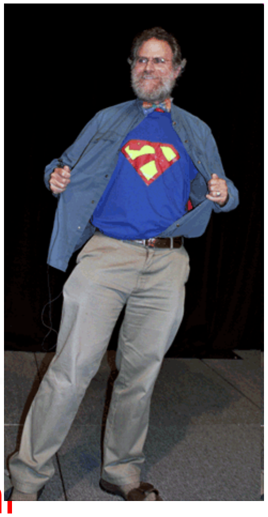
Monády – úvod



Phil Wadler: http://homepages.inf.ed.ac.uk/wadler/topics/monads.html

- Monads for Functional Programming In *Advanced Functional Programming* , Springer Verlag, LNCS 925, 1995,

- Noel Winstanley: What the hell are Monads?, 1999

http://web.cecs.pdx.edu/~antoy/Courses/TPFLP/lectures/MONADS/Noel/research/monads.html

- Jeff Newbern's: All About Monads
  https://www.cs.rit.edu/~swm/cs561/All_About_Monads.pdf

- Dan Bensen: A (hopefully) painless introduction to monads,
  http://www.prairienet.org/~dsb/monads.htm

Monady sú použiteľný nástroj pre programátora poskytujúci:

- modularitu – skladať zložitejšie výpočty z jednoduchších (no side-effects),
- flexibilitu – výsledný kód je ľahšie adaptovateľný na zmeny,
- izoluje side-effect operácie (napr. IO) od čisto funkcionálneho zvyšku.

Maybe monad

Maybe je podobné Exception (Nothing~~Raise String, Just a ~~Return a)

```
data Maybe a = Nothing | Just a

instance Monad Maybe where
    return v        = Just v          -- vráť hodnotu
    fail            = Nothing         -- vráť neúspech

    Nothing  >>= f = Nothing          -- ak už nastal neúspech, trvá do konca
    (Just x) >>= f  = f x             -- ak je zatiaľ úspech, závisí to na výpočte f
```

```
return v              = Just v
fail                  = Nothing
Nothing  >>= f = Nothing
(Just x)  >>= f  = f x
```

Príklady na Maybe Monad

- konštaty: Just 1, Nothing :: Maybe Int
- základné operácie:

```
(Just 1) >>= (\x -> Nothing)                                = Nothing
(Just 1) >>= (\x -> Just (1+x))                             = Just 2
Nothing >>= (\x -> Just (1+x))                              = Nothing

do { x<-(Just 1); Nothing } :: Maybe t                      = Nothing
do { x<-(Just 1); return (1+x) } :: Maybe Int               = Just 2
do { x<-(return 1); return (1+x) }  :: (Monad m, Num b) => m b
do { x<-(return 1); return (1+x) }                          = 2
do { x<-(return 1); return (1+x) } :: Maybe Int             = Just 2
do { x<-(return 1); return (1+x) } :: [Int]                 = [2]
do { x<-Nothing; return (1+x) } :: Maybe t                  = Nothing
```

Maybe sequence

```
sequence        :: Monad m => [m a] -> m [a]
sequence []     = return []
sequence (c:cs) = do { x <- c; xs <- sequence cs; return (x:xs) }
```

```
sequence [Just "a", Just "b", Just "d"]             = Just ["a","b","d"]
sequence [Just "a", Just "b", Nothing, Just "d"]    = Nothing
```

Maybe MonadPlus

```
data Maybe a = Nothing | Just a
```

class Monad m => MonadPlus m where – podtrieda, resp. podinterface

```
    mzero     :: m a                          -- ø
    mplus     :: m a -> m a -> m a            -- disjunkcia

instance MonadPlus Maybe where
    mzero                    = Nothing        -- fail...
    Just x `mplus` y         = Just x         -- or
    Nothing `mplus` y        = y

Just "a" `mplus` Just "b"                       = Just "a"
Just "a" `mplus` Nothing                        = Just "a"
Nothing `mplus` Just "b"                        = Just "b"
```

Zákony monád a monády Plus

- vlastnosti return a >>=:

```
return x >>= f                = f x                    -- return ako identita zľava
p >>= return                  = p                      -- retrun ako identita sprava
p >>= (\x -> (f x >>= g))= (p >>= (\x -> f x)) >>= g  -- "asociativita"
```

