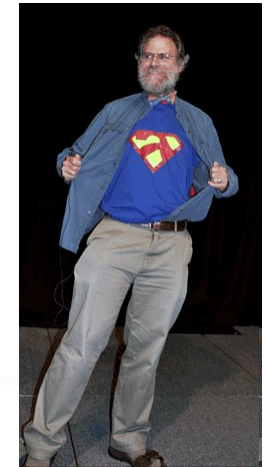


# Monády



Monady sú použiteľný nástroj pre programátora poskytujúci:

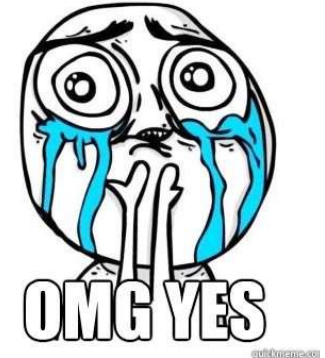
- modularitu – skladať zložitejšie výpočty z jednoduchších (no side-effects),
- flexibilitu – výsledný kód je ľahšie adaptovateľný na zmeny,
- izoluje side-effect operácie (napr. IO) od čisto funkcionálneho zvyšku.

Štruktúra prednášok:

- Monády - prvý dotyk
  - Functor
  - Applicative
  - Monády – princípy a zákony
- Najbežnejšie monády
  - Maybe/Error monad
  - List monad
  - IO monad
  - State monad
  - Reader/Writer monad
  - Continuation monad
- Transformátory monád
- Monády v praxi

# Monáda

(class Monad)



monáda **m** je typ implementujúci dve funkcie:

**class** Applicative m => Monad **m** where

return :: a -> m a

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

-- interface predpisuje tieto funkcie

-- to bude pure z Applicatives

-- náš `bind`

ktoré splňajú isté (sémantické) zákony:

**“neutrálnosť” return vzhľadom na >>=**

■  $\text{return } c \gg= (\lambda x \rightarrow g) = g[x/c]$

■  $m \gg= \lambda x \rightarrow \text{return } x = m$

**“asociativita” >>=**

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

Pravidlá sú nutnou  
podmienkou pre  
do-notáciu

inak zapísané:

$\text{return } c \gg= f = f \ c$  -- ľavo neutrálny prvok

$m \gg= \text{return} = m$  -- pravo neutrálny prvok

$(m \gg= f) \gg= g = m \gg= (\lambda x \rightarrow f \ x \gg= g)$   
-- asociativita >>=

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

# Základný interpreter výrazov

Princíp fungovania monád sme trochu ilustrovali na type

```
data 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 si vysvetlíme najprv na sérii evaluátorov aritmetických výrazov,  
presnejšie zredukovaných len na konštrukcie pozostávajúce z Con a Div:

+-\* je triviálne a len by to odvádzať pozornosť ...

```
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))
42
eval (Div (Div (Con 1972) (Con 0)) (Con 23))
*** Exception: divide by zero
```

monad.hs

Haskell má definované podobné typy  
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 M<sub>1</sub> a = Raise String | Return a** deriving(Show, Read, Eq)

evalExc :: Term -> **M<sub>1</sub> 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)

takto nejako vyzeral náš  
prvý kód, keď sme objavili  
Maybe, isJust, fromJust

```
> 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 operácií div (má stav **type 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)   = \state ->(a,state)
```

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

