

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://www-users.mat.uni.torun.pl/~fly/materialy/fp/haskell-doc/Monads.html>

- Jeff Newbern's: All About Monads
- 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.

return :: a -> M a
>>= :: M a -> (a -> M b) -> M b

Základný interpreter výrazov

Princíp fungovania monád sme trochu ilustrovali na type

M result = Parser result = String -> [(result, String)]

return :: a -> Parser a

return v = \xs -> [(v,xs)]

bind, >>= :: Parser a -> (a -> Parser b) -> Parser b

p >>= qf = \xs -> concat [(qf v) xs' | (v,xs') <- p xs]

... len sme nepovedali, že je to monáda

dnes vysvetlíme na sérii evaluátorov aritmetických výrazov,
presnejšie zredukovaných len na konštrukcie pozostávajúce z Con a Div:

data Term = Con Int | Div Term Term | Add ... | Sub ... | Mult ...
deriving(Show, Read, Eq)

eval :: Term -> Int

eval(Con a) = a

eval(Div t u) = eval t `div` eval u

> eval (Div (Div (Con 1972) (Con 2)) (Con 23))

```
data Either a b = Left a | Right b
data Maybe a   = Nothing | Just a
```



Interpreter s výnimkami

v prvej verzii interpretera riešime problém, ako ošetriť delenie nulou

Toto je výstupný typ nášho interpretera:

```
data M1 a      = Raise String | Return a   deriving(Show, Read, Eq)
```

```
evalExc        :: Term -> M1 Int
```

```
evalExc(Con a)  = Return a
```

```
evalExc(Div t u) = case evalExc t of
```

```
    Raise e -> Raise e
```

```
    Return a ->
```

```
        case evalExc u of
```

```
            Raise e -> Raise e
```

```
            Return b ->
```

```
                if b == 0
```

```
                then Raise "div by zero"
```

```
                else Return (a `div` b)
```

```
> evalExc (Div (Div (Con 1972) (Con 2)) (Con 23))
```

```
Return 42
```

```
> evalExc (Div(Con 1)(Con 0))
```

```
Raise "div by zero"
```



Interpreter so stavom

interpreter výrazov, ktorý počíta počet operácii div (má stav State = Int):

naivne:

`evalCnt :: (Term, State) -> (Int, State)`

resp.:

`evalCnt :: Term -> State -> (Int, State)`

M_2 a - reprezentuje výpočet s výsledkom typu a, lokálnym stavom State ako:

`type M_2 a = State -> (a, State)`
`type State = Int`

`evalCnt :: Term -> M_2 Int`

`evalCnt (Con a) st = (a, st)`

`evalCnt (Div t u) st = let (a, st1) = evalCnt t st in
let (b, st2) = evalCnt u st1 in
(a `div` b, st2+1)`

výsledkom evalCnt t
je funkcia, ktorá po
zadaní počiatočného
stavu povie výsledok
a konečný stav

`> evalCnt (Div (Div (Con 1972) (Con 2)) (Con 23)) 0
(42,2)`



Interpreter s výstupom

tretia verzia je interpreter výrazov, ktorý vypisuje debug.informáciu do reťazca

```
type M3 a      = (Output, a)
type Output     = String

> evalOut (Div (Div (Con 1972) (Con 2)) (Con 23))
("eval (Con 1972) <=1972
eval (Con 2) <=2
eval (Div (Con 1972) (Con 2)) <=986
eval (Con 23) <=23
eval (Div (Div (Con 1972) (Con 2)) (Con 23)) <=42",42)

evalOut          :: Term -> M3 Int
evalOut (Con a)  = (out_a, a)
                  where out_a = line (Con a) a
evalOut (Div t u) = let (out_t, a) = evalOut t in
                  let (out_u, b) = evalOut u in
                  (out_t ++ out_u ++ line (Div t u) (a `div` b), a `div` b)

line            :: Term -> Int -> Output
line t a       = "eval (" ++ show t ++ ") <=" ++ show a ++ "\n"
```



Monadický interpreter

(vícia)

- máme 1+3 verzie interpretera,
- cieľom je napísať jednu, skoro uniformú verziu, z ktorej všetky existujúce vypadnú ako inštancia s malými modifikáciami
- potrebujeme pochopiť typ/triedu/interface nazývaný monáda

```
class Monad m where
  return  :: a -> m a
  >>=    :: m a -> (a -> m b) -> m b
```

- a potrebujeme pochopiť, čo je inštancia triedy

```
instance Monad Mi where
  return = ...
  >>=   = ...
```