- vlastnosti zero a `plus`:

```
zero `plus` p                 = p                      -- zero ako identita zľava
p `plus` zero                 = p                      -- zero ako identita sprava
p `plus` (q `plus` r)         = (p `plus` q) `plus` r     -- asociativita
```

- vlastnosti zero `plus` a >>= :

```
zero >>= f                    = zero                   -- zero ako identita zľava
p >>= (\x->zero)              = zero                   -- zero ako identita sprava
(p `plus` q) >>= f            = (p >>= f) `plus` (q >>= f)     -- distribut.
```

Sheep family

```
adam   = Sheep "Adam"    Nothing Nothing
eve    = Sheep "Eve"     Nothing Nothing
uranus = Sheep "Uranus"  Nothing Nothing
gaea   = Sheep "Gaea"    Nothing Nothing
kronos = Sheep "Kronos"  (Just gaea) (Just uranus)
holly  = Sheep "Holly"   (Just eve) (Just adam)
roger  = Sheep "Roger"   (Just eve) (Just kronos)
molly  = Sheep "Molly"   (Just holly) (Just roger)
dolly  = Sheep "Dolly"   (Just molly) Nothing
```

```
-- a sheep has its name, and maybe mother and father
data Sheep = Sheep {name::String, mother::Maybe Sheep, father::Maybe Sheep}
                        deriving (Eq)
```

starý otec z matkinej strany

```
maternalGrandfather :: Sheep -> Maybe Sheep
maternalGrandfather' o = if mother o == Nothing then               -- klasicky:
                                   Nothing
                              else
                                   father (fromJust (mother o))

maternalGrandfather s =  do{ m <- mother s ;                        -- monadicky:
                             father m }
```

maternalGrandfather dolly = Just “Roger”

matky otca otec

```
mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = do { m  <- mother s ;
                                    gf <- father m ;
                                    father gf }
```

mothersPaternalGrandfather dolly = Just “Kronos”

List monad

- List monad použijeme, ak simulujeme nedeterministický výpočet

```
data List a = Null | Cons a (List a) deriving (Show)        -- alias [a]

instance Functor List where                    -- to je vlastne map
    fmap f Nil = Nil
    fmap f (Cons x xs) = Cons (f x) (fmap f xs)

instance Monad List where
    return x     = [x]                         :: a -> [a]
    m >>= f      = concatMap f m               :: [a] -> (a -> [b]) -> [b]
    concatMap    = concat . map f m
```

```
return :: a -> [a]
>>=    :: [a] -> (a -> [b]) -> [b]
```

List monad

```
type List a         = [a]

instance Functor List where
    fmap = map

instance Monad List where
    return v        = [x]
    [] >>= f        = []
    (x:xs) >>= f    = f x ++ (xs >>= f)        -- concatMap f (x:xs)

instance MonadPlus List where
    mzero                   = []
    [] `mplus` ys           = ys
    (x:xs) `mplus` ys       = x : (xs `plus` ys)  -- mplus je klasický append
```

List monad - vlastnosti

Príklad, tzv. listMonad M a = List a = [a]

return x = [x] :: a -> [a]

m >>= f = concatMap f m :: [a] -> (a -> [b]) -> [b]

concatMap = concat . map f m

Cvičenie: overme platnosť zákonov:

- return c >>= (\x->g) **=** g[x/c]
  - [c] >>= (\x->g) = concatMap (\x->g) [c] = concat . map (\x->g) [c] = concat [g[x/c]] = g[x/c]
- m >>= \x->return x **=** m
  - [c_1, … ,c_n] >>= (\x->return x) = concatMap (\x->return x) [c_1, … ,c_n] = concat . map (\x->return x) [c_1, … ,c_n] = concat [[c_1], … ,[c_n]] = [c_1, … ,c_n]
- m1 >>= (****x->m2 >>= (****y->m3)) **=** (m1 >>= (****x->m2)) >>= (****y->m3)
  - ([c_1, … ,c_n] >>= (\x->[d_1, … ,d_m])) >>= (\y->m3) =
    (concat [[d_1[x/c_1], … ,d_m[x/ c_1]], … [d_1[x/ c_n], … ,d_m[x/ c_n]]]) >>= (\y->m3) =
    ([d_1[x/c_1], … ,d_m[x/ c_1], …, d_1[x/ c_n], … ,d_m[x/ c_n]]) >>= (\y->m3) =
    ([d_1[x/c_1], … ,d_m[x/ c_1], …, d_1[x/ c_n], … ,d_m[x/ c_n]]) >>= (\y->[e_1, … ,e_k]) = …
    [e_i[y/d_j[x/c_l]]]

