Funkcionálne programovanie

Peter Borovanský, KAI, I-18,
borovan(a)ii.fmph.uniba.sk

H.P.Barendregt:

- funkcionálny program pozostáva z výrazu, ktorý je *algoritmus* a **zároveň** jeho *vstup*

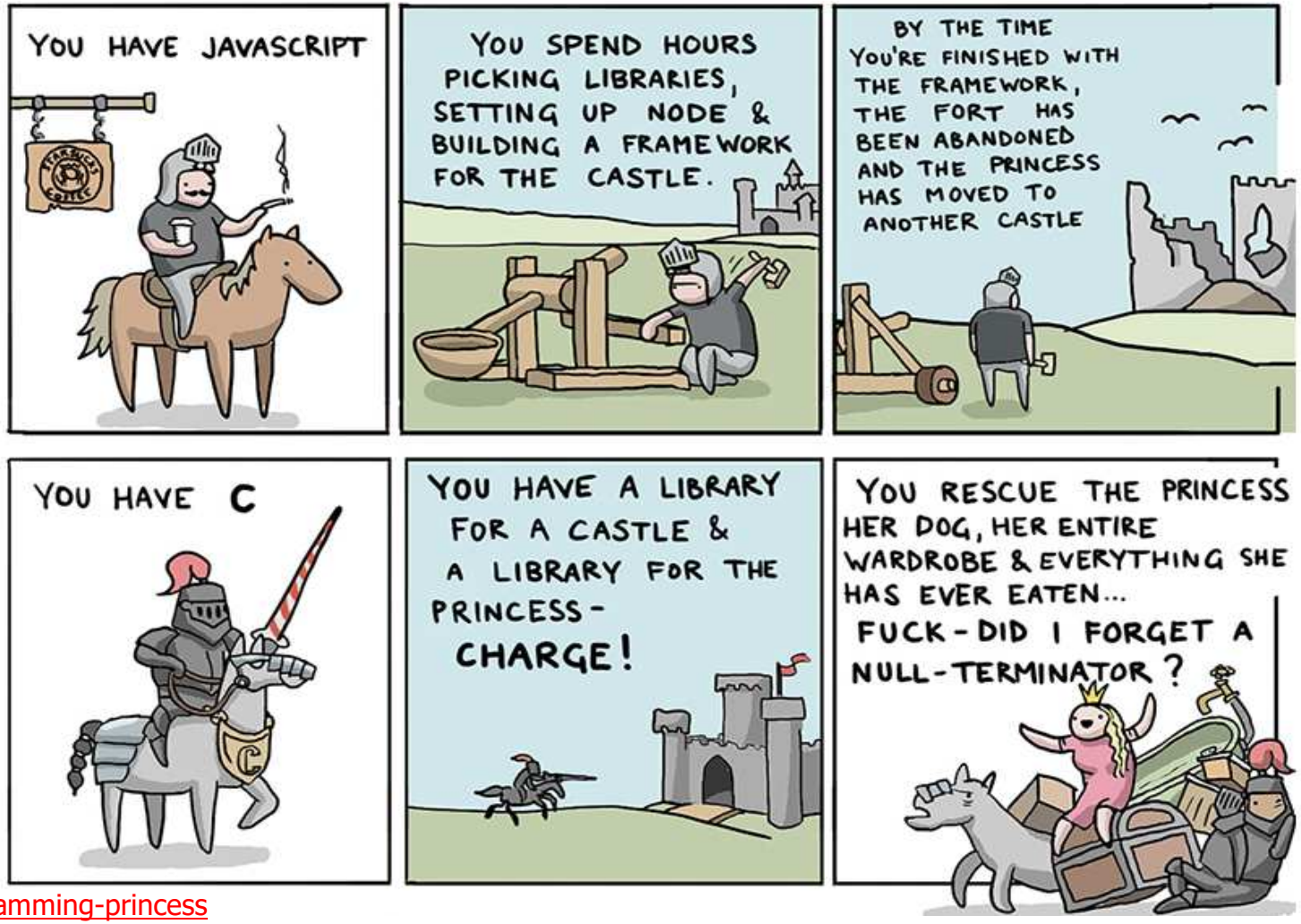
- tento výraz sa redukuje (derivuje) *prepisovacími pravidlami*
- redukcia nahrádza podčasti inými (podľa istých pravidiel)
- redukcia sa vykonáva, kým sa dá....

- výsledný výraz *(normálna forma)*, je výsledkom výpočtu

GIT THE PRINCESS!

