 2 x indukcia}$   
 $\text{if } p2 \ x \text{ then}$   
 $\quad \text{if } p1 \ x \text{ then } x:(\text{filter } (p1 \ \&\& \ p2) \ xs) \text{ else } \text{filter } (p1 \ \&\& \ p2) \ xs$   
 $\text{else } \text{filter } (p1 \ \&\& \ p2) \ xs =$

$\text{filter } p [] = []$   
 $\text{filter } p (x:xs) = \text{if } p \ x \text{ then } x:(\text{filter } p \ xs) \text{ else } \text{filter } p \ xs$

# Dôkaz

$\text{filter } p1 (\text{filter } p2 \ xs) = \text{filter } (p1 \ \&\& \ p2) \ xs$

if  $p2 \ x$  then

if  $p1 \ x$  then  $x:(\text{filter } (p1 \ \&\& \ p2) \ xs)$  else  $\text{filter } (p1 \ \&\& \ p2) \ xs$

else  $\text{filter } (p1 \ \&\& \ p2) \ xs = \dots$  **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  $x:(\text{filter } (p1 \ \&\& \ p2) \ xs)$  else  $\text{filter } (p1 \ \&\& \ p2) \ xs = \dots$  **def.**

$\text{filter } (p1 \ \&\& \ p2) \ (x:xs) = \text{P.S.}$

*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 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>

# Funkcia argumentom

## map

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

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

`map f []`  $= []$

`map f (x:xs)`  $= f\ x : \text{map f xs}$

`map f xs`  $= [ f\ x \mid x \leftarrow xs ]$

- Príklady:

`map (+1) [1,2,3,4,5]`  $= [2,3,4,5,6]$

`map odd [1,2,3,4,5]`  $= [\text{True}, \text{False}, \text{True}, \text{False}, \text{True}]$

`and (map odd [1,2,3,4,5])`  $= \text{False}$

`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

- $\text{map id xs} = \text{xs}$  ☒  $\text{map id} = \text{id}$
- $\text{map (f.g) xs} = \text{map f (map g xs)}$  ☒  $\text{map f} . \text{map g} = \text{map (f.g)}$
- ~~■  $\text{head (map f xs)} = \text{f (head xs)}$  ☒  ~~$\text{head} . \text{map f} = \text{f} . \text{head}$~~~~
- ~~■  $\text{tail (map f xs)} = \text{map f (tail xs)}$  ☒  ~~$\text{tail} . \text{map f} = \text{map f} . \text{tail}$~~~~
- $\text{map f (xs ++ ys)} = \text{map f xs} ++ \text{map f ys}$  ☒
- $\text{length (map f xs)} = \text{length xs}$  ☒  $\text{length} . \text{map f} = \text{length}$
- $\text{map f (reverse xs)} = \text{reverse (map f xs)}$  ☒  $\text{map f} . \text{reverse} = \text{reverse} . \text{map f}$
- ~~■  $\text{sort (map f xs)} = \text{map f (sort xs)}$  ☒  ~~$\text{sort} . \text{map f} = \text{map f} . \text{sort}$~~~~
- $\text{map f (concat xss)} = \text{concat (map (map f) xss)}$  ☒

$\text{map f} . \text{concat} = \text{concat} . \text{map (map f)}$

$\text{concat} :: [[a]] \rightarrow [a]$

$\text{concat []} = []$

$\text{concat (xs:xss)} = \text{xs} ++ \text{concat xss}$



$\text{concat} [[1], [2,3], [4,5,6], []] = [1,2,3,4,5,6]$



# 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:

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

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

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

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

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

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





# Quíz - prémia

nájdite pravdivé a zdôvodnite

---

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

(podobné bolo 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]]
```

# Deliteľnosť 11

(bolo na cvičeniach)

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

JaroF:

foldr (-) 0 \$

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.

Rule for Divisibility by 11

10,813?

10,813

1+8+3= 12

0+1= 1

12 - 1 = 11

11 ÷ 11





# 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
```



- Introduction to QuickCheck  
[https://wiki.haskell.org/Introduction to QuickCheck2](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 Testing of 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>



# 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)



# Definované typy

---

Ak definujeme vlastnú dátovú štruktúru, ako využiť quickCheck ?