Zákony monadPlus pre List

- vlastnosti zero a `plus`:

```
zero `plus` p           = p                     -- [] ++ p = p
p `plus` zero           = p                     -- p ++ [] = p
p `plus` (q `plus` r)   = (p `plus` q) `plus` r     -- asociativita ++
```

- vlastnosti zero `plus` a >>= :

```
zero >>= f              = zero                  -- concat . map f [] = []
p >>= (\x->zero)        = zero                  -- concat . map (\x->[]) p = []
(p `plus` q) >>= f      = (p >>= f) `plus` (q >>= f)
                                                -- concat . map f (p ++ q) =
                                                        concat . map f p
                                                        ++
                                                        concat . map f q
```

List monad vs. comprehension

```
squares lst = do      x <- lst
                      return (x * x)
-- vlastne znamená
squares lst =         lst >>= \x -> return (x * x)
-- po dosadení
squares lst = concat . map (\x -> [x * x]) lst
-- eta redukcia
squares = concat . map (\x -> [x * x])
-- a takto by sme to napísali bez všetkého
squares = map (\x -> x * x)

-- iný príklad: kartézsky súčin
cart xs ys  =   do  x <- xs
                    y <- ys
                    return (x,y)
```

Sheep family

```
adam   = Sheep "Adam"    Nothing Nothing
eve    = Sheep "Eve"     Nothing Nothing
uranus = Sheep "Uranus"  Nothing Nothing
gaea   = Sheep "Gaea"    Nothing Nothing
kronos = Sheep "Kronos"  (Just gaea) (Just uranus)
holly  = Sheep "Holly"   (Just eve) (Just adam)
roger  = Sheep "Roger"   (Just eve) (Just kronos)
molly  = Sheep "Molly"   (Just holly) (Just roger)
dolly  = Sheep "Dolly"   (Just molly) Nothing
```

data Sheep = Sheep {**name**::String, **mother**::Maybe Sheep, **father**::Maybe Sheep}

```
parents_ :: Sheep -> [Maybe Sheep]
parents_ x = [father x, mother x]
```

parents_ dolly = [Nothing,Just "Molly"]

```
parents :: Sheep -> Maybe [Sheep]
parents x = sequence [father x, mother x]
```

parents dolly = Nothing

parents roger = Just ["Kronos","Eve"]

```
parents' :: Sheep -> [Sheep]
parents' x = (if father x == Nothing then [] else [ fromJust (father x) ])  ++
             (if mother x == Nothing then [] else [ fromJust (mother x) ])
```

parents' dolly = ["Molly"]

parents' roger = ["Kronos","Eve"]

```
parents'' :: Sheep -> Maybe [Sheep]
parents'' x = do { o<-father x; return [o] } `mplus` (do m<-mother x; return [m])
```

parents'' dolly = Just ["Molly"]

parents'' roger = Just ["Kronos"]

```
parents''' :: Sheep -> Maybe [Sheep]
parents''' x = do { o<-father x; m<-mother x; return ([o] `mplus` [m]) }
```

parents''' dolly = Nothing

parents''' roger = Just ["Kronos","Eve"]

Guard
(Control.Monad)

```
pythagoras =   [(x, y, z) |  z <- [1..],             -- pythagorejské trojuholníky
                             x <- [1..z],
                             y <- [x..z],
                             x*x+y*y == z*z]
pythagoras' =   do  z <- [1..]
                    x <- [1..z]
                    y <- [x..z]                                  -- zlé riešenie, prečo ?
                    if x*x+y*y == z*z then  return (x,y,z)  else  return ()

pythagoras'' ...    if x*x+y*y == z*z then return "hogo-fogo"    else []
                    return (x,y,z)                resp. ["hogo-fogo"]

pythagoras''' ...   if x*x+y*y == z*z then return () else []
                    resp. guard (x*x+y*y == z*z)
                    return (x,y,z)
```