HOW TO SAVE THE PRINCESS USING 8 PROGRAMMING LANGUAGES

BY toggl Goon Squad



https://toggl.com/programming-princess

900,930 REPUTATION

487 6535 7461

Jon Skeet

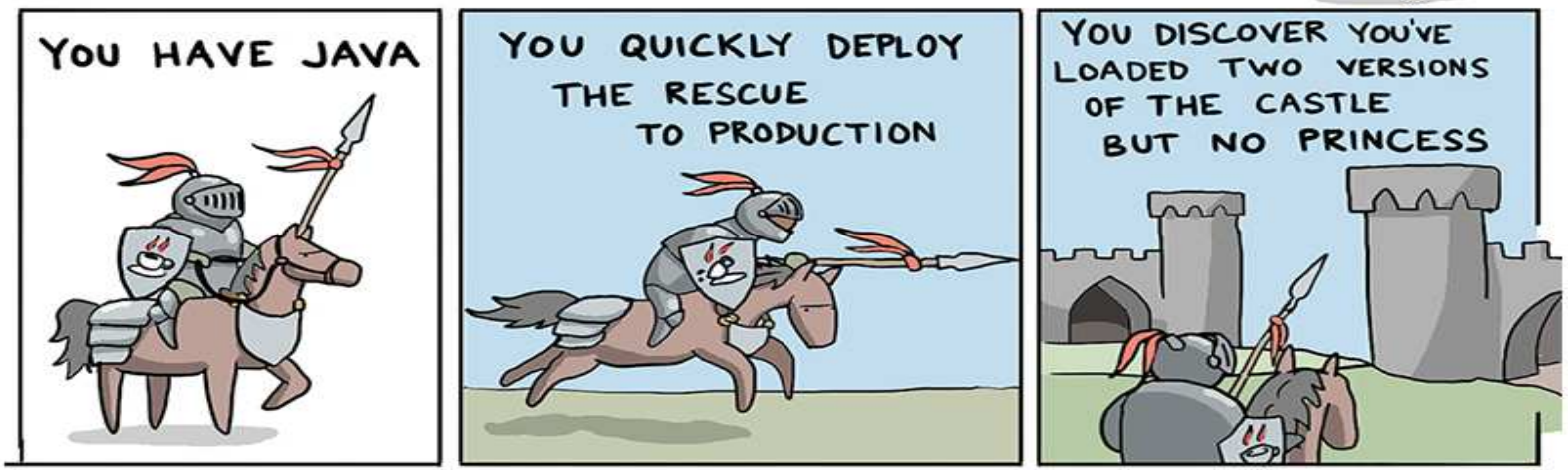
top 0.01% overall

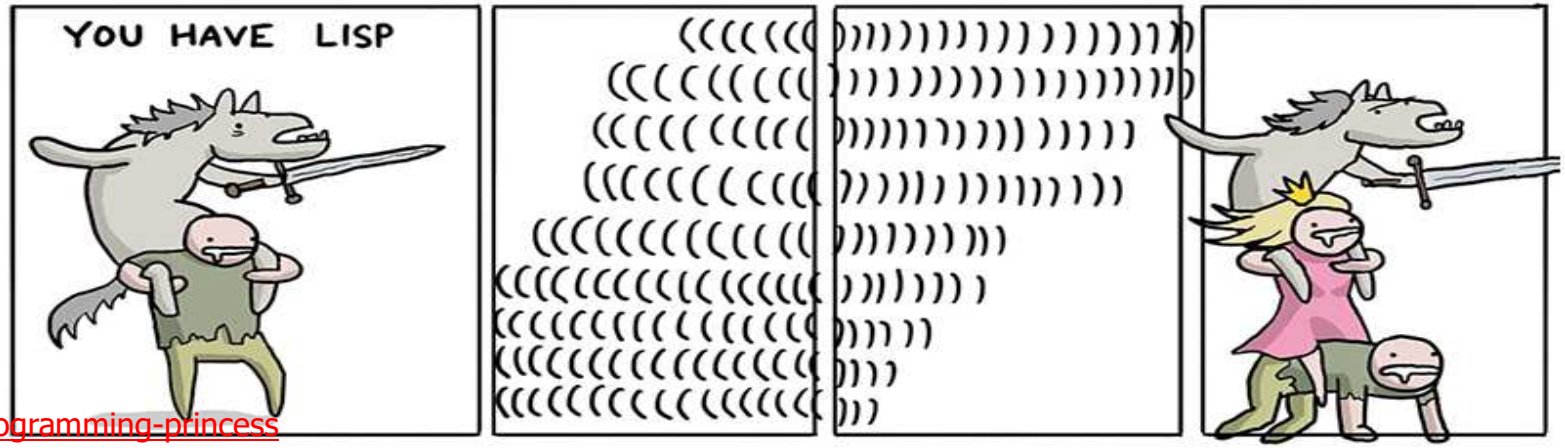
Senior Software Engineer at Google

Author of C# in Depth.
Currently a software engineer at Google, London.
Usually a Microsoft MVP (C#, 2003-2010, 2011-)