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

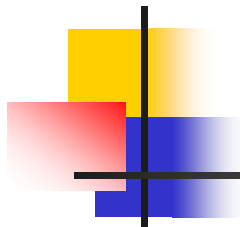
- dva konštruktory **Nil** a **Node** \_ \_ \_
- deriving popisuje patričnosť do triedy class - (resp. implements interface)
  - Show – automaticky vygenerovaná funkcia show :: BVS t ->String
  - Eq – automaticky vygenerované funkcie ==,/= :: BVS t -> BVS t -> Bool

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 ?



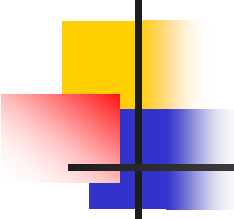


# Java a Reflexivita

(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](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

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
```



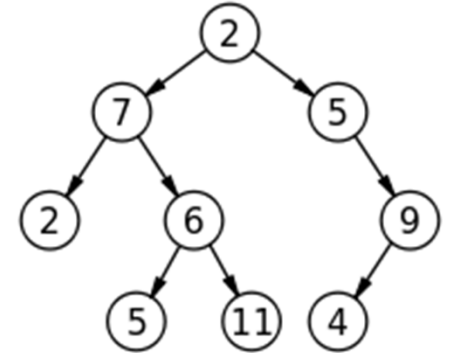
# Generátory - zoznam

