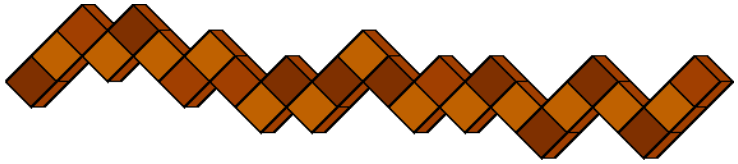
Funkcionálne programovanie 3

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

- parametrický polymorfizmus na príkladoch funkcionálov (map, filter, foldl, foldr)
- backtracking (ako príklad na list-comprehension)

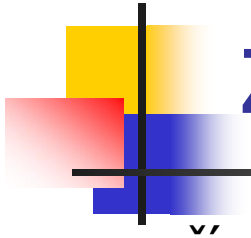


Cvičenie:

- funkcionálny štýl (map, filter, foldr, …)
- backtracking v Haskelli,

Phil Wadler, λ-man

Funkcia je hodnotou - zatiaľ len argumentom



učíme sa z Prelude.hs:
štandardná knižnica haskellu obsahuje množstvo:
- užitočných funkcií,
- vzorových funkcií.

- zober zo zoznamu tie prvky, ktoré spĺňajú bool-podmienku (test)

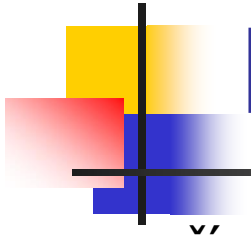
booleovská podmienka príde ako argument funkcie a má typ (a -> Bool):

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

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

- rozdeľ zoznam na zoznam menších, rovných a väčších prvkov ako prvý: riešenie menej efektívne ale v istom zmysle elegantnejšie

```
tripivot (x:xs)  = ( filter (<x) xs, filter (==x) xs, filter (>x) xs )
```



Funkcia (predikát) argumentom

učíme sa z Prelude.hs:

- ber zo zoznamu prvky, kým platí logická podmienka (test):

```
takeWhile                       :: (a -> Bool) -> [a] -> [a]
takeWhile p []                  = []
takeWhile p (x:xs) | p x        = x : takeWhile p xs
                   | otherwise  = []
```

> takeWhile (>0) [1,2,-1,3,4]
[1,2]

- vyhoď tie počiatočné prvky zoznamu, pre ktoré platí podmienka:

```
dropWhile                       :: (a -> Bool) -> [a] -> [a]
dropWhile p []                  = []
dropWhile p xs@(x:xs')  | p x   = dropWhile p xs'
                        | otherwise = xs
```

> dropWhile (>0) [1,2,-1,3,4]
[-1,3,4]

Príklad (porozdeľuj)

Definujte porozdeluj :: (a -> Bool) -> [a] -> [[a]] , ktorá rozdelí zoznam na podzoznamy, v ktorých súvisle platí podmienka daná 1. argumentom

porozdeluj (>0) [1,2,0,3,4,5,-1,6,7] = [[1,2],[3,4,5],[6,7]]
porozdeluj (\x -> x `mod` 3 > 0) [1..10] = [[1,2],[4,5],[7,8],[10]].

```
porozdeluj p [] = []
porozdeluj p xs =
        (takeWhile p xs) :          -- prefix, kým platí p je prvým prvkom
            porozdeluj p            -- rekurzívne volanie na ďalšie prvky
                (dropWhile (\x -> (not (p x))) -- odstráň, kým neplatí p
                    (dropWhile p xs))          -- odstráň, kým platí p
```

```
Main> porozdeluj (>0) [1,2,0,0,3,4,-1,5]
[[1,2],[3,4],[5]]
```

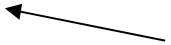
Funktor map

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

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

- Príklad použitia:

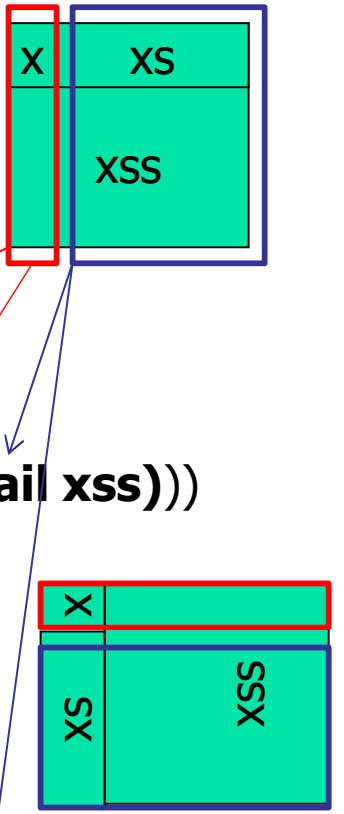
```
map (+1) [1,2,3,4,5]                    = [2,3,4,5,6]
map odd  [1,2,3,4,5]                    = [True,False,True,False,True]
and (map odd  [1,2,3,4,5])              = False
all p xs = and (map p xs)     -- all p xs = p platí pre všetky prvky zoznamu xs

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]]
map (++[0]) [[1],[2],[3]]               = [[1,0],[2,0],[3,0]]
```



