



# 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



# 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](#)),



# 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](http://comp.lang.functional))



# Funkcia ako argument

---

doteraz sme poznali (??) posielanie funkcie ako argument  
program example;

```
function first(function f(x: real): real): real;  
begin  
    first := f(1.0) + 2.0;  
end;  
  
function second(x: real): real;  
begin  
    second := x/2.0;  
end;  
  
begin  
    writeln(first(second));  
end.
```

To isté v jazyku C:

```
float first(float (*f)(float)) {  
    return (*f)(1.0)+2.0;  
    return f(1.0)+2.0; // alebo  
}  
  
float second(float x) {  
    return (x/2.0);  
}  
  
printf("%f\n",first(&second));
```



# Funkcia ako hodnota

(požičané z goovského cvičenia)

```
type realnaFunckia /*=*/ func(float64) float64

func kompozicia(f, g realnaFunckia) realnaFunckia {
    return (func(x float64) float64 {// kompozicia(f,g) = f.g
        return f(g(x))
    })
}

// iteracia(n,f)=f^n
func iteracia(n int, f realnaFunckia) realnaFunckia {
    if n == 0 {
        return (func(x float64) float64 { return x }) //id
    } else {
        // f . iter(n-1,f)
        return kompozicia(f, iteracia(n-1, f))
    }
}
```



# Closures

(len pre fajňšmeckerov a/alebo pythonistov)

```
def addN(n):                # výsledkom addN je funkcia,  
    return (lambda x:n+x)  # ktorá k argumentu pripočína N
```

```
add5 = addN(5)              # toto je jedna funkcia x 5+x  
add1 = addN(1)              # toto je iná funkcia y 1+y  
                                # ... môžem ich vyrobiť neobmedzene veľa  
print(add5(10))             # 15  
print(add1(10))             # 11
```

```
def iteruj(n,f):             # výsledkom je funkcia  $f^n$   
    if n == 0:  
        return (lambda x:x) # identita  
    else:  
        return(lambda x:f(iteruj(n-1,f)(x))) #  $f(f^{n-1}) = f^n$ 
```

```
add5SevenTimes = iteruj(7,add5) # +5(+5(+5(+5(+5(+5(+5(100)))))))  
print(add5SevenTimes(100))      # 135
```



# 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.



# 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))))
```

pozbiarať *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 striktné typovaný, aj keď typy nevyžaduje (ale ich inferuje)
- je lenivý (v spôsobe výpočtu) . počíta len to čo treba



# Prvá funkcia v Haskell

- $\lambda y \rightarrow y+1$   
 $y.(+ y 1)$   
 $\text{add1 } y \quad = y + 1$

Mená funkcií a premenných  
malým písmenom



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

- $\text{add } x \ y \quad = x + y$        $\lambda x \rightarrow \lambda y \rightarrow (x+y)$  ☺
- $\text{add}' (x,y) \quad = x+y$        $\lambda (x,y) \rightarrow (x+y)$  ☹

$\text{add1} \quad = \text{add } 1$   
 $\text{add1 } y \quad = y + 1$

$\lambda y \rightarrow (1+y)$

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+y`

- funkčný typ:  $t_1 \rightarrow t_2$  (funkcia z typu  $t_1$  do typu  $t_2$ )

asociativita typového operátora **doprava**:

- $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4$  znamená  $t_1 \rightarrow (t_2 \rightarrow (t_3 \rightarrow t_4))$



# Skladanie funkcií

- aplikácia funkcií je **ľavo-asociatívna a nekomutatívna**:  
 $f \ g \ 5 = (f \ g) \ 5 \neq f \ (g \ 5) = (f \ . \ g) \ 5$   
 $\neq g \ f \ 5 \quad \text{!!! zátvorkujte, zátvorkujte, zátvorkujte !!!}$

- operátor `.` znamená skladanie funkcií  
 $(.) :: (u \rightarrow v) \rightarrow (t \rightarrow u) \rightarrow (t \rightarrow v)$ ,  
alebo zjednodušene  
 $(.) :: (t \rightarrow t) \rightarrow (t \rightarrow t) \rightarrow (t \rightarrow t)$

- Príklady:

dvakrát  $f = f \ . \ f$

inak zapísané

dvakrát  $f \ x = f \ (f \ x)$

naDruhu  $x = x * x$

naStvrtu = dvakrát 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  
`nsd 0 y` = y  
`nsd x y` = `nsd y (x `rem` y)`

Klauzule sa aplikujú v  
poradí zhora nadol

--  $x > 0, y > 0$



<http://www.haskell.org/>



WinGHC

[illegible]



# 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 definujeme!