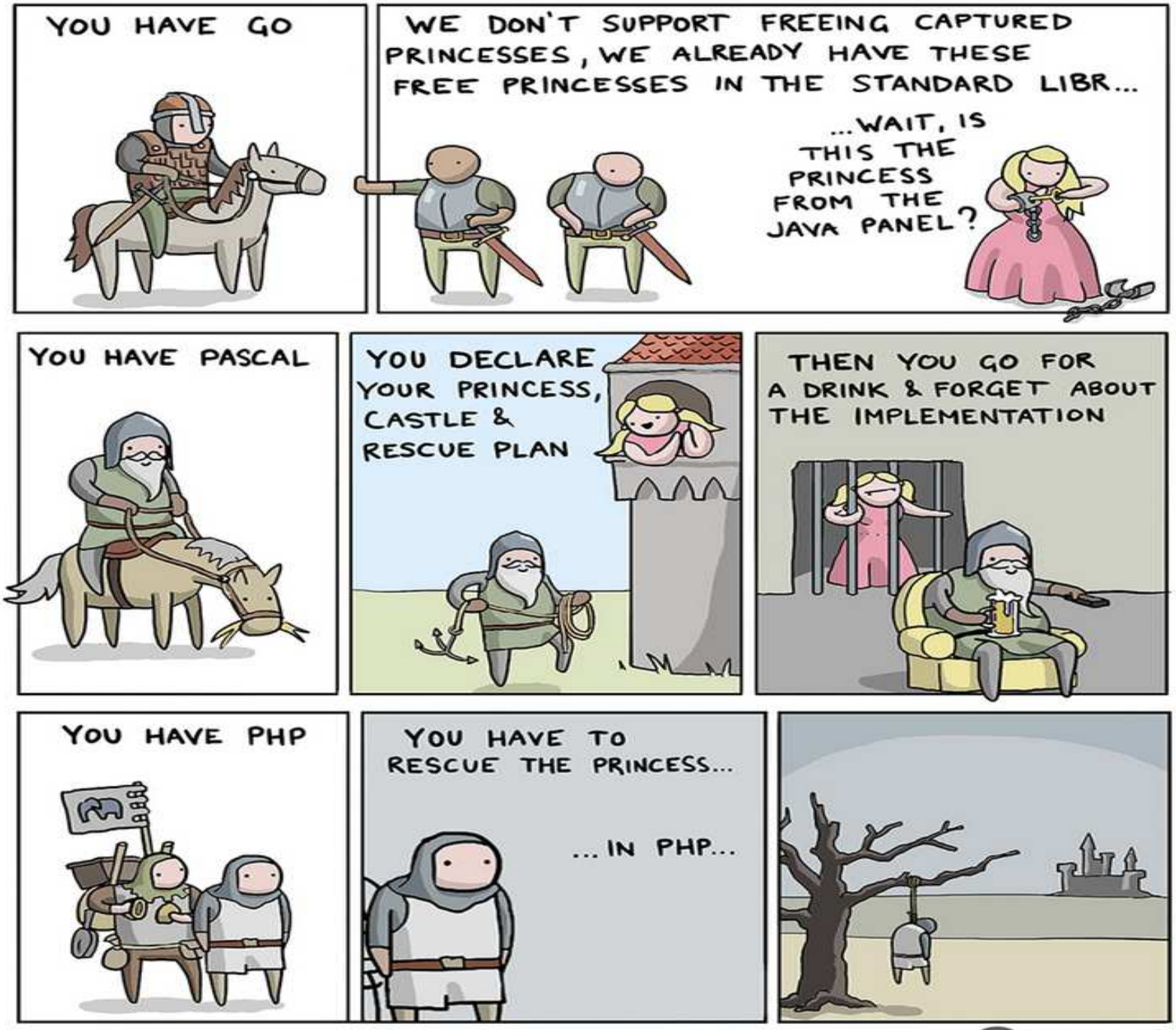
Sites:

- C# in Depth
- Coding blog
- C# articles
- Twitter updates (@jonskeet)
- Google+ profile



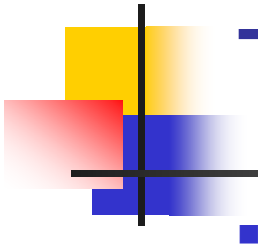


https://toggl.com/programming-princess



https://toggl.com/programming-princess

Trochu z histórie FP

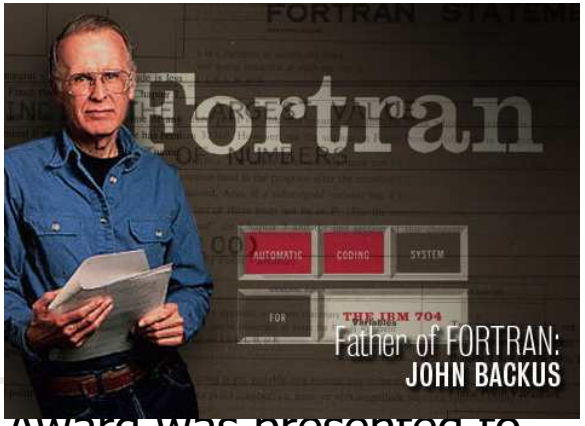


- 1930, Alonso Church, lambda calculus
  - teoretický základ FP
  - kalkul funkcií: abstrakcia, aplikácia, kompozícia
  - Princeton: A.Church, A.Turing, J. von Neumann, K.Gödel - skúmajú formálne modely výpočtov
  - éra: WWII, prvý von Neumanovský počítač: Mark I (IBM), balistické tabuľky
- 1958, Haskell B.Curry, logika kombinátorov
  - alternatívny pohľad na funkcie, menej známy a populárny
  - „premenné vôbec nepotrebujeme"
- 1958, LISP, John McCarthy
  - implementácia lambda kalkulu na „von Neumanovskom HW"

Niektoré jazyky FP:

- 1.frakcia: Lisp, Common Lisp, ..., Scheme (MIT,DrScheme,Racket)
- 2.frakcia: Miranda, Gofer, Erlang, Clean, Haskell Platform(Hugs),

John Backus



- 40 years ago, on October 17th, 1977, the Turing Award was presented to John Backus for his contribution to the design of high-level programming systems, most notably the Fortran programming language.
- He gave a lecture entitled Can programming be liberated from the Von Neumann style? in which he criticized some of the mainstream languages of the day, including Fortran, for their shortcomings. He also proposed an alternative: a **functional style of programming**.