```
evalCnt (Div t u) state = let (a,state1) = evalCnt t state in  
                           let (b,state2) = evalCnt u state1 in  
                           (a `div` b, state2+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)  
> evalCnt (Div (Div (Con 1972) (Con 2)) (Div (Con 6) (Con 2))) 0  (328,3)
```



# Interpreter s výstupom

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

**type  $M_3$  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)  
  
> putStr$fst$evalOut (Div (Div (Con 1972) (Con 2)) (Con 23))
```

evalOut                :: Term ->  **$M_3$  Int**

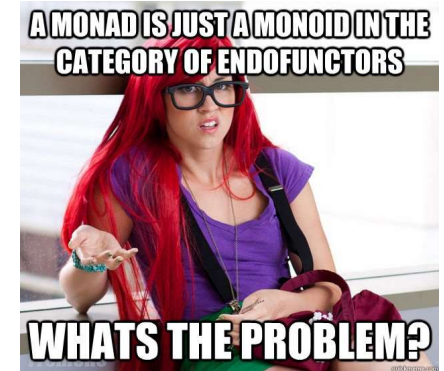
evalOut (Con a)    = let out\_a = line (Con a) a in (out\_a, a)

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 interpretra (Identity/Exception/State/Output)
- cieľom je napísať **jednu**, skoro uniformú verziu, z ktorej všetky existujúce vypadnú ako inštancia, s malými modifikáciami ...
- potrebujeme pochopiť typ/triedu/interface/design pattern monáda

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

- a potrebujeme pochopiť, čo je inštancia triedy (implementácia interface):

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

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

*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 = ... , >>= ...

```
eval                :: Term -> Mi Int
eval (Con a)        = return a
eval (Div t u)       = eval t >>= \valT ->
                      eval u >>= \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)
                        }
```

a čítame:  
hodnotu z výpočtu t priradíme do valT  
hodnotu z výpočtu u priradíme do valU  
výsledok `div` vrátime ako hodnotu



# Identity monad

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

```
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)
```

**Cvičenie:** dokážte, že platia vlastnosti:

```
return c >>= f      = f c           -- ľavo neutrálny prvok
m >>= return        = m             -- pravo neutrálny prvok
(m >>= f) >>= g      = m >>= (\x-> f x >>= g)
```

```
> evalIdentM (Div (Div (Con 1972) (Con 2)) (Con 23))
42
```

# 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**  $M_1 = \text{Exception } a = \text{Raise String} \mid \text{Return } a$  deriving (Show, Read, Eq)

**instance** Monad Exception where

return v = Return v

p >>= f = case p of

    Raise e -> Raise e

    Return a -> Return (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"
```

**Cvičenie:** dokážte, že platia 3 vlastnosti ...

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
                             SM g = f a
                             in g st1)
```

typovacia pomôcka:

$p :: \text{State} \rightarrow (a, \text{State})$

$f :: a \rightarrow \text{SM}(\text{State} \rightarrow (a, \text{State}))$

$g :: \text{State} \rightarrow (a, \text{State})$

```
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))
```

-- SM bez výsledku

-- zmení stav na +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 monáda SM Int, t.j. niečo ako **SM(State->(Int,State))**.

Preto si definujeme pomôcku, podobne ako (parse) pri monadických parseroch:

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

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

```
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
                             SM g = f a
                             in g st1)
```

typovacia pomôcka:

$p :: \text{State} \rightarrow (a, \text{State})$

$f :: a \rightarrow \text{SM}(\text{State} \rightarrow (a, \text{State}))$

$g :: \text{State} \rightarrow (a, \text{State})$

**Cvičenie:** dokážte, že platia vlastnosti:

$\text{return } c \gg= f$	$=$	$f \ c$	-- ľavo neutrálny prvok
$m \gg= \text{return}$	$=$	$m$	-- pravo neutrálny prvok
$(m \gg= f) \gg= g$	$=$	$m \gg= (\lambda x \rightarrow f \ x \gg= g)$	

Pravidlá sú nutnou  
podmienkou pre  
do-notáciu

```
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
                  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)) <=42",42)

let Out(s,_) = evalOutM (Div (Div (Con 1972) (Con 2)) (Con 23))
in putStr s
```