\x->x++[0]

Transponuj maticu

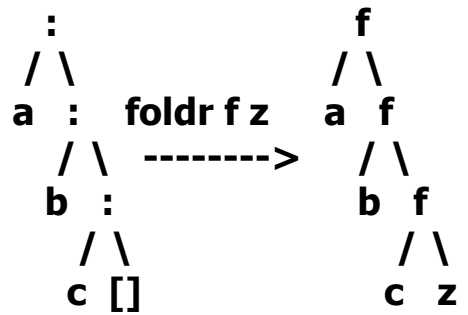
```
-- transponuj pomocou map, nie list-comprehension

transponuj                  :: Matica -> Matica

transponuj []               = []
transponuj  ([]:xss)        = transponuj xss
transponuj ((x:xs):xss)     = (x:(map head xss)):
                                    (transponuj (xs:(map tail xss)))

-- riešenie z minulej prednášky
transpose []                = []
transpose ([] : xss)        = transpose xss
transpose ((x:xs) : xss)    = (x : [h | (h:t) <- xss]) :
                                    transpose (xs : [t | (h:t) <- xss])
```



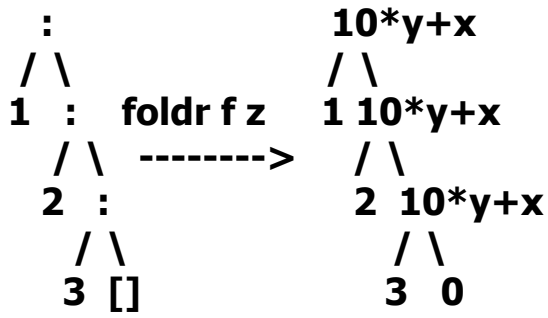
Ďalšie (známe) funkcionály –

(foldr – schéma rekurzie na zoznamoch)

```
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))
```




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



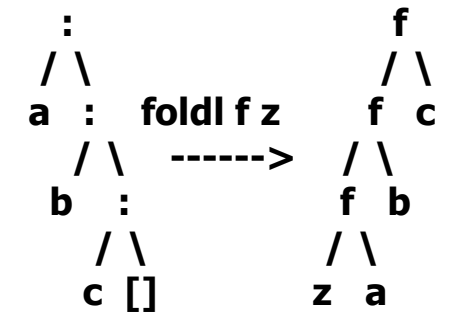

```
Main> foldr (\x y->10*y+x) 0 [1,2,3]
321
```

nepomenovaná analógia fcie: pom x y = 10*y+x

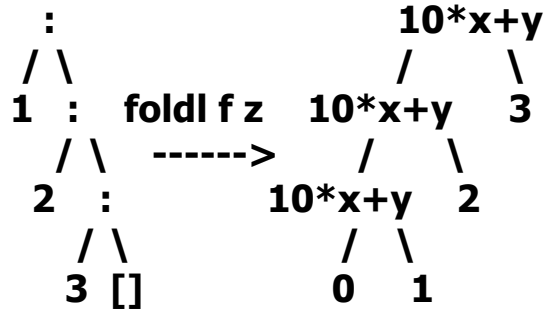
```
-- g je vnorená lokálna funkcia

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z = g
  where    g []     = z
           g (x:xs) = f x (g xs)
```

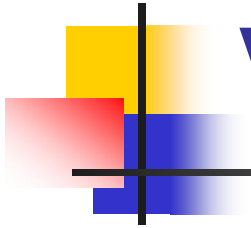
Ďalšie (známe) funkcionály -
(foldl – schéma iterácie na zoznamoch)

```
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
```