BNF (Backus-Naur Form)

```
(2.0 * PI) / n
<expression>  ::=  <expression> + <term>
              |    <expression> - <term>
              |    <term>
<term>        ::=  <term> * <factor>
              |    <term> / <factor>
              |    <factor>
<factor>      ::=  number
              |    name
              |    ( <expression> )
```

CSE 341, S. Tanimoto — Concepts 1- 5

Viac:

- https://hackernoon.com/practical-functional-programming-6d7932abc58b
- **An Adequate Introduction to Functional Programming**

https://dev.to/mr_b/an-adequate-introduction-to-functional-programming-1gcl?fbclid=IwAR0BNSSE0h2Q_tAc0GSoSv1-fcPDLkcfLFtw47JDE_hzrjsBOOA0FNA435w

Literatúra

- Henderson, Peter (1980): *Functional Programming: Application and Implementation,* Prentice-Hall International
- R.Bird: Introduction Functional Programming using Haskell
- P.Hudak, J.Peterson, J.Fasel: *Gentle Introduction to Haskell*
- H.Daume: *Yet Another Haskell Tutorial*
- D.Medak, G.Navratil: *Haskell-Tutorial*
- Peyton-Jones, Simon (1987): *The Implementation of Functional Programming Languages,* Prentice-Hall International
- Thompson, Simon (1999): *The Craft of Functional Programming*, Addison-Wesley
- Hughes, John (1984): *Why Functional Programming Matters*
- *Fokker, Jeroen:Functional Programming alebo Functional Parsers*
- Wadler, Phil: *Monads for functional programming*

Frequently Asked Questions (comp.lang.functional)

1960 LISP

- LISP je rekurzívny jazyk
- LISP je vhodný na list-processing
- LISP používa dynamickú alokáciu pamäte, GC
- LISP je skoro beztypový jazyk
- LISP používal dynamic scoping
- LISP má globálne premenné, priradenie, cykly a pod.
  - ale nič z toho vám neukážem ☺
- LISP je vhodný na prototypovanie a je *všelikde*
- Scheme je LISP dneška, má viacero implementácií, napr. DrRacket

Scheme - syntax

<Expr> ::= <Const> |
<Ident> |
(<Expr0> <Expr1> … <Exprn>**)** |
(lambda (<Ident1>…<Identn>**)** <Expr>**)** |
(define <Ident> <Expr>**)**

definícia funkcie:

```
(define gcd
  (lambda (a b)
    (if (= a b)
        a
        (if (> a b)
            (gcd (- a b) b)
            (gcd a (- b a))))))
```

volanie funkcie:

```
(gcd 12 18)
        6
```

Rekurzia na číslach


```
(define fac (lambda (n)
   (if (= n 0)
      1
      (* n (fac (- n 1))))))
```

```
(fac 100)
     933262....000
```

```
(define fib (lambda (n)
   (if (= n 0)
      0
      (if (= n 1)
         1
         (+ (fib (- n 1)) (fib (- n 2)))))))
```

```
(fib 10)
     55
```

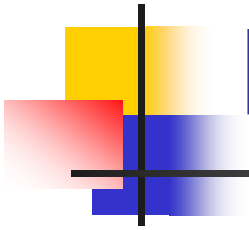
```
(define ack (lambda (m n)
   (if (= m 0)
      (+ n 1)
      (if (= n 0)
         (ack (- m 1) 1)
         (ack (- m 1) (ack m (- n 1)))))))
```

```
(ack 3 3)
          61
```

```
(define prime (lambda (n k)
   (if (> (* k k) n)
      #t
      (if (= (remainder n k) 0)
         #f
         (prime n (+ k 1))))))

(define isPrime?(lambda (n)
     (and (> n 1) (prime n 2))))
```

pozbierať *dobré myšlienky* FP
výskumu použiteľné pre výuku



Haskell (1998)

- nemá globálne premenné
- nemá cykly
- nemá side-effect (ani I/O v klasickom zmysle)
- referenčná transparentnosť
  funkcia vždy na rovnakých argumentoch dá rovnaký výsledok
- je striktne typovaný, aj keď typy nevyžaduje (ale ich inferuje)
- je lenivý (v spôsobe výpočtu) – počíta len to čo „treba“

FPCA, 1988

A History of Haskell: Being LazyWith Class

1. *It should be suitable for teaching, research, and applications, including building large systems.*
2. *It should be completely described via the publication of a formal syntax and semantics.*
3. *It should be freely available.* Anyone should be permitted to implement the language and distribute it to whomever they please.
4. *It should be usable as a basis for further language research.*
5. *It should be based on ideas that enjoy a wide consensus.*
6. *It should reduce unnecessary diversity in functional programming languages.* More specifically, we initially agreed to base it on an existing language, namely OL.

https://www.google.sk/url?sa=t&rct=j&q=&esrc=s&source=web&cd=6&ved=0ahUKEwicvoLuuN7XAhXQDewKHR0ZDnYQFghOMAU&url=http%3A%2F%2Fhaskell.cs.yale.edu%2Fwp-content%2Fuploads%2F2011%2F02%2Fhistory.pdf&usg=AOvVaw0KBkRMnd-Wo5HqVb3VJFRj

Prvá funkcia v Haskelli

- \y -> y+1
  λ y.(+ y 1)
  add1 y = y + 1

Mená funkcií a premenných malým písmenom

n-árna funkcia je n-krát unárna, nie však funkcia s argumentom n-tice

- add x y = x + y \x -> \y -> (x+y) ☺
- add' (x,y) = x+y \(x,y) -> (x+y) ☹

add1 = add 1 \y -> (1+y)

add1 y = y + 1

Zdrojový kód
haskell1.hs

Funkcie a funkčný typ

- **pomenované** funkcie

```
odd    :: Int -> Bool
odd 5  = True
```

- **anonymné** funkcie:

```
\x -> x*x
```

- pomenovanie (definícia) funkcie:

```
f = \x -> x*x   je rovnocenný zápis s    f x = x*x

g = add 1       = (\x -> \y -> x+y) 1    = \y -> 1+x
```

- **funkčný typ**: t_1 -> t_2 (funkcia z typu t_1 do typu t_2)

asociativita typového operátora **doprava**:

- t_1 -> t_2 -> t_3 -> t_4 znamená t_1 -> (t_2 -> (t_3 -> t_4))

Skladanie funkcií

- aplikácia funkcií je **l'avo-asociatívna a nekomutatívna**:
  f g 5 = (f g) 5 != f (g 5) = (f . g) 5
  != g f 5 !!! zátvorkujte, zátvorkujte, zátvorkujte !!!