# 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) }

**Cvičenie:** Definujte sequence pomocou foldr

-- ak nezáleží na výsledkoch

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

sequence\_ = foldr (>>) (return ())

sequence\_ [m<sub>1</sub>,m<sub>2</sub>,...m<sub>n</sub>] = m<sub>1</sub> >>= \\_ ->  
m<sub>2</sub> >>= \\_ ->  
...  
m<sub>n</sub> >>= \\_ ->  
return ()

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



# Kde nájsť v *praxi* monádu ?

Prvý pokus

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

```
]
```

```
Return [42,2,3] :: Exception [Int]
```

**:: Exception Int**

**:: 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"**

```
> sequence [Just 1, Nothing, Just 4]
```

```
Nothing
```

```
> sequence [[1], [2,3]]
```

```
[[1,2],[1,3]]
```





# Kde nájsť v *praxi* monádu ?

Ďalší prvý pokus

```
> sequence [[1..3], [1..4], [7..9]]
[[1,1,7],[1,1,8],[1,1,9],[1,2,7],[1,2,8],[1,2,9],[1,3,7],[1,3,8],[1,3,9],[1,4,7],[1,4,8],[1,4,9],[2,1,7],
 [2,1,8],[2,1,9],[2,2,7],[2,2,8],[2,2,9],[2,3,7],[2,3,8],[2,3,9],[2,4,7],[2,4,8],[2,4,9],[3,1,7],[3,1,8],
 [3,1,9],[3,2,7],[3,2,8],[3,2,9],[3,3,7],[3,3,8],[3,3,9],[3,4,7],[3,4,8],[3,4,9]]
Kartézsky súčin...
```

Takže [] je monáda, tzv. List-Monad, ale čo sú funkcie **return** a **>>=**

**Cvičenie:** napíšte sequence pomocou foldr

**instance** Monad [] where

return x	= [x]	:: a -> [a]
m >>= f	= concat (map f m)	:: [a] -> (a -> [b]) -> [b]
m >>= f	= concatMap f m	:: [a] -> (a -> [b]) -> [b]
		concatMap :: (a -> [b]) -> [a] -> [b]

Podobný bind (>>=) ste videli v parseroch, tiež to bola analógia List-Monad

**Cvičenie:** dokážte, že platia 3 vlastnosti ...



# IO monáda

Druhý pokus :-)

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

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

-- hack

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

```
echo :: IO ()
echo = getChar >>= putChar
```

-- IO Char >>= (Char -> IO ())

```
do { c<-getChar; putChar c }
-- do { ch <-getChar; putStr [ch,ch] }
```

-- do { c<-getChar; putChar c } :: IO ()



# Interaktívny Haskell

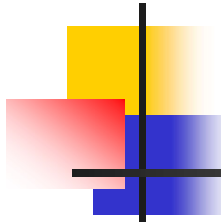
---

```
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"
[(),(),()]
```



# IO Monáda

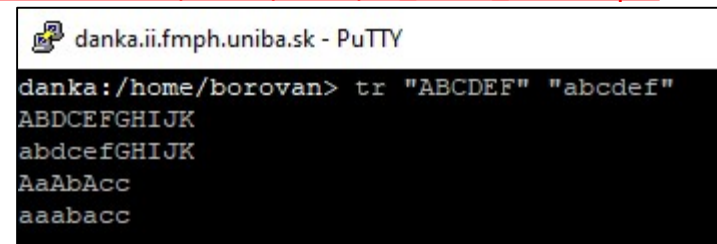
---

- Haskell je funkcionálne čistý *ad absurdum*, tzv. referenčne transparentný
- nejde napísať funkciu, ktorá by pre rovnaké argumenty vracala rôzne výsledky
- preto nejde napísať `getChar :: Char`, `nextRandomInt :: Int`
- IO monáda je logický hack, resp. side-effect
- kým hodnoty iných monád predstavujú nejaký výpočet, IO monáda predstavuje IO akciu (napr. čítanie, zápis, ...)
- ak váš kód niekde použije IO ..., všetky nadradené funkcie budú IO ...
- dôsledok: ak v kóde robíte nejakú IO akciu, váš main bude typu IO ...
- IO monáda rozdeľuje váš kód na funkcionálne **čistý**, a **akčný**, závislý na IO