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

Main> foldl (\x y->10*x+y) 0 [1,2,3]
123
```





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]

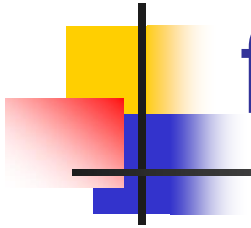
- foldr (-) 0 [1..100] =

(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

foldr a foldl ešte raz



- foldl (+) 0 [1,2,3,4]
  foldr (+) 0 [1,2,3,4]

  10

- foldr max (-999) [1,2,3,4]
  foldl max (-999) [1,2,3,4]

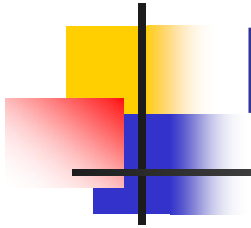
  4

- foldr (_ -> \y ->(y+1)) 0 [3,2,1,2,4]
  foldl (\x -> _ ->(x+1)) 0 [3,2,1,2,4]

  5

- rozpoj :: [(a,b)] -> ([a],[b])
  rozpoj = foldr (\(x,y) -> \(xs,ys) -> (x:xs, y:ys)) ([], [])

  rozpoj [(1,11),(2,22),(3,33)]
  ([1,2,3],[11,22,33])



Funkcia je hodnotou

- [a->a] je zoznam funkcií typu a->a
  napríklad: [(+1),(+2),(*3)] je [\x->x+1,\x->x+2,\x->x*3]

- čo je foldr (.) id [(+1),(+2),(*3)] ??
  akého je typu [a->a]
  foldr (.) id [(+1),(+2),(*3)] 100 303
  foldl (.) id [(+1),(+2),(*3)] 100 ???

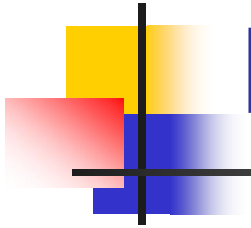
lebo skladanie fcií je asociatívne:

- ((f . g) . h) x = (f . g) (h x) = f (g (h x)) = f ((g . h) x) = (f . (g . h)) x

- mapf :: [a->b] -> [a] -> [b]
  mapf [] _ = []
  mapf _ [] = []
  mapf (f:fs) (x:xs) = (f x):mapf fs xs

-- rovnako dlhé zoznamy

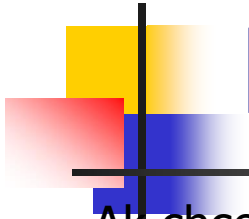
mapf [(+1),(+2),(+3)] [10,20,30] [11,22,33]



Kvíz

```
foldr (:) [] xs =     xs

foldr (:) ys xs =     xs++ys
```

```
foldr ? ? xs = reverse xs
```



Priemerný prvok

Ak chceme vypočítať aritmetický priemer (a-priemer) prvkov zoznamu, matice, ... potrebujeme poznať ich súčet a počet. Ako to urobíme na jeden prechod štruktúrou pomocou foldr/foldl ? ...počítame dvojicu hodnôt, súčet a počet:

- priemerný prvok zoznamu

```
priemer  xs = sum/count where (sum, count) = sumCount xs
sumCount xs = foldr (\x -> \(sum, count) -> (sum+x, count+1)) (0, 0) xs
```

- priemerný prvok matice

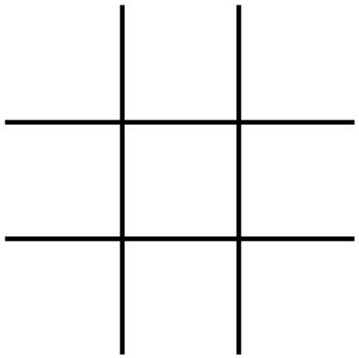
  je a-priemer a-priemerov riadkov matice a-priemerom hodnôt matice ?

```
sumCount'          :: [[Float]] -> (Float,Float)
sumCount' xs =
    foldr (\x -> \(sum, count)->scitaj (sumCount x) (sum, count)) (0, 0) xs
    where scitaj (a,b) (c,d) = (a+c, b+d)

priemer' :: [[Float]] -> Float
priemer' = uncurry (/) . sumCount'
```

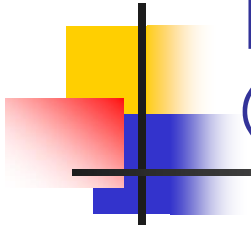
```
uncurry :: (a->b->c) -> (a,b) -> c
uncurry f (a,b) = f a b
curry    :: ((a,b) -> c) -> (a->b->c)
curry g a b    = g (a,b)
```

Bactracking
(ľahký úvod)

- vložte 6 kameňov do mriežky 3x3, tak aby v žiadnom smere (riadok, stĺpec, uhlopriečka) neboli tri.



- pri najivnom prehľadávaní - všetkých možností je 2^9 = 512
- ak poznáme kombinácie bez opakovania - možností je už len 9 nad 6, teda 9 nad 3, čo je 84



Riešenie „brute force“

(s ním prežijete len pri smiešne jednoduchých úlohách)

```
subset' ys xs = and (map (\x -> elem x xs) ys)
subset'' xs ys = all (`elem` ys) xs

isOk       :: [Int] -> Bool
isOk xs =
   not (subset' [0,1,2] xs) && not (subset' [3,4,5] xs) && not (subset' [6,7,8] xs) &&
   not (subset' [0,3,6] xs) && not (subset' [1,4,7] xs) && not (subset' [2,5,8] xs) &&
   not (subset' [0,4,8] xs) && not (subset' [2,4,6] xs)
```

vygenerujeme všetky podmnožiny [0..8] a vyberieme správne dĺžky 6

```
powerSet            :: [Int] -> [[Int]]
powerSet   []       = [[]]
powerSet  (x:xs)  = map (x:) ps ++ ps  where ps = powerSet  xs

solve3x3 ' = filter (\x -> 6 == length x) (filter isOk (powerSet [0..8]))
```

Riešenie „kombinatorické“
(s ním prežijete len pri smiešne jednoduchých úlohách)

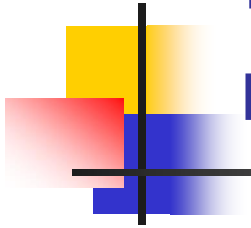
generujeme len 6 prvkové kombinácie [0..8] a z nich vyberieme správne