- operátor . znamená skladanie funkcií
  (.) :: (u->v) -> (t->u) -> (t->v),
  alebo zjednodušene
  (.) :: (t->t) -> (t->t) -> (t->t)

- Príklady:

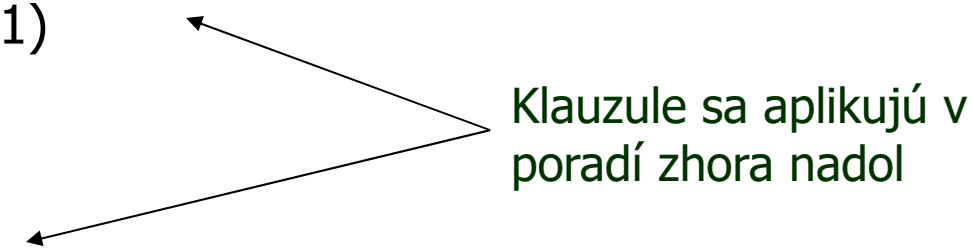
	inak zapísané
dvakrat f = f . f	dvakrat f x = f (f x)
naDruhu x = x*x	
naStvrtu = dvakrat naDruhu	
naStvrtu = naDruhu . naDruhu	naStvrtu x= (naDruhu.naDruhu) x
posledny = head . reverse	posledny x = head (reverse x)

Číselné funkcie

Faktoriál:

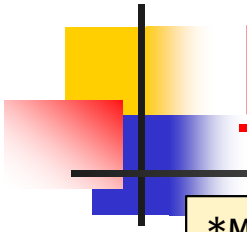
- fac n = **if** n == 0 **then** 1 **else** n*fac(n - 1)

- fac' 0 = 1
  fac' n = n*fac'(n-1)



Najväčší spoločný deliteľ

- nsd 0 0 = error "nsd 0 0 nie je definovany"
  nsd x 0 = x -- klauzálna definícia
  nsd 0 y = y -- klauzule sa skúšajú v textovom poradí
  nsd x y = nsd y (x `rem` y) -- x>0, y>0, `rem`=`mod`=%



http://www.haskell.org/

```
*Main> :set +s
*Main> fac'' 1000
402387260077093773543702433923003985719374864210714632 ...
000000000000000000000000
(0.07 secs, 2,576,296 bytes)

*Main> if fac'' 100000 > 0 then 1 else 0
1
(8.60 secs, 11,295,332,968 bytes)

*Main> let f = fac'' 100000 in 1
1
(0.01 secs, 65,064 bytes)

*Main> let f = fac'' 100000 in 1+f
... 000000000000000000000000000001
(19.59 secs, 11,615,113,024 bytes)
```

Podmienky (if-then-else, switch-case)

- ```
  fac''' n | n <  0        = error "nedefinovane"
           | n == 0        = 1
           | n >  0        = product [1..n]
  ```

- power2 :: Int -> Int *(typ funkcie)*

  -- typ funkcie nemusíme zadať, ak si ho vie systém odvodiť sám.
  -- Avšak s typom funkcie si ujasníme, čo vlastne definujeme. -----
  -- Preto ho definujme!

```
power2 n
        | n==0          = 1
        | n>0           = 2 * power2 (n-1)
        | otherwise     = error "nedefinovane"
```

```
Main> fac''' 5
120
Main> power2 5
32
Main> power2 (-3)
Program error:
nedefinovane
```
```

where blok
(priradenie do lokálnej premennej)

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

```
quadsolve a b c | delta < 0     = error "complex roots"
                | delta == 0    = [-b/(2*a)]
                | delta > 0     = [-b/(2*a) + radix/(2*a),
                                   -b/(2*a) - radix/(2*a)]
                where
                    delta = b*b - 4*a*c
                    radix = sqrt delta
```

pod-výrazy sa počítajú zhora nadol

```
Main> quadsolve 1 (-2) 1
[1.0]
Main> quadsolve 1 (-3) 2
[2.0,1.0]
Main> quadsolve 1 0 1
[Program error: complex roots]
Main> :type quadsolve
quadsolve :: f -> f -> f -> [f]
Main> :type error
error :: String -> [f]
```

Logaritmicky x^n

`mod` = `rem` - zvyšok po delení
`div` = `quot` - celočís.podiel'
`divMod`- (div,mod)

Vypočítajte x^n s logaritmickým počtom násobení:

Matematický zápis: $x^n = x^{2^{(n/2)}}$ ak n je párne
$x^n = x * x^{n-1}$ ak n je nepárne

```
power              :: Int -> Int -> Int
power x n          | n == 0                = 1
                   | (n `mod` 2 == 0)      = power (x*x) (n `div` 2)
                   | otherwise             = x*power x (n-1)
```

alebo: $x^n = x^{(n/2)^2}$ ak n je párne

