Monády

Monady sú použiteľný nástroj pre programátora poskytujúci:

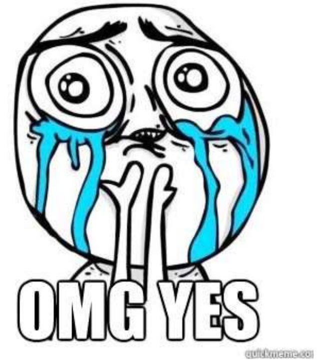
- modularitu – skladať zložitejšie výpočty z jednoduchších (no side-effects),
- flexibilitu – výsledný kód je ľahšie adaptovateľný na zmeny,
- izoluje side-effect operácie (napr. IO) od čisto funkcionálneho zvyšku.

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        -- interface predpisuje tieto funkcie
   return   :: a -> m a                      -- to bude pure z Applicatives
   >>=      :: m a -> (a -> m b) -> m b      -- náš `bind`
```

ktoré spĺňajú isté (sémantické) zákony:

Pravidlá sú nutnou podmienkou pre do-notáciu

"neutrálnost'" return vzhľadom na >>=

- return c >>= (\x->g) = g[x/c]
- m >>= \x->return x = m

"asociativita" >>=

- m1 >>= (\x->m2 >>= (\y->m3)) = (m1 >>= (\x->m2)) >>= (\y->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)
                                    -- asocitativita >>=
```

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 odvádzalo 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

Zdroj: http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf

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 interpretra:

```
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)
```

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

```
> evalExc (Div (Div (Con 1972) (Con 2)) (Con 23))
Retrun 42
> evalExc (Div (Con 1) (Con 0))
Raise "div by zero"
```

monad.hs

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 M2 a              = State -> (a, State)
type State             = Int

evalCnt                :: Term -> M2 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)
```

monad.hs

Interpreter s výstupom

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

```
> 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))
```

type M_3 a = (Output, a)

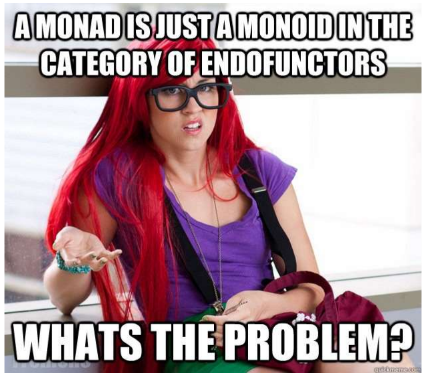
type Output = String

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"
```

monad.hs

Monadický interpreter
(vízia)



- 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 M_i where

```
    return = …
    >>= …
```

Cieľ: ukážeme, ako v monádach s typmi **M0**, **M1**, **M2**, **M3** dostaneme požadovaný intepreter 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 M0, M1, M2, M3 dostaneme požadovaný intepreter ako inštanciu všeobecného monadického interpretra:
instance Monad M_i where return = … , >>= …

```
eval                :: Term -> M_i 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

monad.hs

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

Pre identity monad:

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

Identity monad

na verziu **M_0 a = a** sme zabudli, volá sa **Identity monad**, resp. M_0 = 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))
42
```

Cvičenie1: 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)
```

monad.hs

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

```
> evalExceptM (Div (Div (Con 1972) (Con 2)) (Con 23))
Return 42
> evalExceptM (Div (Div (Con 1972) (Con 2)) (Con 0))
Raise "div by zero"
```

Cvičenie2: dokážte, že platia 3 vlastnosti ...

```
instance Monad  Exception where
  return v    = Return v
  p >>= f    =  case p of
                  Raise e   -> Raise e
                  Return a -> Return (f a)

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)
```

monad.hs

```
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))                  typovacia pomôcka:
  (SM p) >>= f    = SM (\st ->let (a,st1) = p st         p::State->(a,State)
                                  SM g = f a             f::a->SM(State->(a,State))
                              in g st1)                  g::State->(a,State)

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

incState          :: SM ()                               -- SM bez výsledku
incState          = SM (\s -> ((),s+1))                  -- zmení stav na +1
```

monad.hs

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 definujme pomôcku, podobne ako (parse) pri monadickýchparseroch:

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

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

monad.hs

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

Cvičenie3: 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)
```

Pravidlá sú nutnou podmienkou pre do-notáciu

monad.hs

```
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
```

monad.hs

Monadic Prelude

```
class Monad m where                                -- definition:(>>=), return
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b                   -- zahodíme výsledok prvej monády
    p >> q  = p >>= \ _ -> q

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

Cvičenie4: Definujte sequence pomocou foldr