```
kbo                 :: [Int] -> Int -> [[Int]]
kbo _ 0             = [[]]
kbo [] _            = []
kbo (x:xs) k        = [x:ys | ys <- kbo xs (k-1)] ++ kbo xs k

solve3x3''          = filter isOk (kbo [0..8] 6)
```

generujeme len správne 6 prvkové kombinácie [0..8]

Backtracking = testujeme správnosť riešenia už počas jeho tvorby

```
kbo'                :: [Int] -> Int -> [[Int]]
kbo' _ 0            = [[]]
kbo' [] _           = []
kbo' (x:xs) k       = [x:ys | ys <- kbo' xs (k-1), isOk (x:ys)] ++ kbo' xs k

solve3x3            = kbo' [0..8] 6
```



Backtracking
magické číslo

```
? 381654729
2| 38
3| 381
4| 3816
5| 38165
6| 381654
7| 3816547
8| 38165472
9| 381654729
```

číslo s neopakujúcimi sa ciframi 1..9, ktorého prvých i-cifier je delitelných i

```
-- (čiastočným) riešením je zoznam cifier od konca, t.j. [3,2,1] je číslo 123
type Riesenie = [Int]                    -- typ riešenia

-- nájdi všetky magické riešenia, ktoré obsahujú predpísané cifry
magicBacktrack              :: [Int] -> [Riesenie]      -- hľadáme zoznam
magicBacktrack []= [[]]                                 -- riešení
magicBacktrack cifry        = [ c:ciastocneRiesenie     |
                              c<-cifry,
                              ciastocneRiesenie<-magicBacktrack (diff cifry [c]),
                              jeMagicke (c:ciastocneRiesenie) ]

-- diff (rozdiel zoznamov) poznáme z minulej prednášky, alebo cifry \\ [c]

-- jeMagicke je test, ktorý overí konzistentnosť nového čiastočného riešenia
```

Backtracking
jeMagické

```
jeMagicke                   :: Riesenie -> Bool
jeMagicke cs                = jeMagicke' 1 0 (reverse cs)

jeMagicke'                  :: Int->Int->Riesenie -> Bool
jeMagicke' _ _ []           = True
jeMagicke' i stareCislo (cifra:cs)
                            = noveCislo `mod` i == 0
                                    &&
                              jeMagicke' (i+1) noveCislo cs
                            where
                             noveCislo = 10*stareCislo+cifra
```

```
Main> magicBacktrack [1,2,3,4,5,6,7,8,9]
[[9,2,7,4,5,6,1,8,3]]
```

t.j. jediné riešenie je
381654729

opäť jeMagicke

Definujme pomocou schém rekurzie foldr a iterácie foldl

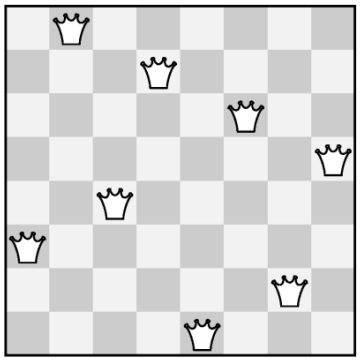
foldr:

```
jeMagicke''          :: Riesenie -> Bool
jeMagicke'' xs       = res
            where (_,_,res) =
            foldr (\cifra -> \(count, cislo, result) ->
                              (count+1, 10*cislo+cifra,
                              result  && ( (10*cislo+cifra) `mod` (count+1) == 0)))
                  (0, 0, True)
                  xs
```

foldl: na cvičení...

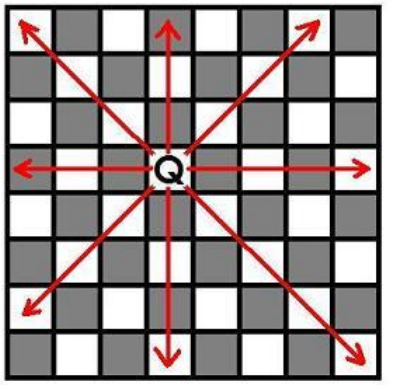
```
Main> jeMagicke'' [9,2,7,4,5,6,1,8,3]
True
```

Backtracking
8 dám na šachovnici