# Homomorfizmus písmen

-- zjednodusený kód z All About Monads [https://www.cs.rit.edu/~swm/cs561/All\\_About\\_Monads.pdf](https://www.cs.rit.edu/~swm/cs561/All_About_Monads.pdf)

```
import Control.Monad
import System.Environment
import System.IO
```



danka.ii.fmph.uniba.sk - PuTTY

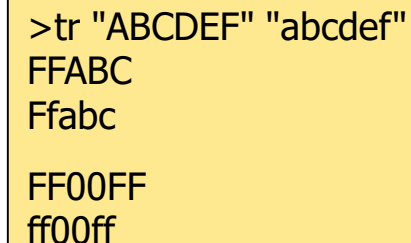
```
danka:/home/borovan> tr "ABCDEF" "abcdef"
ABDCEFGHIJK
abdcefGHIJK
AaAbAcc
aaabacc
```

`tr :: String -> String -> Char -> Char` -- translate char in set1 to char in set2

```
tr [] _ c = c
tr (x:xs) [] c = if x == c then ' ' else tr xs [] c
tr (x:xs) [y] c = if x == c then y else tr xs [y] c
tr (x:xs) (y:ys) c = if x == c then y else tr xs ys c
```

-- translates stdin to stdout based on command line arguments

```
main :: IO ()
main = do [set1,set2] <- getArgs
          contents <- hGetContents stdin
          putStr $ map (tr set1 set2) contents
```



```
>tr "ABCDEF" "abcdef"
FFABC
FfabC

FF00FF
ff00ff
```

tr.hs

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

# 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

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

**Cvičenie:** dokážte, že platia vlastnosti:



# Maybe MonadPlus

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

```
class Monad m => MonadPlus m where  
  mzero    :: m a  
  mplus    :: m a -> m a -> m a
```

-- podtrieda, resp. podinterface  
--  $\emptyset$   
-- disjunkcia

```
instance MonadPlus Maybe where  
  mzero      = Nothing  
  Just x `mplus` y  = Just x  
  Nothing `mplus` y = y
```

-- fail...  
-- or

```
> Just "a" `mplus` Just "b"  
Just "a"  
> Just "a" `mplus` Nothing  
Just "a"  
> Nothing `mplus` Just "b"  
Just "b"
```

kde použiť Maybe monad  
ak váš kód vyzerá takto:  
case ... of  
 Nothing -> Nothing  
 Just x -> case ... of  
 Nothing -> Nothing  
 Just y -> ...

a mal by vyzeráť takto:  
do x <- ...  
 y <- ...



# Zákony monád a monádPlus

- vlastnosti **return** a **>>=**:

<code>return x &gt;&gt;= f</code>	<code>= f x</code>	-- return ako identita zľava
<code>p &gt;&gt;= return</code>	<code>= p</code>	-- return ako identita sprava
<code>p &gt;&gt;= (\x -&gt; (f x &gt;&gt;= g))</code>	<code>= (p &gt;&gt;= (\x -&gt; f x)) &gt;&gt;= g</code>	-- "asociativita"

- vlastnosti **zero** a **`plus`**:

<code>zero `plus` p</code>	<code>= p</code>	-- zero ako identita zľava
<code>p `plus` zero</code>	<code>= p</code>	-- zero ako identita sprava
<code>p `plus` (q `plus` r)</code>	<code>= (p `plus` q) `plus` r</code>	-- asociativita

- vlastnosti **zero**, **`plus`** a **>>=**:

<code>zero &gt;&gt;= f</code>	<code>= zero</code>	-- zero ako identita zľava
<code>p &gt;&gt;= (\x-&gt;zero)</code>	<code>= zero</code>	-- zero ako identita sprava
<code>(p `plus` q) &gt;&gt;= f</code>	<code>= (p &gt;&gt;= f) `plus` (q &gt;&gt;= f)</code>	-- distribut.