```
arbitraryListMax8Len :: Arbitrary a => Gen [a]    -- náhodný zoznam len <= 8
arbitraryListMax8Len =                           "?: " 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 fg = fg 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)
]
```



# 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] -- opäť riešenie Jara/Turbo
```

```
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 ??



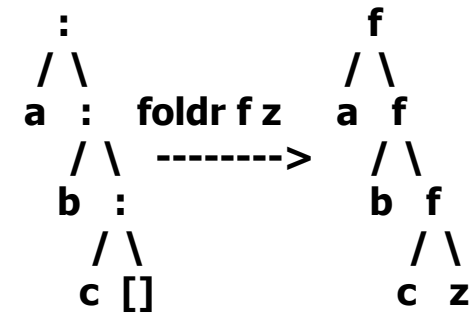
# Haskell – foldr

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



-- `g` je vnorená lokálna funkcia

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

`5050`

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

`foldr f z = g`

where `g []` = `z`

`g (x:xs)` = `f x (g xs)`

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

`4321`



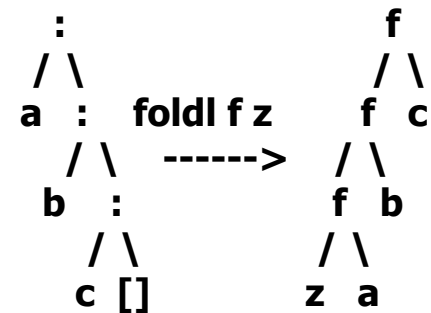
# Haskell – foldl

`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]`

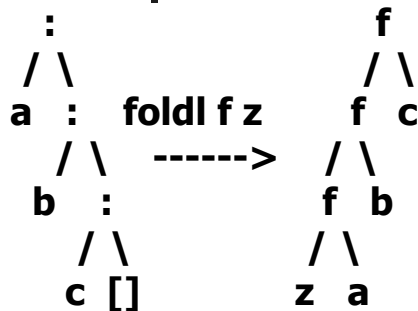
- `foldr (-) 0 [1..100] =`

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

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

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

# Kvíz

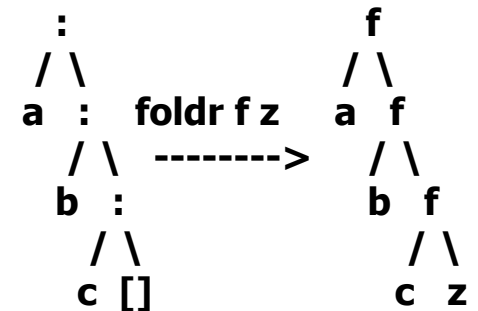


$$\text{foldr } (:) [] \text{ xs} = \text{xs}$$

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

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

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



<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  $[\backslash x \rightarrow x+1, \backslash x \rightarrow x+2, \backslash x \rightarrow x*3]$
- čo je foldr  $(.)$  id  $[(+1), (+2), (*3)]$  ??  
akého je typu  $[a \rightarrow a]$   
foldr  $(.)$  id  $[(+1), (+2), (*3)]$  100 303  
foldl  $(.)$  id  $[(+1), (+2), (*3)]$  100 ???

lebo skladanie fcií je asociatívne:

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





# Zákon fúzie – pre foldr

Fussion Law:

Nech  $g_1, g_2$  sú binárne funkcie,  $z_1, z_2$  konštanty

Ak pre funkciu  $f$  platí :

$$f\ z_1 = z_2 \ \&\& \ f\ (g_1\ a\ b) = g_2\ a\ (f\ b)$$

potom platí

$$f . \text{foldr } g_1\ z_1\ xs = \text{foldr } g_2\ z_2\ xs$$

Príklad použitia Fussion Law:

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

Dôkaz (pomocou Fussion Law): overíme predpoklady

čo je čo ?!:

$$f = (n^*),\ z_1 = z_2 = 0,\ g_1 = (+),\ g_2 = (+)\cdot(n^*)$$

treba overiť:

- $(n^*)\ 0 = 0$  ☒
- $L.S. = (n^*)\ (a+b) = (n^*a + n^*b) = (+)\cdot(n^*)\ a\ ((n^*)\ b) = P.S.$  ☒

# Vlastnosti

Acid Rain (fold/build/deforestation theorem)



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

$\underbrace{\hspace{10em}}_{t \rightarrow u}$

Intuícia: Keď máme vytvoriť zoznam pomocou funkcie  $g$  zo zoznamových konštruktorov  $(:) []$ , na ktorý následne pustíme  $\text{foldr}$ , 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  $f :: x \rightarrow u \rightarrow u$ ,  $\text{foldr } f \ z :: [x] \rightarrow u$ .

Ľavá strana:  $([x] \rightarrow u) \cdot (t \rightarrow [x])$  výsledkom je typ  $t \rightarrow u$

Pravá strana:  $g :: (x \rightarrow u \rightarrow u) \rightarrow u \rightarrow (t \rightarrow u)$

$$\text{foldr } f \ z \ . \ g \ (:) \ [] = g \ f \ z$$

$$\text{length} \ . \ \text{map } \_ = \text{length}$$

$\text{map} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$

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