```
-- definujeme typ popisujúci riešenie problému
type RiesenieDam = [Int]          -- pozície dám v stĺpcoch 1,2,3, ..., n
                                  -- príklad riešenia [3,1,6,2,5,7,4,0]
                                  -- hľadáme permutáciu [0..N-1], [0..7]
-- definujeme rekurzívnu funkciu, ktorá vráti zoznam všetkých riešení
damyBacktrack                :: Int -> [RiesenieDam] -- všetky riešenia,
                                -- argument Int určuje, ako ďaleko v "riešení" sme

damyBacktrack 0              = [[]]   -- jedno triviálne riešenie pre 0x0
damyBacktrack (n+1)          = [      -- začiatok list-comprehension
    dama:ciastocneRiesenie   |        -- nové riešenie daj do zoznamu riešení

          ciastocneRiesenie <- damyBacktrack n, -- rekurzívne volanie nájde
                                                -- čiastočné riešenie
          dama <- [0..7],                       -- rozšírime čiastočné riešenie
                                                -- o ďalšiu dámu na nové r.
          damyOk n dama ciastocneRiesenie]      -- otestuj, či je to ešte riešenie
```

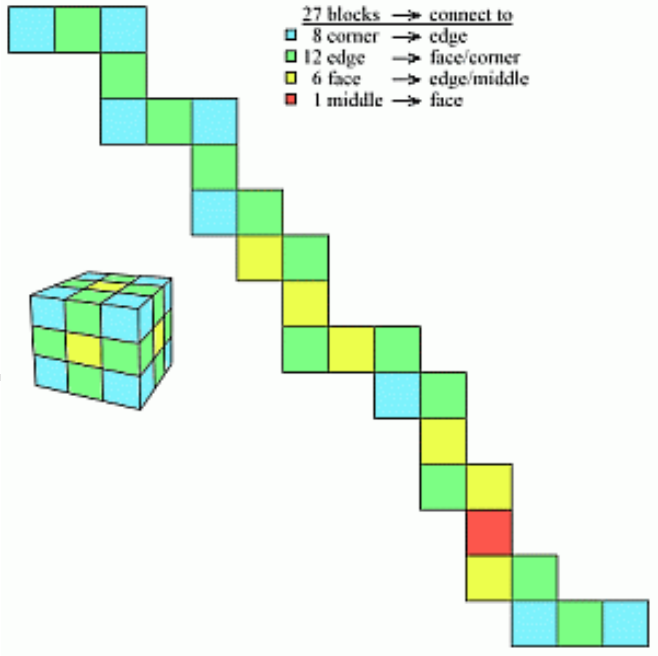
Backtracking
(ťažké: orezávanie stromu možností)




```
damyOk                  :: Int -> Int -> RiesenieDam -> Bool

damyOk n ndama ciastocneRiesenie          = and [    -- dámy sú ok, ak žiadna
    not (damaVOhrozeni ndama stlpec ciastocneRiesenie) |   -- z nich nie je
    stlpec <- [0..(n-1)] ]                  -- v ohrození, pre všetky už položené dámy


damaVOhrozeni           :: Int-> Int-> RiesenieDam->Bool
                                                    -- dáma je v ohrození
damaVOhrozeni ndama stlpec ciastocneRiesenie  =     -- od dámy v stĺpec
    (ndama==ciastocneRiesenie!!stlpec) ||           -- ak sú v rovnakom riadku
    (abs(ndama - ciastocneRiesenie!!stlpec)==stlpec+1)  -- alebo diagonále
```

```
Main> damy
[[3,1,6,2,5,7,4,0],[4,1,3,6,2,7,5,0],[2,4,1,7,5,3,6,0],[2,5,3,1,7,4,6,0]
```

Drevený had



- ako na to ?
- popíšeme hada dĺžkami jednotlivých „rebier"

```
type Had  = [Int]
had1      :: Had
had1      = [3,3,3,3,2,2,2,3,3,2,2,3,2,3,2,2,3]
```

- počet článkov = počet prvkov v zozname = `length(mysnake)-1 = 16`
- každý kocka je započítaný 2x, preto

  `(sum mysnake) - ((length mysnake)-1) == 27`
- každý kocka má 4 možné polohy, takže naivne 4^{16} = 4.294.967.296 možností
- ak si kocku predstavíme v 3-rozmernom priestore, každé črebro má smer niektorého z vektorov [±1,0,0], [0,±1,0], [0,0,±1]
- v kocke sa had môže=musí zohnúť o 90º, ako vyzerá smer ďalšieho rebra ?
- kolmé vektory k vektoru [±1,0,0] sú [0,±1,0] [0,0,±1,]

  k vektoru [0,±1,0] sú [±1,0,0] [0,0,±1,]

  k vektoru [0,0,±1] sú [±1,0,0] [0,±1,0]

Let's code

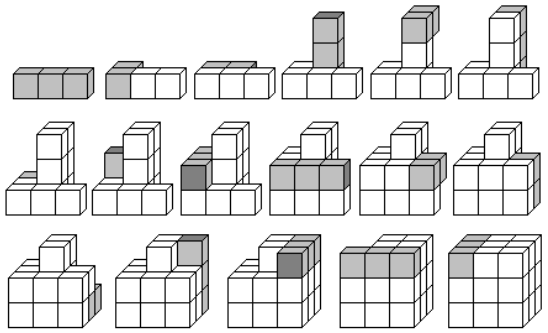
- kolmé vektory k vektoru