```
-- ak nezáleží na výsledkoch
sequence_      :: (Monad m) => [m a] -> m ()
sequence_      = foldr (>>) (return ())
```

sequence_ $[m_1,m_2,\ldots m_n]$ = m_1 >>= _ ->
m_2 >>= _ ->
…
m_n >>= _ ->
return ()

do { m_1 ;
m_2 ;
…
m_n ;
return ()

monad.hs

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

monad.hs

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 **>>=**

Cvičenie4: 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

Cvičenie5: dokážte, že platia 3 vlastnosti ...

monad.hs

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

Cvičenie5: 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
  - $[c_1, \dots ,c_n]$ >>= (\x->return x) = concatMap (\x->return x) $[c_1, \dots ,c_n]$ = concat . map (\x->return x) $[c_1, \dots ,c_n]$ = concat $[[c_1], \dots ,[c_n]]$ = $[c_1, \dots ,c_n]$
- m1 >>= (****x->m2 >>= (****y->m3)) **=** (m1 >>= (****x->m2)) >>= (****y->m3)
  - ($[c_1, \dots ,c_n]$ >>= (\x->$[d_1, \dots ,d_m]$)) >>= (\y->m3) =
    (concat $[[d_1[x/c_1], \dots ,d_m[x/ c_1]], \dots [d_1[x/ c_n], \dots ,d_m[x/ c_n]]]$) >>= (\y->m3) =
    ($[d_1[x/c_1], \dots ,d_m[x/ c_1], \dots, d_1[x/ c_n], \dots ,d_m[x/ c_n]]$) >>= (\y->m3) =
    ($[d_1[x/c_1], \dots ,d_m[x/ c_1], \dots, d_1[x/ c_n], \dots ,d_m[x/ c_n]]$) >>= (\y->$[e_1, \dots ,e_k]$) = …
    $[e_i[y/d_j[x/c_l]]]$

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 { c<-getChar; putChar c } :: IO ()
-- do { ch <-getChar; putStr [ch,ch] }
```

monad.hs

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

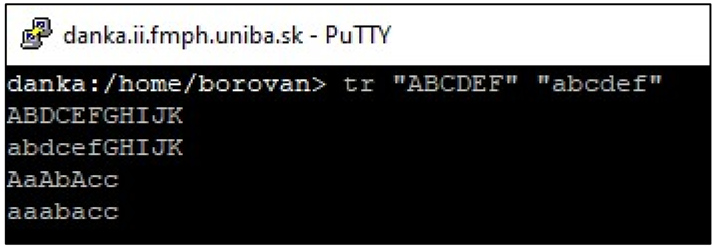
monad.hs

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

```
-- zjednoduseny kod z All About Monads https://www.cs.rit.edu/~swm/cs561/All_About_Monads.pdf
import Control.Monad
import System.Environment
import System.IO

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

IO Monad
(vstup čísla)

```
type Kopa = Int

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

valid :: Kopa -> Int -> Bool                -- korektný ťah
valid kopa beriem = (kopa >= beriem) && beriem < 4 && beriem > 0

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

nim1.hs

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))
                beriem <- getDigit "kolko beries : "
                if valid kopa beriem then
                   play2 (kopa - beriem) (not $ hrac)
                else
                   do putStrLn "zly tah"
                        play2 kopa hrac

nim2 :: IO ()
nim2 = play2 (nextInt 10 20) True                   -- generujeme náhodnú kopu 10..19
```

nim1.hs

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 beriem <- getDigit "kolko beries : "
          if valid kopa beriem then
            let kopa' = kopa - beriem 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)
```

nim1.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
```

Cvičenie6 (podobné ako Exception monad): dokážte, že platia vlastnosti: monad.hs

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"
```

kde použiť Maybe monad ak váš kód vyzerá takto:

```
case ... of
    Nothing -> Nothing
    Just x -> case ... of
        Nothing -> Nothing
        Just y -> ...
```

a mal by vyzerať takto:

```
do x <- ...
   y <- ...
```

monad.hs

```
mzero                = Nothing
Just x `mplus` y     = Just x
Nothing `mplus` y = y
```

Zákony monád a monádPlus

- 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.
```

Cvičenie 7: dokážte vlastnosti MonadPlus pre Maybe

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                 -- klasicky:
                              Nothing                                 -- o :: Sheep
                         else                                 -- mother o :: Maybe Sheep
                              father (fromJust (mother o))    -- 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 <- …

https://www.cs.rit.edu/~swm/cs561/All_About_Monads.pdf

Sheep.hs

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 ;                  -- monadicky:
                             father m }                       -- 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"

https://www.cs.rit.edu/~swm/cs561/All_About_Monads.pdf

Sheep.hs

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     = concat . map f m            :: [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
```

Zákony monádPlus pre List

Cvičenie 8: Dokážte vlastnoti MonadPlus pre List Monad

- vlastnosti zero a `plus`:

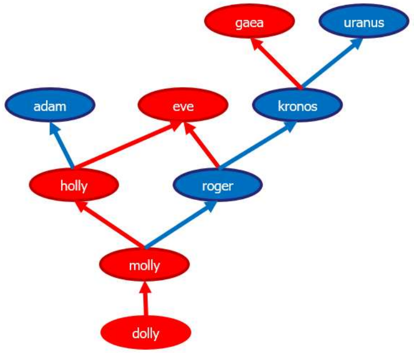
```
zero `plus` p           = p                      -- [] ++ p = p
p `plus` zero           = p                      -- p ++ [] = p
p `plus` (q `plus` r)   = (p `plus` q) `plus` r  -- asociativita ++
```