```
power'             :: Float -> Int -> Float
power' x n         | n == 0                = 1
                   | (n `mod` 2 == 0)      = pom*pom
                   | otherwise             = x*power' x (n-1)
                   where   pom = power' x (n `div` 2)
-- alebo ešte inak
                   | (n `rem` 2 == 0)      = (power' x (n `div` 2))^2
```

Základné typy

Základné typy a ich konštanty:

- - 5 :: Int, (59182717273930281293 :: Integer)
  (máme `mod`, `div`, odd, even …)
- - "retazec" :: String = [Char], 'a' :: Char
- - True, False :: Bool (máme &&, ||, not – používajme ich)

n-tice:

(False, 3.14) :: (Bool, Double) (pre 2-ice máme fst (False,3.14)=False,
snd (False,3.14)=3.14)

Výrazy:

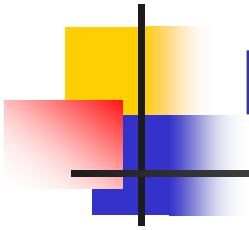
- if *Bool* then *t* else *t* :: *t* – typy then a else musia byť rovnaké

Príklad:

- 1 - if n `mod` 2 == 0 then 1 else 0
- if if n > 2 then n > 3 else n < 2 then 'a' else 'b' :: Char

- if n > 2 then if n > 3 then 4 else 3
  else if n > 1 then 2 else 1 :: Int

Zoznam je to, čo
má hlavu a chvost



n-tice a zoznamy

- n-tice

  (t_1, t_2, ..., t_n**)** n >= 2

  Konštanty:

  (5, False) :: (Int, Bool)

  (5, (False, 3.14)) :: (Int, (Bool, Double)) != (Int, Bool, Double)

- zoznamy

  konštruktory: h**:**t, **[]**

  konštanta: **[**1,2,3**]**

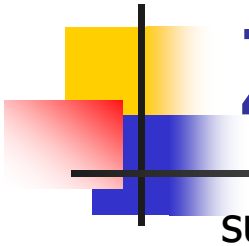
  [t**]**

  napr. [Int], [Char]

  vieme zapísať konštanty *typu* zoznam a poznáme konvencie

  1 : 2 : 3 : [] = [1,2,3]

V Haskelli nie sú polia, preto sa musíme naučiť narábať so zoznamami, ako primárnou dátovou štruktúrou



Zoznamy sú homogénne

sú vždy homogénne (na rozdieľ napr. od Lispu a Pythonu) vždy sú typu
List<t> = [t]

```
[2,3,4,5]          :: [Int],
[False, True]      :: [Bool]
```

- konštruktory x:xs, []

```
1:2:3:[]                          [1,2,3]
0:[1,2,3]                         [0,1,2,3]
1:[2,3,4] = 1:2:[3,4] = 1:2:3:[4] = 1:2:3:4:[]
```

- základné funkcie:

```
head :: [t] -> t          head [1,2,3] = 1
tail :: [t] -> [t]        tail [1,2,3] = [2,3]
null :: [t] -> Bool       null [1,2,3] = False
```

Najčastejšie operácie

[1,2] je 2-prvkový zoznam
(x:xs) je zoznam s hlavou x::t
a chvostom xs::[t]
[x:xs] je 1-prvkový zoznam
typu [[t]] obsahujúci (x:xs)

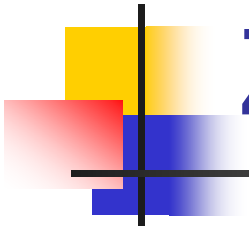
- zreťazenie append (++) :: [t] -> [t] -> [t]
  [1,2,3] ++ [4,5] = [1,2,3,4,5]
  ["Mon","Tue","Wed","Thur","Fri"] ++ ["Sat","Sun"]
  ["Mon","Tue","Wed","Thur","Fri","Sat","Sun"]

- priamy prístup k prvkom zoznamu !!
  [0,1,2,3]!!2 = 2
  indexovanie (od 0) (!!): [t] -> Int -> t
  [1,2,3]!!0 = 1

- aritmetické postupnosti ..
  [1..5] [1,2,3,4,5]
  [1,3..10] [1,3,5,7,9]

Cheat sheets:

- https://www.programming-idioms.org/cheatsheet/Haskell
- https://cheatsheet.codeslower.com/CheatSheet.pdf
- https://matela.com.br/pub/cheat-sheets/haskell-ucs-0.4.pdf
- https://github.com/paradigmy/Kod/blob/master/CV06/ListBasicsForDummies.hs



Zoznamová rekurzia 1

- len :: [a] -> Int -- polymorfická funkcia

```
len []          = 0          -- vypočíta dĺžku zoznamu
len (z:zs)      = 1 + len zs
```

- selectEven :: [Int] -> [Int] -- vyberie párne prvky zo zoznamu

```
selectEven [] = []
selectEven (x:xs)
                | even x      = x : selectEven xs
                | otherwise   =     selectEven xs
```

```
Main> len [1..4]
4
Main> selectEven [1..10]
[2,4,6,8,10]
```

Zoznamová rekurzia 2

```
-- ++
append              :: [a] -> [a] -> [a] -- zreťazenie zoznamov rovnakého typu
append  [] ys       = ys                 -- triviálny prípad
append  (x:xs) ys   = x:(append xs ys)   -- rekurzívne volanie
```

```
> append [1,2] [3,4]
[1,2,3,4]
```

```
rev                 :: [a] -> [a]        -- otočenie zoznamu od konca
rev []              = []                 -- triviálny prípad
rev(x:xs)           = (rev xs) ++ [x]    -- rekurzívne volanie,
                                            append [x] na koniec
```

```
> rev [1..4]
[4,3,2,1]
```

```
-- iteratívny reverse                    -- otočenie ale iteratívne
reverse xs          = rev' xs []
rev' []    ys       = ys
rev' (x:xs) ys      = rev' xs (x:ys)
```