Guard

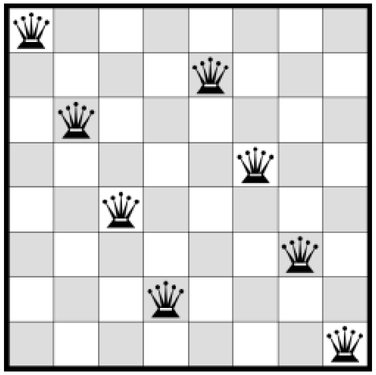
(Control.Monad)

```
guard :: (MonadPlus m) => Bool -> m ()
guard True   = return ()
guard False  = mzero
> guard (9 >5) >> return "hogo" :: [String]
["hogo"]
> guard (5 > 9) >> return "fogo" :: [String]
[]
```

Kartézsky súčin

```
listComprehension xs ys = [(x,y) | x<-xs, y<-ys ]
listComprehension [1,2,3] ['a','b','c'] = [(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c'),(3,'a'),(3,'b'),(3,'c')]

guardedListComprehension xs ys = [(x,y) | x<-xs, y<-ys, x<=y, x*y == 24 ]
guardedListComprehension [1..10] [1..10]          = [(3,8),(4,6)]

monadComprehension xs ys = do { x<-xs; y<-ys; return (x,y) }
monadComprehension [1,2,3] ['a','b','c'] = [(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c'),(3,'a'),(3,'b'),(3,'c')]

guardedMonadComprehension xs ys =
        do { x<-xs; y<-ys; guard (x<=y); guard (x*y==24); return (x,y) }
guardedMonadComprehension [1..10] [1..10]        = [(3,8),(4,6)]
```

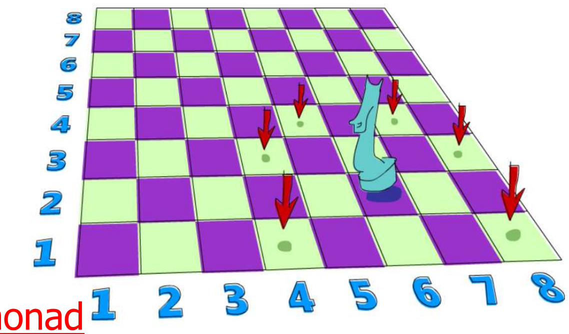
Backtracking



```
check (i,j) (m,n)      = (i==m) || (j==n) || (j+i==n+m) || (j+m==i+n)

safe p n   = all (True==)  [not (check (i,j) (m+1,n)) | (i,j) <- zip [1..m] p]
                  where m=length p

-- backtrack
queens n = queens1 n n
queens1 n v   | n==0 = [[]]
              | otherwise = [y++[p] | y <- queens1 (n-1) v, p <- [1..v], safe y p]
length $ queens 10 = 724 (3.16 secs, 1,897,245,320 bytes)

mqueens n = mqueens1 n n
mqueens1 n v        | n==0 = return []
                    | otherwise =   do   y <- mqueens1 (n-1) v
                                         p <- [1..v]
                                         guard (safe y p)
                                         return (y++[p])
length $ mqueens 10 = 724 (3.12 secs, 1,954,471,256 bytes)
```

Kôň

zdroj : http://learnyouahaskell.com/a-fistful-of-monads#the-list-monad




```
type KnightPos = (Int,Int)

moveKnight :: KnightPos -> [KnightPos]
moveKnight (c,r) = do (c',r') <- [(c+2,r-1),(c+2,r+1),(c-2,r-1),(c-2,r+1),(c+1,r-2),(c+1,r+2),(c-1,r-2),(c-1,r+2)]
                      guard (c' `elem` [1..8] && r' `elem` [1..8])
                      return (c',r')
```

kam sa dostane kôň na k krokov

```
ink :: Int -> KnightPos -> [KnightPos]
ink 0 start = return start
ink k start = do m <- moveKnight start
                 mm <- ink (k-1) m
                 return mm
```