```
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)]

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

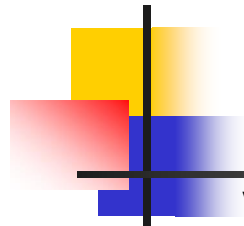
error :: String -> [f]

**where**

delta = b\*b - 4\*a\*c

radix = sqrt delta

pod-výrazy sa počítajú zhora nadol



# Logaritmicky $x^n$

``mod`` = ``rem`` - zvyšok po delení  
``div`` = ``quot`` - celočísl.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

$(t_1, t_2, \dots, t_n)$

$n \geq 2$

Konštanty:

$(5, \text{False}) :: (\text{Int}, \text{Bool})$

$(5, (\text{False}, 3.14)) :: (\text{Int}, (\text{Bool}, \text{Double})) \neq (\text{Int}, \text{Bool}, \text{Double})$

- zoznamy

$[t]$

napr.  $[\text{Int}]$ ,  $[\text{Char}]$

konštruktory:  $h:t, []$

konštanta:  $[1,2,3]$

vieme zapísať konštanty *typu* zoznam a poznáme konvencie

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

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

# Zoznamy sú homogénne

sú vždy homogénne (na rozdiel 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)

- zretáženie 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]



# 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] -- zret'azenie 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, > rev [1..4]  
[4,3,2,1]  
append [x] na koniec

-- iteratívny reverse -- otočenie ale iteratívne

reverse xs = rev' xs []

rev' [] ys = ys > reverse [1..4]

rev' (x:xs) ys = rev' xs (x:ys) [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.d'.

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

`spoj (x:xs) (y:ys) = (x,y) : spoj xs ys`

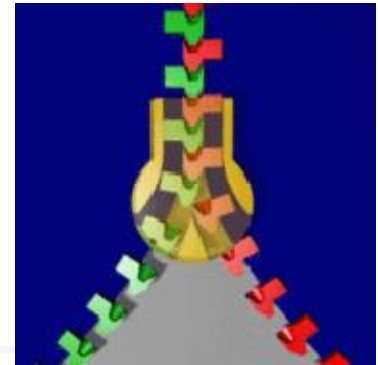
`spoj (x:xs) [] = []`

`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.

■  $\text{rozpoj} :: [(a,b)] \rightarrow ([a],[b])$

$\text{rozpoj []} = ([], [])$

$\text{rozpoj } ((x,y):ps) = (x:xs, y:ys)$

where

$(xs,ys) = \text{rozpoj } ps$

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

Táto funkcia sa štandardne volá unzip

dvojica ako pattern

$\text{rozpoj } ((x,y):ps) = \text{let } (xs,ys) = \text{rozpoj } ps \text{ in } (x:xs, y:ys)$



# Syntax – let-in

- rozpoj  $:: [(a,b)] \rightarrow ([a],[b])$

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

- let** pattern<sub>1</sub> = výraz<sub>1</sub> ;  
...  
pattern<sub>n</sub> = výraz<sub>n</sub> ;  
**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<sub>1</sub> -> výraz<sub>1</sub> ;  
...  
hodnota<sub>n</sub> -> výraz<sub>n</sub>



# Currying

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

prečo nie je zipovacia fcia definovaná

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

ale je

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

v takom prípade musí vyzerat':

`spoy (x:xs,y:ys) = (x,y) : spoy (xs,ys)`

`spoy (x:xs,[]) = []`

`spoy ([],zs) = []`

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

```
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:  
 $[n*n \mid n \leftarrow [1..100]] \quad \{n*n \mid n \in \{1, \dots, 100\}\}$
- zoznam druhých mocnín párnych čísel z intervalu 1..100:  
 $[n*n \mid n \leftarrow [1..100], \text{even } n] \quad \{n*n \mid n \in \{1, \dots, 100\} \ \& \ 2|n\}$
- zoznam párnych čísel zoznamu:  
 $\text{selectEven } xs = [x \mid x \leftarrow xs, \text{even } x] \quad \{x \mid x \in xs \ \& \ \text{even } x\}$   
**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 Matica = [Riadok]
```

– type definuje typové synonymum

- i-ty riadok jednotkovej matice

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

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

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

- jednotková matica

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

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

```
[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]]**

x	xs
xss	





# 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]) ]`  
-- rozdiel' zoznamov x y (tie, čo patria do x a nepatria do y)  
`diff x y` =  `[ z | z <- x, notElem z y]`

**Main> :type perms**

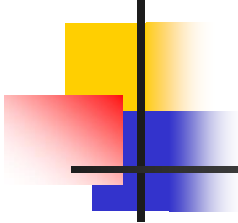
**perms :: Eq a => [a] -> [[a]]**

**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 vyskytnúť vzor typu:

- konštruktorový vzor, n-tica

<code>reverse []</code>	<code>= []</code>
<code>reverse (a:x)</code>	<code>= reverse x ++ [a]</code>

- **n+k - vzor**

<code>ack 0 n</code>	<code>= n+1</code>
<code>ack (m+1) 0</code>	<code>= ack m 1</code>
<code>ack (m+1) (n+1)</code>	<code>= ack m (ack (m+1) n)</code>

- wildcards (anonymné premenné)

<code>head (x:_)</code>	<code>= x</code>
<code>tail (_:xs)</code>	<code>= xs</code>

- @-vzor (aliasing)

<code>zopakuj_prvy_prvok s@(x:xs)</code>	<code>= x:s</code>
--	--------------------



# @-aliasing

## (záležitosť efektívnosti)

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

-- prvé riešenie (ďalšie alternatívy, vid' 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
```