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)
```

Čo je newtype vs. data vs. type

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

State s a má rovnakú reprezentáciu ako (s -> (a,s)), ale nie je to

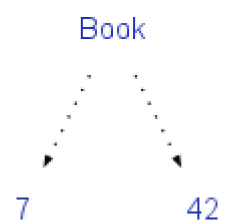
type State s a = s -> (a,s)

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

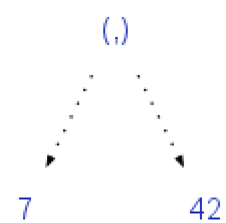
State s a je reprezentovaná krabicou State s pointrom na (s -> (a,s))

Príklad:

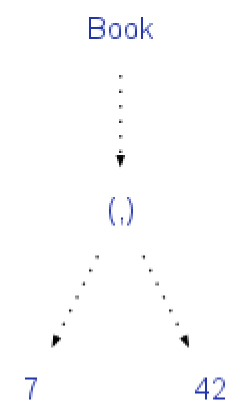
data Book = Book Int Int



newtype Book = Book (Int, Int)



data Book = Book (Int, Int)



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

State s a

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

- runState :: State s a -> s -> (a, s) -- vráti funkciu state monády
- evalState :: State s a -> s -> a -- vráti výsledok state monády pre stav s
- execState :: State s a -> s -> s -- vráti výsledný stav state monády pre vstupný stav s

```
:t runState ((return "hello") :: State Int String)
runState ((return "hello") :: State Int String) :: Int -> (String, Int)

runState ((return "hello") :: State Int String) 77  = ("hello",77)
evalState ((return "hello") :: State Int String) 77  = "hello"
execState ((return "hello") :: State Int String) 77  = 77
```

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

State s a

```
return :: a -> State s a
return x s = (x,s)                     -- return x = \s -> (x,s)

get :: State s s          -- stav state monády je jej výsledkom
get s = (s,s)                          -- get = \s -> (s,s)
runState get 1 = (1,1)

put :: s -> State s ()    -- prepíše stav state monády x
put x s = ((),x)                       -- put x = \s -> ((),x)
runState (put 5) 1 = ((),5)
runState (do { put 5; return 'X' }) 1 = ('X',5)

modify :: (s -> s) -> State s ()
modify f = do { x <- get; put (f x) }
runState (modify (+3)) 1 = ((),4)
runState (do { modify (+3); return "hello"}) 1 = ("hello",4)
```

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

State s a

let increment = do { x <- get; put (x+1); return x } in runState increment 77
= (77,78)

gets :: (s -> a) -> State s a
gets f = do { x <- get; return (f x) }

runState (gets (+1)) 77 = (78,77)

evalState (gets (+1)) 77 = 78 -- vráti výsledok state monády pre stav s
execState (gets (+1)) 77 = 77 -- vráti výsledný stav state monády pre vstupný stav s

runState (modify (+1)) 77 = ((),78)

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

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

State Stack

```
type Stack = [Int]
                         stav   výsledok

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

evalState vráti výslednú hodnotu

> **evalState** (pushAll 10) []

"10987654321"

execState vráti výsledný stav

> **execState** (pushAll 10) []

[]

```
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) []
()
```

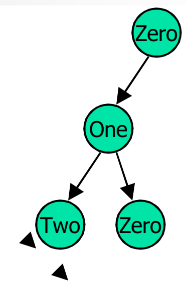
Prečíslovanie binárneho stromu

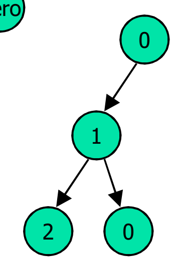
```
index :: Tree a -> State Int (Tree Int)          -- stav a výstupná hodnota
index Nil              = return Nil
index (Node value left right) =
                         do {
                                     i <- get;
                                     put (i+1);
                           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

Stovkári

...	Meno	Priezvisko	Spolu ▼
1.	Róbert	Ruska	**119.02**
2.	Matej Zajo	Králik	**116.23**
3.	Samuel	Sladek	**114.775**
4.	Vician	Tomáš	**107.26**
5.	Emanuel	Tesař	**104.905**
6.	Jaroslava	Kokavcova	**102.85**
7.	Adam	Halász	**102.41**