length $ ink 7 (0,0) = 45016

length $ nub $ ink 7 (0,0) = 32

filterM
(Control.Monad)

```
filterM     :: (Monad m) => (a -> m Bool) -> [a] -> m [a]

> filterM (\x->[True, False]) [1,2,3]            -- potenčná množina, powerset
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]

filterM          :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
filterM _ []      =  return []
filterM p (x:xs) =  do
                                flg <- p x
                                ys  <- filterM p xs
                                return (if flg then x:ys else ys)
```

mapM, forM
(Control.Monad)

```
mapM     :: (Monad m) => (a -> m b) -> [a] -> m [b]
mapM f   = sequence . map f

forM      :: (Monad m) => [a] -> (a -> m b) -> m [b] -- len zámena args.
forM      = flip mapM
```

```
> mapM (\x->[x,11*x]) [1,2,3]
[[1,2,3],[1,2,33],[1,22,3],[1,22,33],[11,2,3],[11,2,33],[11,22,3],[11,22,33]]

> forM [1,2,3] (\x->[x,11*x])
[[1,2,3],[1,2,33],[1,22,3],[1,22,33],[11,2,3],[11,2,33],[11,22,3],[11,22,33]]

> mapM print [1,2,3]
1
2
3
[(),(),()]
```

```
> mapM_ print [1,2,3]
1
2
3
```

foldM
(Control.Monad)

```
foldM              :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
foldM _ a []       = return a
foldM f a (x:xs)   = f a x >>= \y -> foldM f y xs
```

foldM f a_1 [x_1, …, x_n] =
    do {
        a_2 <- f a_1 x_1;
        a_3 <- f a_2 x_2;
        …
        a_n <- f a_{n-1} x_{n-1};
        return f a_n x_n }

```
> foldM (\y -> \x ->
      do { print (show x++"..."++ show y);
           return (x*y)})
    1 [1..10]
???
```

```
> foldM (\y -> \x ->  do print (show x++"..."++ show y); return (x*y))  1 [1..10]
???
```

```
newtype Either a b = Right a | Left b
instance (Error e) => Monad (Either e) where
      return x = Right x
      Right x >>= f = f x
      Left err >>= f = Left err
      fail msg = Left (strMsg msg)
```

Error monad

```
data Term = Con Int | Div Term Term deriving(Show, Read, Eq)

eval          :: Term -> Either String Int
eval(Con a)   = return a
eval(Div t u) = do
                  valT <- eval t
                  valU <- eval u
                  if valU == 0 then
                     fail "div by zero"
                  else
                     return (valT `div` valU)
> eval (Div (Con 1972) (Con 23))
Right 85
> eval (Div (Con 1972) (Con 0))
*** Exception: div by zero
```

Writer monad
(Control.Monad.Writer)

```
newtype Writer w a = Writer { runWriter :: (a, w) }
instance (Monoid w) => Monad (Writer w) where
     return x = Writer (x, mempty)
     (Writer (x,v)) >>= f =
         let (Writer (y, v')) = f x in Writer (y, v `mappend` v')
```

```
data Term = Con Int | Div Term Term deriving(Show, Read, Eq)

out :: Int -> Writer [String] Int
out x = writer (x, ["number: " ++ show x])

eval          :: Term -> Writer [String] Int
eval(Con a)   = out a
eval(Div t u) = do
                   valT <- eval t
                   valU <- eval u
                   out (valT `div` valU)
                   return (valT `div` valU)
```

```
> eval (Div (Con 1972) (Con 23))
WriterT (Identity (85,["number: 1972","number: 23","number: 85"]))
> runWriter  $ eval (Div (Con 1972) (Con 23))
(85,["number: 1972","number: 23","number: 85"])
```