```
type SVektor = (Int,Int,Int)      -- dx, dy, dz
kolme          :: SVektor -> [SVektor]
kolme (_,0,0) = [(0,1,0),(0,-1,0),(0,0,1),(0,0,-1)]
kolme (0,_,0) = [(1,0,0),(-1,0,0),(0,0,1),(0,0,-1)]
kolme (0,0,_) = [(0,1,0),(0,-1,0),(1,0,0),(-1,0,0)]
```

- test, ľ i kociľ ka je v kocke 3x3x3 (ľahko parametrizovateľné pre 4x4x4):

```
type Pozicia          = (Int,Int,Int)      -- x, y, z
vKocke3               ::  Pozicia -> Bool
vKocke3 (x,y,z)       = x `elem` [1..3] &&
                        y `elem` [1..3] &&
                        z `elem` [1..3]
```

Let's code



- Ňalšia kocka v danom smere má súradnice

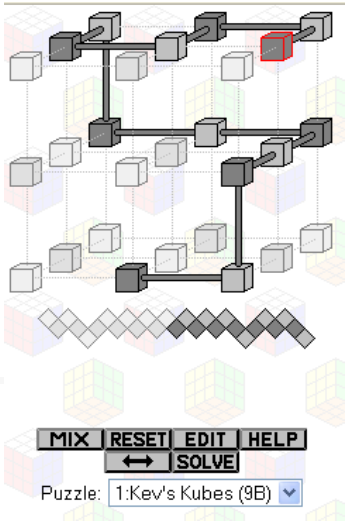
```
nPoz          :: Pozicia -> SVektor -> Pozicia
nPoz (x,y,z) (dx,dy,dz) = (x+dx,y+dy,z+dz)
```

- kocky celého rebra danej dĺžky (2, 3, event. 4)

```
type Rebro              = [Pozicia]
rebro                   :: Pozicia -> SVektor -> Int -> Rebro
rebro start smer 2      = [nPoz start smer]
rebro start smer 3      = [nPoz (nPoz start smer) smer,
                           nPoz start smer]
-- pre 4x4x4 ešte pridáme:
rebro start smer 4      = [nPoz(nPoz(nPoz start smer) smer) smer,
                           nPoz (nPoz start smer) smer,
                           nPoz start smer]
```

Zlož hada

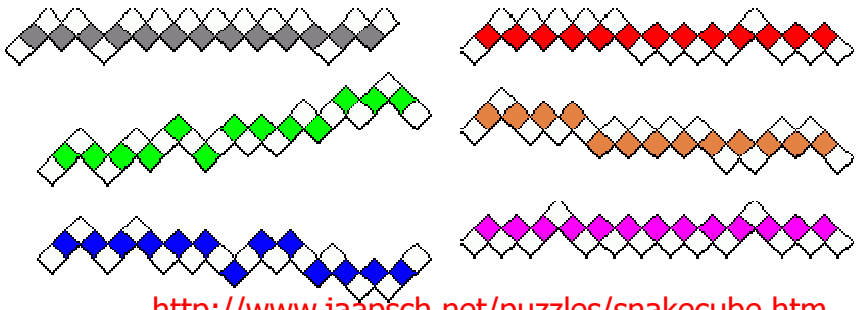
http://www.jaapsch.net/puzzles/javascript/snakecubej.htm




```
zloz [[(1,1,1)]] (0,0,1) had1
```

```
zloz :: Solution -> SVektor -> Had -> [Solution]
zloz ciastocne smer []  = [ciastocne]
zloz ciastocne smer (len:rebra) =
  [ riesenie |
      kolmySmer <- kolmo smer,
      koniecHada <- [head (head ciastocne)],
      noveRebro <- [rebro koniecHada kolmySmer len],
      all vKocke3 noveRebro,
      all (`notElem` concat ciastocne) noveRebro,
      riesenie <-zloz (noveRebro:ciastocne) kolmySmer rebra
  ]
```

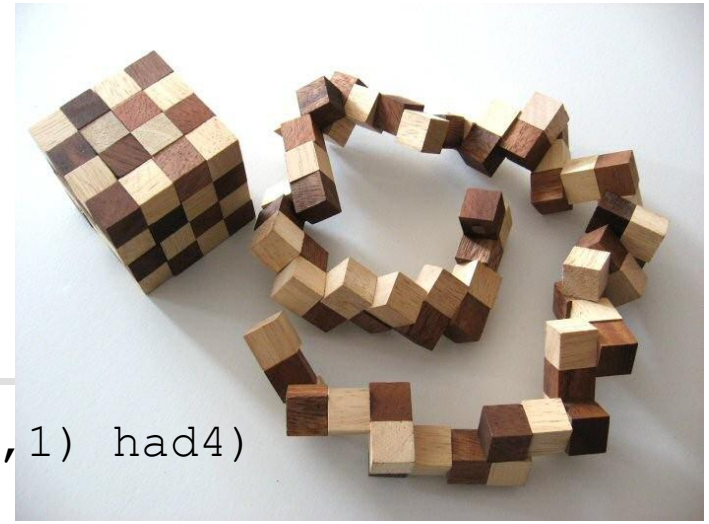
Výsledky 3x3x3



http://www.jaapsch.net/puzzles/snakecube.htm