```
> reverse [1..4]
[4,3,2,1]
```

Zip

Definujme binárnu funkciu *spoj*, ktorá spojí dva rovnako dlhé zoznamy do jedného zoznamu dvojíc, prvý s prvým, druhý s druhým, a t.ď.

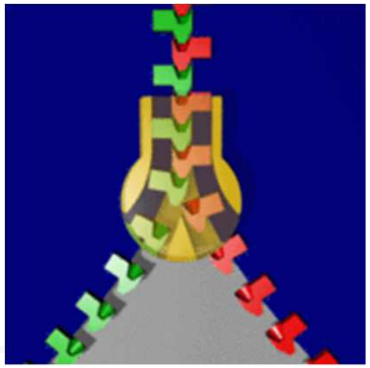
- spoj:: [a] -> [b] -> [(a,b)]

```
spoj (x:xs) (y:ys)      = (x,y) : spoj xs ys
spoj (x:xs) []          = []
spoj []     zs          = []
```

```
Main> spoj [1,2,3] ["a","b","c"]
[(1,"a"),(2,"b"),(3,"c")]
```

Táto funkcia sa štandardne volá zip.

Unzip



Definujme unárnu funkciu *rozpoj*, ktorá takto zozipsovaný zoznam rozpojí na dva zoznamy.

Funkcia **nemôže** vrátiť dve hodnoty, ale môže vrátiť dvojicu hodnot.

- rozpoj :: [(a,b)] -> ([a],[b])

```
Main> rozpoj
[(1,"a"),(2,"b"),(3,"c")]
([1,2,3],["a","b","c"])
```

```
rozpoj []                = ([],[])
rozpoj ((x,y):ps)        = (x:xs,y:ys)
                             where
                             (xs,ys) = rozpoj ps
```

Táto funkcia sa štandardne volá unzip

dvojica ako pattern

```
rozpoj ((x,y):ps)  = let (xs,ys) = rozpoj ps in (x:xs,y:ys)
```

Syntax – let-in

- rozpoj :: [(a,b)] -> ([a],[b])

```
rozpoj []           = ([],[])
rozpoj ((x,y):ps)   = let (xs,ys) = rozpoj ps
                      in (x:xs,y:ys)
```

- **let** $pattern_1$ = $výraz_1$;
  ...
  $pattern_n$ = $výraz_n$
  in výraz

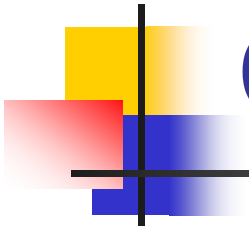
- let x = 3 ;
  y = x*x
  in x*y

Syntax – case-of

```
fib" n = case n of
           0 -> 0;
           1 -> 1;
           m -> fib"(m-1)+
                fib"(m-2)
```

- **case** výraz **of**
  $hodnota_1$ -> $výraz_1$;
  ...
  $hodnota_n$ -> $výraz_n$

```
spoj:: [a] -> [b] -> [(a,b)]
spoj (x:xs) (y:ys)     = (x,y) : spoj xs ys
spoj (x:xs) []         = []
spoj []     zs         = []
```



Currying

prečo nie je zipsovacia fcia definovaná

```
spoy :: ([a],[b]) -> [(a,b)]
```

ale je

```
spoj :: [a] -> [b] -> [(a,b)]
```

v takom prípade musí vyzerať:

```
spoy (x:xs,y:ys) = (x,y) : spoy (xs,ys)
spoy (x:xs,[])   = []
spoy ([],zs)     = []
```

```
Main> spoy ([1,2,3],["a","b","c"])
[(1,"a"),(2,"b"),(3,"c")]
```

f(t1, t2, …, tn)::t

f ::(t1, t2, …, tn) -> t

žijeme vo svete unárnych fcií

f :: t1->t2->…->tn->t

f :: t1->(t2->(…->(tn->t)))

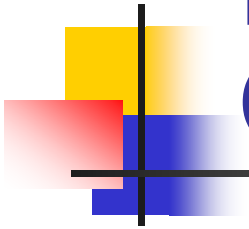
príklad:

```
spoj123 = spoj [1,2,3]
```

```
spoj123::[a] -> [(Int,a)]
```

```
Main> spoj123 [True,False,True]
[(1,True),(2,False),(3,True)]
```

List comprehension
(množinová notácia)



- pri písaní programov používame efektívnu konštrukciu, ktorá pripomína matematický množinový zápis.
- z programátorského hľadiska táto konštrukcia v sebe skrýva cyklus/rekurziu na jednej či viacerých úrovniach.

Príklad:

- zoznam druhých mocnín čísel z intervalu 1..100:

  [n*n | n <- [1..100]] { n*n | n ∈ { 1, …, 100 }}
- zoznam druhých mocnín párnych čísel z intervalu 1..100:

  [n*n | n <- [1..100], even n] { n*n | n ∈ { 1, …, 100} & 2|n}
- zoznam párnych čísel zoznamu:

  selectEven xs = [x | x<-xs , even x] { x | x ∈ xs & even x }

  Main> selectEven [1..10]
  [2,4,6,8,10]

List comprehension
(množinová notácia)

Syntax

[výraz **|** (generátor alebo test)* **]**

<generátor> ::= <pattern> **<-** <výraz typu zoznam (množina)>
<test> ::= <booleovský výraz>

- zoznam vlastných deliteľov čísla

```
factors n       = [ i | i <- [1..n-1], n `mod` i == 0 ]
```

Main> factors 24
[1,2,3,4,6,8,12,24]

- pythagorejské trojuholníky s obvodom <= n