Cieľ: ukážeme, ako v monádach s typmi **M0**, **M1**, **M2**, **M3** dostaneme požadovaný interpreter ako inštanciu všeobecného monadického interpretera



Functor – definícia

Zoberme jednoduchšiu triedu, z modulu Data.Functor je definovaná takto:

```
class Functor f where                                -- musí mať funkciu fmap s profilom
  fmap :: (a -> b) -> f a -> f b
```

a každá jej inštancia musí spĺňať dve pravidlá (to je sémantika, mimo syntaxe)

- $\text{fmap id} = \text{id}$
- $\text{fmap } (p . q) = (\text{fmap } p) . (\text{fmap } q)$

Príklad inštancie pre typ M1 (overte, že platia obe pravidlá):

```
data M1 a      =  Raise String | Return a  deriving(Show, Read, Eq)
instance Functor M1 where
  fmap f (Raise str)    =  Raise str
  fmap f (Return x)     =  Return (f x)
```



Functor – príklad

Skúste definovať inštanciu triedy Functor pre typy:

```
data MyMaybe a = MyJust a | MyNothing deriving (Show)      -- alias Maybe a
data MyList a = Nil | Cons a (MyList a) deriving (Show)      -- alias [a]
```

```
> fmap (\s -> even s) (Cons 1 (Cons 2 Nil))                -- f : Int->Bool
```

```
Cons False (Cons True Nil)
```

```
> fmap (\s -> s+s) (Cons 1 (Cons 2 Nil))                    -- f : Int->Int
```

```
Cons 2 (Cons 4 Nil)
```

```
> fmap (\s -> show s) (Cons 1 (Cons 2 Nil))                 -- f : Int->String
```

```
Cons "1" (Cons "2" Nil)
```

```
> fmap ((\t -> t++t) . (\s -> show s)) (Cons 1 (Cons 2 Nil)) -- f : (String->String).(Int->String)
```

```
Cons "11" (Cons "22" Nil)
```

```
> fmap (\t -> t++t) (fmap (\s -> show s) (Cons 1 (Cons 2 Nil)))
```

```
Cons "11" (Cons "22" Nil)
```

```
> fmap id (Cons 1 (Cons 2 Nil))
```

```
Cons 1 (Cons 2 Nil)
```




Functor – strom

- Binárny:

```
data LExp a = Var a | Appl (LExp a) (LExp a) | Abs a (LExp a) deriving (Show)
instance Functor LExp where
```

<code>fmap f (Var x)</code>	<code>= Var (f x)</code>
<code>fmap f (Appl left right)</code>	<code>= Appl (fmap f left) (fmap f right)</code>
<code>fmap f (Abs x right)</code>	<code>= Abs (f x) (fmap f right)</code>

```
omega = Abs "x" (Appl (Var "x") (Var "x"))
```

```
> fmap (\t -> t++t) omega
```

```
Abs "xx" (Appl (Var "xx") (Var "xx"))
```

ľubovoľne-árny strom (prezývaný RoseTree alias Rhododendron):

```
data RoseTree a = Node a [RoseTree a]
```

```
instance Functor RoseTree where
```

<code>fmap f (Node a bs)</code>	<code>= Node (f a) (map (fmap f) bs)</code>
---------------------------------	---



Monáda

monáda je iná trieda parametrizovaná typom a pozostáva z dvoch funkcií:

```
class Monad m where
```

```
  return :: a -> m a
```

```
  >>=    :: m a -> (a -> m b) -> m b
```

ktoré spĺňajú isté zákony:

- $\text{return } c \gg= (\backslash x \rightarrow g)$ $=$ $g[x/c]$
- $m \gg= \backslash x \rightarrow \text{return } x$ $=$ m
- $m1 \gg= (\backslash x \rightarrow m2 \gg= (\backslash y \rightarrow m3)) = (m1 \gg= (\backslash x \rightarrow m2)) \gg= (\backslash y \rightarrow m3)$

inak zapísané:

```
return c >>= f
```

```
= f c
```

-- ľavo neutrálny prvok

```
m >>= return
```

```
= m
```

-- pravo neutrálny prvok

```
(m >>= f) >>= g
```

```
= m >>= (\x -> f x >>= g)
```

-- asociativita >>=



ListMonad

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

$return\ x = [x]$

$:: a \rightarrow [a]$

$m \gg= f = concatMap\ f\ m$

$:: [a] \rightarrow (a \rightarrow [b]) \rightarrow [b]$

$concatMap = concat . map\ f\ m$

overme platnosť zákonov:

■ $return\ c \gg= (\backslash x \rightarrow g) = g[x/c]$

■ $[c] \gg= (\backslash x \rightarrow g) = concatMap\ (\backslash x \rightarrow g)\ [c] = concat . map\ (\backslash x \rightarrow g)\ [c] = concat\ [g[x/c]] = g[x/c]$

■ $m \gg= \backslash x \rightarrow return\ x = m$

■ $[c_1, \dots, c_n] \gg= (\backslash x \rightarrow return\ x) = concatMap\ (\backslash x \rightarrow return\ x)\ [c_1, \dots, c_n] = concat . map\ (\backslash x \rightarrow return\ x)\ [c_1, \dots, c_n] = concat\ [[c_1], \dots, [c_n]] = [c_1, \dots, c_n]$

■ $m1 \gg= (\backslash x \rightarrow m2 \gg= (\backslash y \rightarrow m3)) = (m1 \gg= (\backslash x \rightarrow m2)) \gg= (\backslash y \rightarrow m3)$

■ $([c_1, \dots, c_n] \gg= (\backslash x \rightarrow [d_1, \dots, d_m])) \gg= (\backslash y \rightarrow m3) = (concat\ [[d_1[x/c_1], \dots, d_m[x/c_1]], \dots, [d_1[x/c_n], \dots, d_m[x/c_n]]]) \gg= (\backslash y \rightarrow m3) = ([d_1[x/c_1], \dots, d_m[x/c_1], \dots, d_1[x/c_n], \dots, d_m[x/c_n]]) \gg= (\backslash y \rightarrow m3) = ([d_1[x/c_1], \dots, d_m[x/c_1], \dots, d_1[x/c_n], \dots, d_m[x/c_n]]) \gg= (\backslash y \rightarrow [e_1, \dots, e_k]) = \dots [e_i[y/d_j[x/c_i]]]$

monadický znamená, že je typu,
ktorá je inštanciou triedy Monad



Monadický interpreter

```
class Monad m where  
  return  :: a -> m a  
  >>=    :: m a -> (a -> m b) -> m b
```

ukážeme, ako v monádach s typmi M_0, M_1, M_2, M_3 dostaneme požadovaný
interpreter ako inštanciu všeobecného monadického interpretera:
instance Monad M_i where return = ... , $\gg =$...

eval	:: Term -> M_i Int
eval (Con a)	= return a
eval (Div t u)	= eval t $\gg =$ \valT -> eval u $\gg =$ \valU -> return(valT `div` valU)

čo vďaka *do* notácii zapisujeme:

```
eval (Div t u)    = do { valT<-eval t; valU<-eval u; return(valT `div` valU) }
```

```
return :: a -> M a
>>=   :: M a -> (a -> M b) -> M b
```



Identity monad

```
Pre identity monad:
return :: a -> a
>>=   :: a -> (a -> b) -> b
```

na verziu M_0 $a = a$ sme zabudli, volá sa identity monad, resp. $M_0 = \text{Id}$:

```
type Identity a = a           -- trochu zjednodušené oproti monad.hs
```

```
instance Monad Identity where
```

```
    return v          = v
    p >>= f            = f p
```

```
evalIdentM          :: Term -> Identity Int
evalIdentM(Con a)    = return a
evalIdentM(Div t u)  = evalIdentM t >>= \valT->
                        evalIdentM u >>= \valU ->
                        return(valT `div` valU)
```

```
> evalIdentM (Div (Div (Con 1972) (Con 2)) (Con 23))
```



Exception monad

```
return :: a -> M a
>>=   :: M a -> (a -> M b) -> M b
```

```
Pre Exception monad:
return :: a -> Exception a
>>=   :: Exception a ->
        (a -> Exception b) ->
        Exception b
```

```
data M1 = Exception a = Raise String | Return a deriving(Show, Read, Eq)
```

```
instance Monad Exception where
```

```
  return v    = Return v
```

```
  p >>= f    = case p of
                  Raise e -> Raise e
                  Return a -> f a
```

```
> evalExceptM (Div (Div (Con 1972)
                      (Con 2)) (Con 23))
```

```
Return 42
```

```
> evalExceptM (Div (Div (Con 1972)
                      (Con 2)) (Con 0))
```

```
Raise "div by zero"
```

```
evalExceptM      :: Term -> Exception Int
```

```
evalExceptM(Con a) = return a
```

```
evalExceptM(Div t u) = evalExceptM t >>= \valT->
                        evalExceptM u >>= \valU ->
                        if valU == 0 then Raise "div by zero"
                        else return(valT `div` valU)
```

```
evalExceptM (Div t u) = do valT<-evalExceptM t
                          valU<-evalExceptM u
                          if valU == 0 then Raise "div by zero"
                          else return(valT `div` valU)
```

```
return :: a -> M a
>>=   :: M a -> (a -> M b) -> M b
```