Writer monad

(Control.Monad.Writer)

```
-- tell :: MonadWriter w m => w -> m ()

gcd' :: Int -> Int -> Writer [String] Int
gcd' a b  | b == 0 = do
                  tell ["result " ++ show a]
                  return a
          | otherwise = do
                  tell [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)]
                  gcd' b (a `mod` b)
```

```
> gcd' 18 12
WriterT (Identity (6,["18 mod 12 = 6","12 mod 6 = 0","result 6"]))

> runWriter (gcd' 2016 48)
(48,["2016 mod 48 = 0","result 48"])

> mapM putStrLn (snd $ runWriter (gcd' 2016 48))
2016 mod 48 = 0
result 48
[(),()]
```

IO Monad

```
type Kopa = Int

finished :: Kopa -> Bool                    -- kedy hra konci
finished = (== 0)

valid :: Kopa -> Int -> Bool     -- korektny tah
valid kopa num = (kopa >= num) && num < 4

getDigit :: String -> IO Int
getDigit prompt = do putStr prompt
                     x <- getChar
                     if isDigit x
                       then return (digitToInt x)
                     else
                       getDigit ""
```

IO Monad

(dvaja hráči)

```
play2 :: Kopa -> Bool -> IO ()
play2 kopa hrac =
  do putStrLn ("kopa:" ++ (show kopa))
    if finished kopa then
      putStrLn ("Hrac " ++ (show (not $ hrac)) ++ " vyhral!")
    else
      do putStrLn ("Ide hrac " ++ (show hrac))
        num <- getDigit "kolko beries : "
        if valid kopa num then
          play2 (kopa - num) (not $ hrac)
        else
          do putStrLn "zly tah"
            play2 kopa hrac

nim2 :: IO ()
nim2 = play2 (nextInt 10 20) True
```

IO Monad
(jeden hráč proti kompu)

```
play1 :: Kopa -> IO ()
play1 kopa =
  do putStrLn ("kopa:" ++ (show kopa))
    if finished kopa then
      putStrLn "prehral si :("
    else
      do num <- getDigit "kolko beries : "
        if valid kopa num then
          let kopa' = kopa - num in
             if finished kopa' then
               putStrLn "vyhral si :)"
             else
               do putStrLn ("ja beriem:" ++ (show (kopa' - (strategia kopa'))))
                 play1 (strategia kopa')
        else
          do putStrLn "zly tah"
            play1 kopa
```

```
strategia :: Int -> Int
strategia kopa
   | kopa `mod` 4 == 0 = kopa-1
   | otherwise = kopa - (kopa `mod` 4)

nim1 :: IO ()
nim1 = play1 (nextInt 10 20)
```

State monad
(Control.Monad.State)

```
newtype State s a = State { runState :: (s -> (a,s)) }

instance Monad (State s) where
   return a          = State \s -> (a,s)
   (State x) >>= f = State \s ->
                                 let (v,s') = x s in runState (f v) s',

class (Monad m) => MonadState s m | m -> s where
   get :: m s                         -- get vráti stav z monády
   put :: s -> m ()                   -- put prepíše stav v monáde

modify :: (MonadState s m) => (s -> s) -> m ()
modify f = do    s <- get
                 put (f s)
```

newtype

newtype State s a = State { runState :: (s -> (a,s)) }

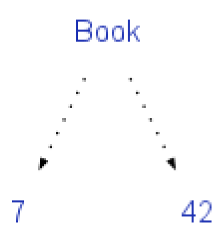
State s a má rovnakú reprezentáciu ako (s -> (a,s))

data State s a = State { runState :: (s -> (a,s)) }

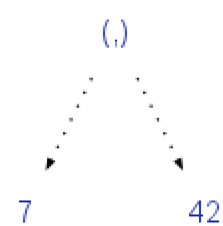
State s a je reprezentovaná krabicou State s pointrom na (s -> (a,s))

Príklad:

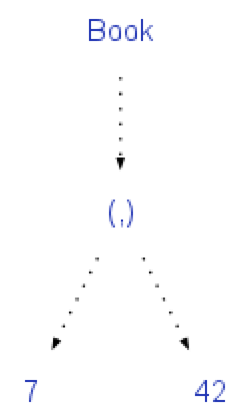
data Book = Book Int Int



newtype Book = Book (Int, Int)



data Book = Book (Int, Int)



http://stackoverflow.com/questions/5889696/difference-between-data-and-newtype-in-haskell