```
pyth n = [ ( a, b, c ) |  a <- [1..n],
                          b <- [1..n],        -- určite aj efektívnejšie ...
                          c <- [1..n],
                          a + b + c <= n,
                          a^2 + b^2 == c^2 ]
```

Main> pyth 25
[(3,4,5),(4,3,5),(6,8,10),(8,6,10)]

Main> :type pyth
pyth :: (Num a, Enum a, Ord a) => a -> [(a,a,a)]

List comprehension (matice)

- malá násobilka:

  nasobilka = [(i, j, i*j) | i <- [1..10], j <- [1..10]]

  [(1,1,1),(1,2,2),(1,3,3), …] :: [(Int,Int,Int)]

  nasobilka' = [[(i,j,i*j) | j <- [1..10]] | i <- [1..10]]

  [[(1,1,1),(1,2,2),(1,3,3),…],
  [(2,1,2),(2,2,4),…….],
  [(3,1,3),…],
  …
  ] :: [[(Int,Int,Int)]]

type Riadok = [Int] – type definuje typové synonymum

type Matica = [Riadok]

- i-ty riadok jednotkovej matice

  riadok i n = [if i==j then 1 else 0 | j <- [1..n]]

  [1,0,0,0]

- jednotková matica

  jednotka n = [riadok i n | i <- [1..n]]

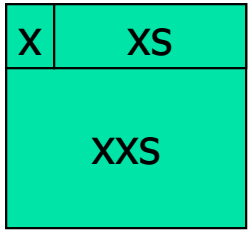
  [[1,0,0,0]
  [0,1,0,0]
  [0,0,1,0]
  [0,0,0,1]]

List comprehension (matice)

- sčítanie dvoch matíc – vivat Pascal ☺

```
scitajMatice        :: Matica -> Matica -> Matica
scitajMatice m n    =
        [ [(m!!i)!!j + (n!!i)!!j | j <- [0..length(m!!0)-1] ]
                                 | i <- [0..length m-1] ]
```

- transponuj maticu pozdĺž hlavnej diagonály




```
transpose                   ::  Matica  ->  Matica
transpose []                =  []
transpose ([] : xss)        =  transpose xss
transpose ((x:xs) : xss)    =  (x : [h | (h:t) <- xss]) :
                                    transpose (xs : [t | (h:t) <- xss])
```

m1 = [[1,2,3],[4,5,6],[7,8,9]]
m2 = [[1,0,0],[0,1,0],[0,0,1]]
m3 = [[1,1,1],[1,1,1],[1,1,1]]
scitajMatice m2 m3 = [[2,1,1],[1,2,1],[1,1,2]]
transpose m1 = [[1,4,7],[2,5,8],[3,6,9]]

List comprehension
(permutácie-kombinácie)

- vytvorte zoznam všetkých 2^n n-prvkových kombinácií {0,1}

```
pre n=2, kombinácie 0 a 1 sú: [[0,0],[1,0],[1,1],[0,1]]
kombinacie 0            = [[]]
kombinacie (n+1)        = [ 0:k | k <- kombinacie n] ++
                          [ 1:k | k <- kombinacie n]
```

- vytvorte permutácie prvkov zoznamu

```
perms []        = [[]]
perms x         = [ a:y | a <- x, y <- perms (diff x [a]) ]
-- rozdieľ zoznamov x y (tie, čo patria do x a nepatria do y)
diff x y        = [ z | z <- x, notElem z y]
```

```
Main> :type perms
perms :: Eq a => [a] -> [[a]]
Main> :type diff
diff :: Eq a => [a] -> [a] -> [a]
```

```
Main> perms [1,2,3]
[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

List comprehension (quicksort)

- quicksort

```
qs              :: [Int] -> [Int]
qs []           = []
qs (a:as)       = qs [x | x <- as, x <= a]
                        ++
                        [a]
                        ++
                  qs [x | x <- as, x > a]
```

Main> qs [4,2,3,4,6,4,5,3,2,1,2,8]
[1,2,2,2,3,3,4,4,4,5,6,8]

Porovnávanie so vzorom
(pattern matching)

V hlavičke klauzule či vo *where*/*let* výraze sa môže vystytnúť vzor typu:

- konštruktorový vzor, n-tica

```
reverse []     = []
reverse (a:x)  = reverse x ++ [a]
```

- n+k - vzor

```
ack 0 n          = n+1
ack (m+1) 0      = ack m 1
ack (m+1) (n+1) = ack m (ack (m+1) n)
```

- wildcards (anonymné premenné)

```
head (x:_)  = x
tail (_:xs) = xs
```

- @-vzor (aliasing)

```
zopakuj_prvy_prvok s@(x:xs) = x:s
```

@-aliasing
(záležitosť efektívnosti)

- definujte test, či zoznam [Int] je usporiadaným zoznamom:

```
-- prvé riešenie (ďalšie alternatívy, viď cvičenie):
usporiadany                         :: [Int] -> Bool
usporiadany []                      = True
usporiadany [_]                     = True
usporiadany (x:y:ys)        | x < y   = usporiadany (y:ys)
                            | otherwise= False
```

- @ alias použijeme vtedy, ak chceme mať prístup (hodnotu v premennej) k celému výrazu (xs), aj k jeho častiam (y:ys), bez toho, aby sme ho najprv deštruovali a následne hneď konštruovali (čo je neefektívne):

```
-- v tomto príklade xs = (y:ys)
usporiadany'''                      :: [Int] -> Bool
usporiadany''' []                   = True
usporiadany''' [_]                  = True
usporiadany''' (x:xs@(y:ys))        = x < y && usporiadany''' xs
```