State monad

```
data M2 = SM a = SM(State -> (a, State)) -- funkcia obalená v konštruktore SM
                                           -- type State = Int
```

instance Monad SM where

```
return v      = SM (\st -> (v, st))
(SM p) >>= f  = SM (\st -> let (a,st1) = p st in
                           let SM g = f a in
                           g st1)
```

pomôcka:

```
p::State->(a,State)
f::a->SM(State->(a,State))
g::State->(a,State)
```

```
evalSM      :: Term -> SM Int
evalSM(Con a) = return a
evalSM(Div t u) = evalSM t >>= \valT ->
                  evalSM u >>= \valU ->
                  incState >>= \_ ->
                  return(valT `div` valU)
```

-- Int je typ výsledku

```
-- evalSM t :: SM Int
-- valT :: Int, valU :: Int
-- ():()
```

```
incState      :: SM ()
incState      = SM (\s -> ((),s+1))
```



do notácia

```
evalSM'          :: Term -> SM Int
evalSM'(Con a)   = return a
evalSM'(Div t u) = do { valT<-evalSM' t;
                       valU<-evalSM' u;
                       incState;
                       return(valT `div` valU) }
```

Problémom je, že výsledkom evalSM, resp. evalSM', nie je stav, ale stavová monada SM Int, t.j. niečo ako $\text{SM}(\text{State} \rightarrow (\text{Int}, \text{State}))$.

Preto si definujeme pomôcku, podobne ako pri parseroch:

```
goSM'            :: Term -> State
goSM' t          = let SM p = evalSM' t in
                    let (result,state) = p 0 in state
```

```
> goSM' (Div (Div (Con 1972) (Con 2)) (Con 23))
2
```



```
return :: a -> M a
>>=   :: M a -> (a -> M b) -> M b
```



Output monad

```
data M3 = Out a      = Out(String, a)      deriving(Show, Read, Eq)
```

```
instance Monad Out where
```

```
  return v      = Out("",v)
  p >>= f       = let Out (str1,y) = p in
                  let Out (str2,z) = f y in
                  Out (str1++str2,z)
```

```
out      :: String -> Out ()
out s    = Out (s,())
```

```
evalOutM      :: Term -> Out Int
evalOutM(Con a) = do { out(line(Con a) a); return a }
```

```
evalOutM(Div t u) = do { valT<-evalOutM t; valU<-evalOutM u;
                        out (line (Div t u) (valT `div` valU) );
                        return (valT `div` valU) }
```

```
> evalOutM (Div (Div (Con 1972) (Con 2)) (Con 23))
Out ("eval (Con 1972) <=1972
eval (Con 2) <=2
eval (Div (Con 1972) (Con 2)) <=986
eval (Con 23) <=23
eval (Div (Div (Con 1972) (Con 2)) (Con 23)) <=1972")
```



Monadic Prelude

```
class Monad m where
```

```
    return :: a -> m a
```

```
    (>>=) :: m a -> (a -> m b) -> m b
```

```
    (>>)   :: m a -> m b -> m b
```

```
    p >> q = p >>= \_ -> q
```

-- definition:(>>=), return

-- zahodíme výsledok prvej monády

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

```
sequence [] = return []
```

```
sequence (c:cs) = do { x <- c; xs <- sequence cs; return (x:xs) }
```

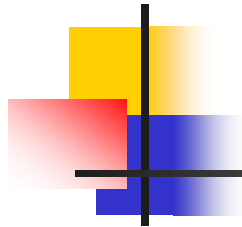
-- ak nezáleží na výsledkoch

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

```
sequence_   = foldr (>>) (return ())
```

```
sequence_ [m1,m2,...mn] = m1 >>= \_ ->  
                             m2 >>= \_ ->  
                             ...  
                             mn >>= \_ ->  
                             return ()
```

```
do { m1 ;  
    m2 ;  
    ...  
    mn ;  
    return ()
```



Kde nájst' v *praxi* monádu ?

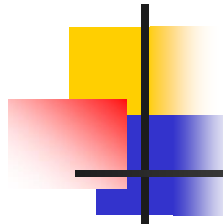
Prvý pokus :-)

```
> sequence [evalExceptM (Div (Div (Con 1972) (Con 2)) (Con 23)),  
            evalExceptM (Div (Con 8) (Con 4)),           :: Exception Int  
            evalExceptM (Div (Con 7) (Con 2))           :: Exception Int  
            ]  
Return [42,2,3] :: Exception [Int]
```

```
> sequence [evalExceptM (Div (Div (Con 1972) (Con 2)) (Con 23)),  
            evalExceptM (Div (Con 8) (Con 4)),  
            evalExceptM (Div (Con 7) (Con 0))  
            ]
```

???