State Stack

```haskell
pop :: State Stack Int
pop = state(\(x:xs) -> (x,xs))

push :: Int -> State Stack ()
push a = state(\xs -> ((),a:xs))
```

```haskell
type Stack = [Int]

pushAll :: Int -> State Stack String
pushAll 0    = return ""
pushAll n    = do
               push n
               str <- pushAll (n-1)
               nn <- pop
               return (show nn ++ str)
```

```
> evalState (pushAll 10) []
"10987654321"
> execState (pushAll 10) []
[]
```

```haskell
type Stack = [Int]

pushAll' :: Int -> State Stack String
pushAll' 0    = return ""
pushAll' n    = do
                stack <- get  -- push n
                put (n:stack)
                str <- pushAll (n-1)
                (nn:stack') <- get  -- nn <- pop
                put stack'
                return (show nn ++ str)
```

```
> evalState (pushAll' 10) []
"10987654321"
> execState (pushAll' 10) []
[]
```

Preorder so stavom

(Control.Monad.State)

```
data Tree a =       Nil |
                    Node a (Tree a) (Tree a)
                    deriving (Show, Eq)

preorder :: Tree a -> State [a] ()                  -- stav a výstupná hodnota
preorder Nil                              = return ()
preorder (Node value left right)          =
                                          do {
                                                 str<-get; -- get state=preorderlist
                                                 put (value:str);  -- modify (value:)
                                                 preorder left;
                                                 preorder right;
                                                 return () }
```

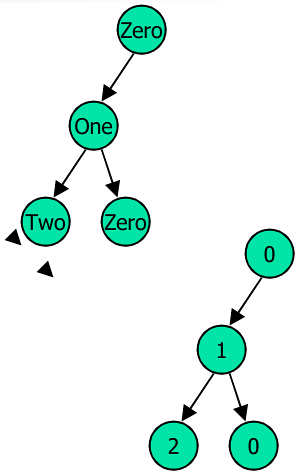
```
e :: Tree String
e = Node "c" (Node "a" Nil Nil) (Node "b" Nil Nil)

> execState (preorder e) []
["b","a","c"]

> evalState (preorder e) []
()
```

Prečíslovanie binárneho stromu

```
index :: Tree a -> State Int (Tree Int)          -- stav a výstupná hodnota
index Nil              = return Nil
index (Node value left right) =
                         do {
                                 i <- get;
                                 put (i+1);
                                 ileft <- index left;
                                 iright <- index right;
                                 return (Node i ileft iright) }
```

```
> e'
Node "d" (Node "c" (Node "a" Nil Nil) (Node "b" Nil Nil)) (Node "c" (Node "a" Nil
    Nil) (Node "b" Nil Nil))

> evalState (index e') 0
Node 0 (Node 1 (Node 2 Nil Nil) (Node 3 Nil Nil)) (Node 4 (Node 5 Nil Nil) (Node 6
    Nil Nil))

> execState (index e') 0
7
```

Prečíslovanie stromu 2




```
type Table a = [a]

numberTree :: Eq a => Tree a -> State (Table a) (Tree Int)
numberTree Nil                  = return Nil
numberTree (Node x t1 t2)       =  do   num <- numberNode x
                                        nt1 <- numberTree t1
                                        nt2 <- numberTree t2
                                        return (Node num nt1 nt2)
    where
    numberNode :: Eq a => a -> State (Table a) Int
    numberNode x                = do    table <- get
                                        (newTable, newPos) <- return (nNode x table)
                                        put newTable
                                        return newPos

    nNode::  (Eq a) => a -> Table a -> (Table a, Int)
    nNode x table               = case (findIndexInList (== x) table) of
                                        Nothing -> (table ++ [x], length table)
                                        Just i  -> (table, i)
```

Prečíslovanie stromu 2

```
numTree :: (Eq a) => Tree a -> Tree Int
numTree t = evalState (numberTree t) []
```

```
> numTree ( Node "Zero"
                  (Node "One" (Node "Two" Nil Nil)
                  (Node "One" (Node "Zero" Nil Nil) Nil)) Nil)

Node 0 (Node 1 (Node 2 Nil Nil) (Node 1 (Node 0 Nil Nil) Nil)) Nil
```