```
Main> head(zloz3 [[(1,1,1)]] (0,0,1) had1)
[[(3,3,3),(2,3,3)],[(1,3,3)],[(1,2,3)],[(2,2,3),(2,2,2)],[(2,2,1)
   ],[(2,1,1),(2,1,2)],[(2,1,3)],[(1,1,3)],[(1,1,2),(1,2,2)],[(1,3
   ,2),(2,3,2)],[(3,3,2)],[(3,2,2)],[(3,2,3)],[(3,1,3),(3,1,2)],[(
   3,1,1),(3,2,1)],[(3,3,1),(2,3,1)],[(1,3,1),(1,2,1)],[(1,1,1)]]
=== 4x4x4
*Main> :load "c:\\had.hs"
[1 of 1] Compiling Main                ( C:\had.hs, interpreted )
Ok, modules loaded: Main.
*Main> head (zloz4 [[(2,1,1)]] (0,0,1) had4)
[[(1,4,2)],[(1,4,1),(1,3,1),(1,2,1)],[(1,1,1)],[(1,1,2)],[(2,1,2)],[(2,
,2,2)],[(1,3,2)],[(2,3,2)],[(2,3,3)],[(2,2,3)],[(1,2,3),(1,3,3)],[(1,4,
,4),(1,3,4),(1,2,4)],[(1,1,4)],[(1,1,3)],[(2,1,3)],[(2,1,4),(2,2,4),(2,
,4,4),(3,4,4)],[(4,4,4)],[(4,3,4)],[(3,3,4)],[(3,3,3)],[(4,3,3)],[(4,4,
)],[(2,4,3),(2,4,2)],[(2,4,1)],[(3,4,1)],[(3,4,2)],[(4,4,2)],[(4,4,1)],
,[(4,3,2)],[(3,3,2)],[(3,2,2),(3,2,3)],[(3,2,4)],[(3,1,4),(3,1,3)],[(3,
,1,2),(4,1,3)],[(4,1,4)],[(4,2,4),(4,2,3),(4,2,2)],[(4,2,1)],[(4,1,1)],
(3,2,1)],[(3,3,1)],[(2,3,1),(2,2,1)],[(2,1,1)]]
```

Výsledky 4x4x4