$-- (:) \ . \ h \ a \ as = (:) (h \ a \ as) = h \ a : as$

$= \underbrace{(\lambda x \rightarrow \lambda y \rightarrow \text{foldr } (x \ . \ h) \ y)}_g (:) \ []$

$\text{length} :: [a] \rightarrow \text{Int}$

$\text{length} = \text{foldr } \underbrace{(\lambda \_ \rightarrow \lambda n \rightarrow n+1)}_f \underbrace{0}_z$

$\text{length} \ . \ \text{map } h = \dots \text{length}$

L.S.  $= (\text{foldr } (\lambda \_ \rightarrow \lambda n \rightarrow n+1) \ 0) \ . \ (\text{foldr } ((:) \ . \ h) \ []) =$

$= \text{podľa Acid Rain theorem } (f = (\lambda \_ \rightarrow \lambda n \rightarrow n+1), z = 0, \text{ ale čo je } g ? \dots)$

$g \ x \ y = (\text{foldr } (x \ . \ h) \ y)$

$g \ f \ z = (\text{foldr } (f \ . \ h) \ z) = \text{foldr } ((\lambda \_ \rightarrow \lambda n \rightarrow n+1) \ . \ h) \ 0 =$

$\text{foldr } ((\lambda \_ \rightarrow \lambda n \rightarrow n+1)) \ 0 = \text{length} = \text{P.S.}$

lebo (tento krok pomalšie):

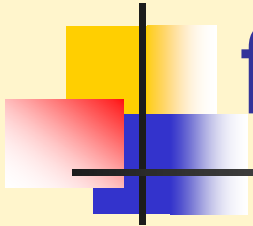
$((\lambda \_ \rightarrow \lambda n \rightarrow n+1) \ . \ h) \ x \ y = (\lambda \_ \rightarrow \lambda n \rightarrow n+1) \ (h \ x) \ y = (\lambda n \rightarrow n+1) \ y = y+1$

$$g \ h \ w \ n = h \ n \ (g \ h \ w \ (n-1))$$

933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000

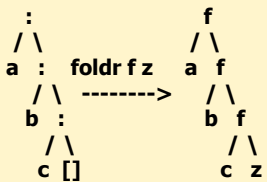
93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000

$$g' n = n : (g' (n-1))$$
$$g''_n = n * (g''_{n-1})$$

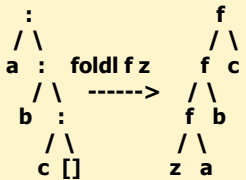


# foldr a foldl pre pokročilejších

definujte foldl pomocou foldr, alebo naopak:



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)



## ■ 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:

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

- $y = \text{myfoldl } f \text{ [b,c]} = \lambda z \rightarrow f (f z b) c$

nech  $x$  je ďalší prvok zoznamu, t.j.

- $x = a$

ako musí vyzerat' funkcia  $?$ , ktorou fold-r-ujeme, aby sme dostali  $\text{myfoldl } f \text{ [a,b,c]} = \lambda z' \rightarrow f (f (f z' a) b) c = ? \ x \ y$

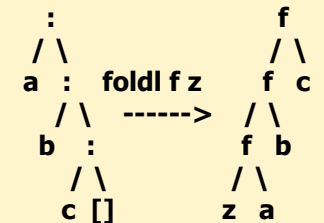
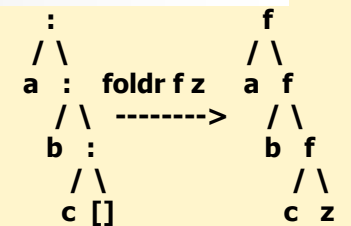
- $? = (\lambda x \ y \ z' \rightarrow y (f z' x))$

dosad'me:

- $(\lambda z' \rightarrow (\lambda z \rightarrow f (f z b) c) (f z' a)) =$

- $(\lambda z' \rightarrow f (f (f z' a) b) c) =$

- $\lambda z' \rightarrow f (f (f z' a) b) c$



# Pre tých, čo neveria, fakt posledný krát

$$? = (\backslash x\ y\ z' \rightarrow y\ (f\ z'\ x))$$

$\text{myfoldl}''' f\ xs\ z = \text{foldr}\ (\backslash x\ y\ z \rightarrow y\ (f\ z\ x))\ id\ xs\ z$

- $\text{myfoldl}''' f\ [] = id$
- $\text{myfoldl}''' f\ [c] = (\backslash x\ y\ z \rightarrow y\ (f\ z\ x))\ c\ id = \backslash z \rightarrow f\ z\ c$
- $\text{myfoldl}''' f\ [b,c] = (\backslash x\ y\ z \rightarrow y\ (f\ z\ x))\ b\ (\backslash w \rightarrow f\ w\ c) =$   
 $\backslash z \rightarrow (\backslash w \rightarrow f\ w\ c)\ (f\ z\ b) =$   
 $\backslash z \rightarrow f\ (f\ z\ b)\ c$
- $\text{myfoldl}''' f\ [a,b,c] = (\backslash x\ y\ z \rightarrow y\ (f\ z\ x))\ a\ (\backslash w \rightarrow f\ (f\ w\ b)\ c) =$   
 $\backslash z \rightarrow (\backslash w \rightarrow f\ (f\ w\ b)\ c)\ (f\ z\ a) =$   
 $\backslash z \rightarrow f\ (f\ (f\ z\ a)\ b)\ c$
- $\text{myfoldl}''' f\ z\ xs = \text{foldr}\ (\backslash x\ y\ z \rightarrow y\ (f\ x\ z))\ id\ xs\ z$

... doma skúste foldr pomocou foldl ...