# Sheep family

klasicky

```
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 Maybe 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
```

```
    Nothing
```

```
  else
```

```
    father (fromJust (mother o))
```

-- klasicky:

-- o :: Sheep

-- mother o :: Maybe Sheep

-- fromJust ... :: Sheep

```
mothersPaternalGrandfather :: Sheep -> Maybe Sheep
```

```
mothersPaternalGrandfather s = case (mother s) of
```

```
    Nothing -> Nothing
```

```
    Just m ->
```

```
        case (father m) of
```

```
            Nothing -> Nothing
```

```
            Just gf -> father gf
```

```
a mal by vyzerať takto:
do x <- ...
  y <- ...
```



# Sheep family

monadicky

```
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 maybe father

```
data Sheep = Sheep {name :: String, mother :: Maybe Sheep, father :: Maybe Sheep}
                    deriving (Eq)
```

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

-- monadicky:

-- m :: Sheep

matky otca otec

maternalGrandfather dolly = Just "Roger"

```
mothersPaternalGrandfather :: Sheep -> Maybe Sheep
```

```
mothersPaternalGrandfather s = do { m <- mother s ;
```

```
    gf <- father m ;
```

-- m, gf :: Sheep

```
    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]

m >>= f      = concat . map f m

**:: a -> [a]**

**:: [a] -> (a -> [b]) -> [b]**

```
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  $M\ a = \text{List } a = [a]$

$\text{return } x = [x] \quad :: a \rightarrow [a]$

$m \gg= f = \text{concatMap } f\ m \quad :: [a] \rightarrow (a \rightarrow [b]) \rightarrow [b]$

$\text{concatMap} = \text{concat} . \text{map } f\ m$

**Cvičenie:** overme platnosť zákonov:

- $\text{return } c \gg= (\backslash x \rightarrow g) = g[x/c]$ 
  - $[c] \gg= (\backslash x \rightarrow g) = \text{concatMap } (\backslash x \rightarrow g)\ [c] = \text{concat} . \text{map } (\backslash x \rightarrow g)\ [c] = \text{concat } [g[x/c]] = g[x/c]$
- $m \gg= \backslash x \rightarrow \text{return } x = m$ 
  - $[c_1, \dots, c_n] \gg= (\backslash x \rightarrow \text{return } x) = \text{concatMap } (\backslash x \rightarrow \text{return } x)\ [c_1, \dots, c_n] = \text{concat} . \text{map } (\backslash x \rightarrow \text{return } x)\ [c_1, \dots, c_n] = \text{concat } [[c_1], \dots, [c_n]] = [c_1, \dots, c_n]$
- $m1 \gg= (\backslash x \rightarrow m2 \gg= (\backslash y \rightarrow m3)) = (m1 \gg= (\backslash x \rightarrow m2)) \gg= (\backslash y \rightarrow m3)$ 
  - $([c_1, \dots, c_n] \gg= (\backslash x \rightarrow [d_1, \dots, d_m])) \gg= (\backslash y \rightarrow m3) = (\text{concat } [d_1[x/c_1], \dots, d_m[x/c_1]], \dots, [d_1[x/c_n], \dots, d_m[x/c_n]]) \gg= (\backslash y \rightarrow m3) = ([d_1[x/c_1], \dots, d_m[x/c_1], \dots, d_1[x/c_n], \dots, d_m[x/c_n]]) \gg= (\backslash y \rightarrow m3) = ([d_1[x/c_1], \dots, d_m[x/c_1], \dots, d_1[x/c_n], \dots, d_m[x/c_n]]) \gg= (\backslash y \rightarrow [e_1, \dots, e_k]) = \dots [e_i[y/d_j[x/c_i]]]$



# Zákony monádPlus 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 ++

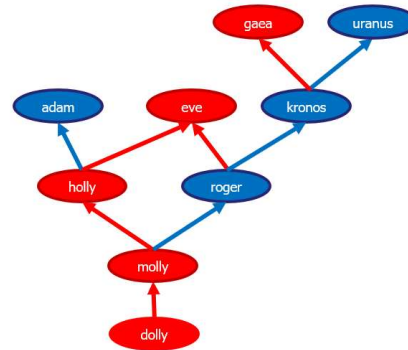
- vlastnosti zero `plus` a >>= :

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



# Sheep family

kto sú rodičia Dolly ?



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) ]) ++  
            maybeToList $ 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]) }
```