== Raise "div by 0"



IO monáda

Druhý pokus :-)

```
> :type print
print :: Show a => a -> IO ()
> print "Hello world!"
"Hello world!"
```

```
data IO a = ... {- abstract -}
```

```
getChar :: IO Char
putChar :: Char -> IO ()
getLine :: IO String
putStr :: String -> IO ()
```

```
echo = getChar >>= putChar
```

```
do { c<-getChar; putChar c }
```



Interaktívny Haskell

```
main1 = putStr "Please enter your name: " >>
        getLine >>= \name ->
        putStr ("Hello, " ++ name ++ "\n")
```

```
main2 = do
    putStr "Please enter your name: "
    name <- getLine
    putStr ("Hello, " ++ name ++ "\n")
```

```
> main2
Please enter your name: Peter
Hello, Peter
```

```
> sequence [print 1 , print 'a' , print "Hello"]
1
'a'
"Hello"
[(),(),()]
```

sequence :: Monad m => [m a] -> m [a]



Maybe monad

Maybe je podobné Exception (Nothing $\sim\sim$ Raise String, Just a $\sim\sim$ Return a)

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

```
instance Monad Maybe where
```

```
    return v      = Just v
```

-- vrát' hodnotu

```
    fail          = Nothing
```

-- vrát' 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

```
> sequence [Just "a", Just "b", Just "d"]
```

```
Just ["a","b","d"]
```

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

```
Nothing
```



Maybe monad – pokračovanie

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

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

```
    mzero    :: m a
```

```
-- ∅
```

```
    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"
```

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



List monad

```
type List a      = [a]
```

```
instance Monad List where
```

```
    return v      = [v]
```

```
    [] >>= 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 vs. Monad Comprehension

Kartézsky súčin

```
guard :: (MonadPlus m) =>  
          Bool -> m ()  
guard True = return ()  
guard False = mzero
```

```
listComprehension xs ys = [(x,y) | x<-xs, y<-ys ]  
guardedListComprehension xs ys =  
    [(x,y) | x<-xs, y<-ys, x<=y, x*y == 24 ]
```

```
> 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 [1..10] [1..10]  
[(3,8),(4,6)]
```

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



filterM (Control.Monad)

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

```
> filterM (\x->[True, False]) [1,2,3]  
[[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

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

forM :: (Monad m) => [a] -> (a -> m b) -> m [b]
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

```
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 a1 [x1, ..., xn] =
  do {
    a2 <- f a1 x1;
    a3 <- f a2 x2;
    ...
    an <- f an-1 xn-1;
    return f an xn }
```

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



Zákony monád

- vlastnosti return a $\gg=$:

$\text{return } x \gg= f = f \ x$ -- return ako identita zľava
 $p \gg= \text{return} = p$ -- return ako identita sprava
 $p \gg= (\lambda x \rightarrow (f \ x \gg= g)) = (p \gg= (\lambda x \rightarrow f \ x)) \gg= g$ -- "asociativita"

- vlastnosti zero a ``plus``:

$\text{zero} \text{ `plus` } p = p$ -- zero ako identita zľava
 $p \text{ `plus` } \text{zero} = p$ -- zero ako identita sprava
 $p \text{ `plus` } (q \text{ `plus` } r) = (p \text{ `plus` } q) \text{ `plus` } r$ -- asociativita

- vlastnosti zero ``plus`` a $\gg=$:

$\text{zero} \gg= f = \text{zero}$ -- zero ako identita zľava
 $p \gg= (\lambda x \rightarrow \text{zero}) = \text{zero}$ -- zero ako identita sprava
 $(p \text{ `plus` } q) \gg= f = (p \gg= f) \text{ `plus` } (q \gg= f)$ -- distribut.



State monad v prelude.hs

```
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 vrátí 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)
```



Preorder so stavom

```
import 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;  
    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"]
```



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)
```

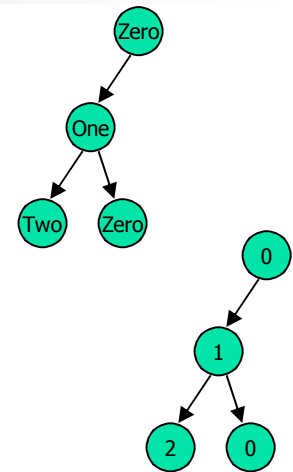
where

```
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
```