```
*Main> head (zloz4 [[(2,1,1)]] (0,0,1) had4)
```

*Main> head (zloz4 [[(2,1,1)]] (0,0,1) had4)

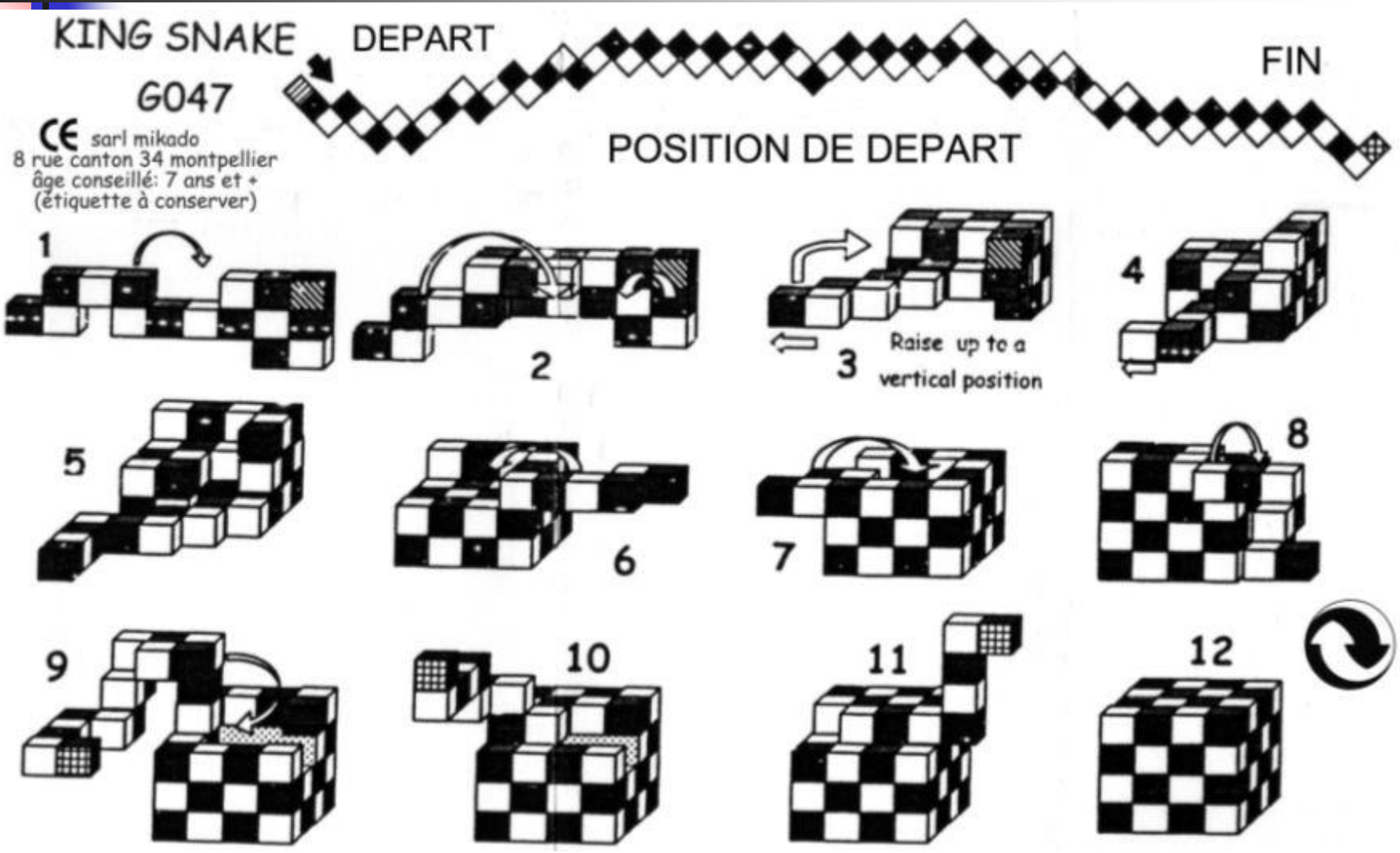
```
[[(1,4,2)],[(1,4,1),(1,3,1),(1,2,1)],[(1,1,1)],[(1,1,2)],[(2,1,2)],[(2,2,2)],[(1
,2,2)],[(1,3,2)],[(2,3,2)],[(2,3,3)],[(2,2,3)],[(1,2,3),(1,3,3)],[(1,4,3)],[(1,4
,4),(1,3,4),(1,2,4)],[(1,1,4)],[(1,1,3)],[(2,1,3)],[(2,1,4),(2,2,4),(2,3,4)],[(2
,4,4),(3,4,4)],[(4,4,4)],[(4,3,4)],[(3,3,4)],[(3,3,3)],[(4,3,3)],[(4,4,3),(3,4,3
)],[(2,4,3),(2,4,2)],[(2,4,1)],[(3,4,1)],[(3,4,2)],[(4,4,2)],[(4,4,1)],[(4,3,1)]
,[(4,3,2)],[(3,3,2)],[(3,2,2),(3,2,3)],[(3,2,4)],[(3,1,4),(3,1,3)],[(3,1,2)],[(4
,1,2),(4,1,3)],[(4,1,4)],[(4,2,4),(4,2,3),(4,2,2)],[(4,2,1)],[(4,1,1)],[(3,1,1),
(3,2,1)],[(3,3,1)],[(2,3,1),(2,2,1)],[(2,1,1)]]
```

```
*Main> head (zloz4 [[(1,1,1)]] (0,0,1) had4)
```

```
[[(4,4,4)],[(4,3,4),(3,3,4),(2,3,4)],[(1,3,4)],[(1,4,4)],[(1,4,3)],[(1,3,3)],[(2
,3,3)],[(2,4,3)],[(2,4,4)],[(3,4,4)],[(3,4,3)],[(4,4,3),(4,3,3)],[(4,2,3)],[(4,2
,4),(3,2,4),(2,2,4)],[(1,2,4)],[(1,2,3)],[(1,1,3)],[(1,1,4),(2,1,4),(3,1,4)],[(4
,1,4),(4,1,3)],[(4,1,2)],[(3,1,2)],[(3,1,3)],[(2,1,3)],[(2,2,3)],[(2,2,2),(2,3,2
)],[(2,4,2),(3,4,2)],[(4,4,2)],[(4,4,1)],[(4,3,1)],[(4,3,2)],[(4,2,2)],[(4,2,1)]
,[(4,1,1)],[(3,1,1)],[(3,2,1),(3,2,2)],[(3,2,3)],[(3,3,3),(3,3,2)],[(3,3,1)],[(3
,4,1),(2,4,1)],[(1,4,1)],[(1,4,2),(1,3,2),(1,2,2)],[(1,1,2)],[(2,1,2)],[(2,1,1),
(2,2,1)],[(2,3,1)],[(1,3,1),(1,2,1)],[(1,1,1)]]
```

http://collection.cassetete.free.fr/1_bois/cube_elastique3x3/cube_elastique_3x3.htm

http://www.jbd-jouetsenbois.com/king-snake-xml-380_385-1238.html



Prémia Japončíci na plti

(http://www.justonlinegames.com/games/river-iq-game.html)