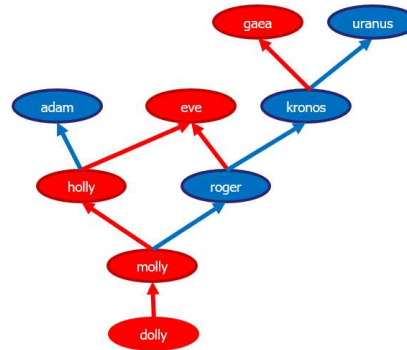
```
parents'" dolly = Nothing
```

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



# Sheep family

## Kto sú rodičia Dolly



```
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
```

```
parents" :: Sheep -> Maybe [Sheep]
```

```
parents" x = do { o<-father x; return [o] } `mplus` (do m<-mother x; return [m])
```

```
parents" dolly = Just ["Molly"]
```

```
parents" x = do {return $ maybeToList(father x)}
```

```
    `mplus`
```

```
    do {return $ maybeToList(mother x)}
```

```
parents" dolly = Just []
```

```
(Just[]) `mplus` (Just [1]) = Just []
```

```
parents" x = return $ maybeToList(father x) `mplus` return $ maybeToList(mother x)
```

```
parents" dolly = []
```

```
parents" x = (Just $ maybeToList(father x)) `mplus` (Just $ maybeToList(mother x))
```

```
parents" dolly = Just []
```

```
parents" x = return $ maybeToList(father x) `mplus` maybeToList(mother x)
```

```
parents" dolly = Just ["Molly"]
```

```
[] `mplus` [1]
```

```
parents" :: Sheep -> Maybe [Sheep]
```

```
parents" x = return $ maybeToMonad(father x) `mplus` maybeToMonad(mother x)
```

```
maybeToMonad :: (MonadPlus m) => Maybe a -> m a
```

```
maybeToMonad Nothing = mzero -- convert a Maybe value into another monad
```

```
maybeToMonad (Just s) = return s
```



# Sheep family

cvičenie

```
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
```

1) Definujte predkov po ženskej línii, teda  
k\_mother 1 je mama, k\_mother 2 je babka,  
k\_mother 3 je prababka, ...

**k\_mother :: Int -> Sheep -> Maybe Sheep**

**k\_mother 0 dolly = Just "Dolly"**

**k\_mother 1 dolly = Just "Molly"**

**k\_mother 2 dolly = Just "Holly"**

**k\_mother 3 dolly = Just "Eve"**

**k\_mother 4 dolly = Nothing**

-----  
2) Definujte všetkých predkov k-tej úrovne,  
opakovať v zozname sa môžu:

**k\_predecessors :: Int -> Sheep -> [Sheep]**

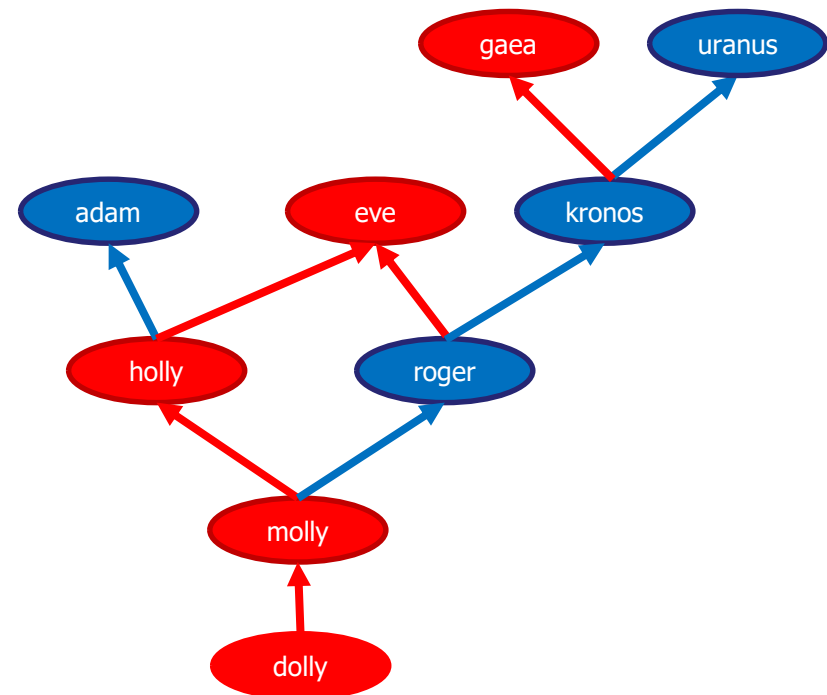
**k\_predecessors 1 dolly = ["Molly"]**

**k\_predecessors 2 dolly = ["Roger", "Holly"]**

**k\_predecessors 3 dolly = ["Kronos", "Eve", "Adam", "Eve"]**

**k\_predecessors 4 dolly = ["Uranus", "Gaea"]**

**k\_predecessors 5 dolly = []**





# 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)
```



# 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 ()
```

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

```
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)

Kartézsky súčin

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

```
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)]
```

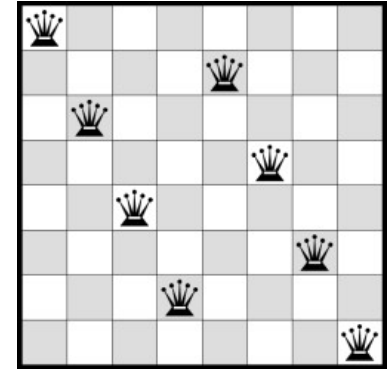
```
monadComprehension xs ys = do { x<-xs; y<-ys; return (x,y) }
```

```
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)]
```

# 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]
```

```
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])
```



# Control.Monad

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

mapM f as = sequence (map f as)
forM = flip mapM

zipWithM :: (Monad m) => (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM f xs ys = sequence (zipWith f xs ys)

replicateM :: (Monad m) => Int -> m a -> m [a]
replicateM n x = sequence (replicate n x)

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

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

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

mapM_ f as = sequence_ (map f as)
forM_ = flip mapM_

zipWithM_ :: (Monad m) => (a -> b -> m c) -> [a] -> [b] -> m ()
zipWithM_ f xs ys = sequence_ (zipWith f xs ys)

replicateM_ :: (Monad m) => Int -> m a -> m ()
replicateM_ n x = sequence_ (replicate n x)

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



# 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 záměna 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



# 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)`





# 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 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]
???
```

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



# 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 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)
```



# Error monad

---

```
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)
```

```
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
```

```
[(),()]
```



# 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"]
```



# State Stack

---

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

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

```
type Stack = [Int]
```

```
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) []
[]
```



# 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)
```

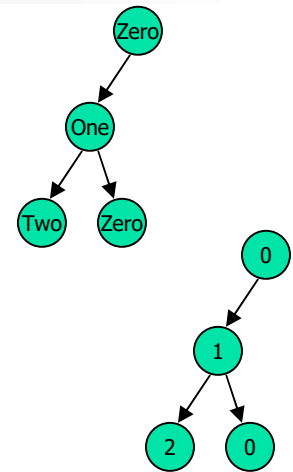
where

```
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
```