- 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"]

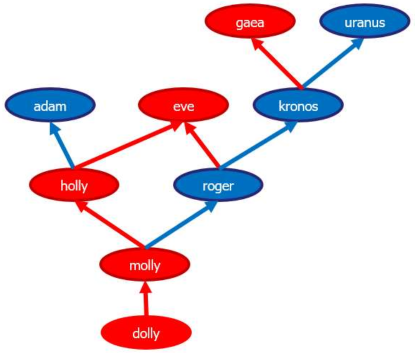
parents''' :: Sheep -> Maybe [Sheep]
parents''' x = do { o<-father x; m<-mother x; return ([o] `**mplus**` [m]) }
parents''' dolly = Nothing
parents''' roger = Just ["Kronos","Eve"]

https://www.cs.rit.edu/~swm/cs561/All_About_Monads.pdf

Sheep.hs

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.hs

https://www.cs.rit.edu/~swm/cs561/All_About_Monads.pdf

```
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
```

Sheep family

Rozcvička

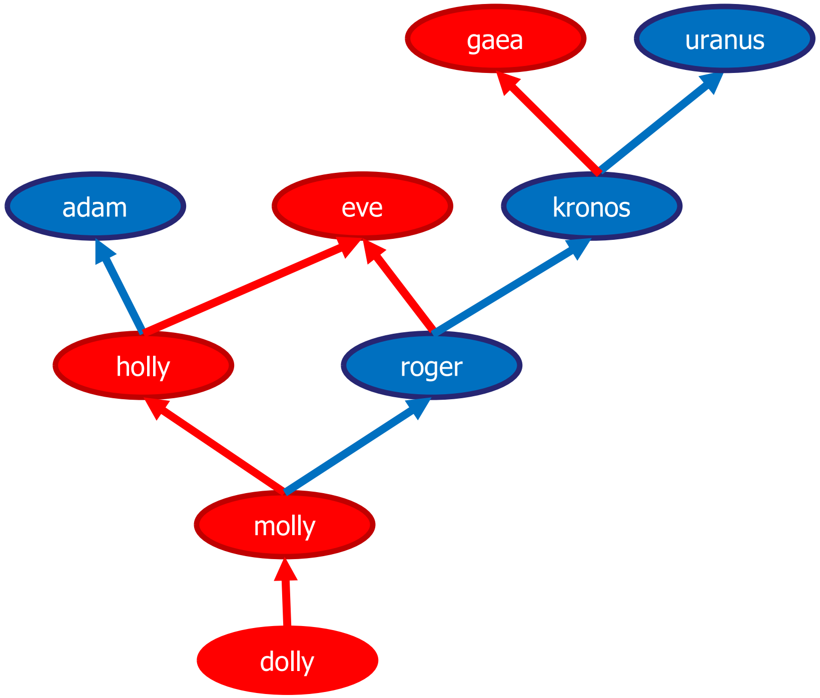
1) Definujte predkov po ženskej línii, teda k_mother 1 je mama, k_mother 2 je babka, k_mother 3 je prababka, ...

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_predecesors :: Int -> Sheep -> [Sheep]
k_predecesors 1 dolly = ["Molly"]
k_predecesors 2 dolly = ["Roger","Holly"]
k_predecesors 3 dolly = ["Kronos","Eve","Adam","Eve"]
k_predecesors 4 dolly = ["Uranus","Gaea"]
k_predecesors 5 dolly = []



https://www.cs.rit.edu/~swm/cs561/All_About_Monads.pdf

Sheep.hs

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)
```

listMonad.hs

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)
```

listMonad.hs

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

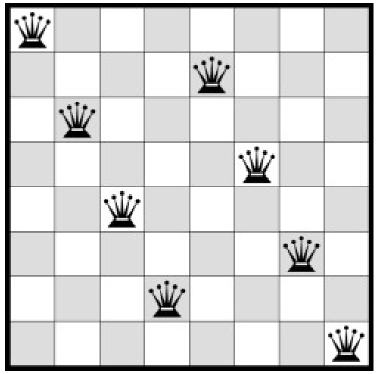
```
listComprehension xs ys = [(x,y) | x<-xs, y<-ys ]
guardedListComprehension xs ys =
                    [(x,y) | x<-xs, y<-ys, x<=y, x*y == 24 ]
```

```
> 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)]
```

listMonad.hs



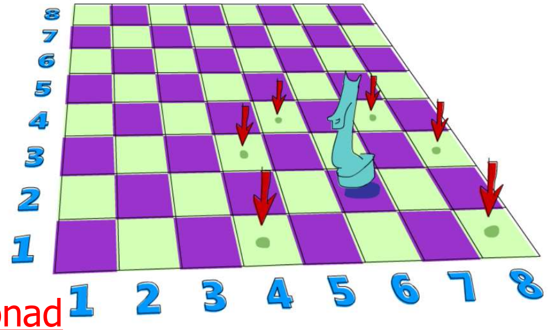
Backtracking

```
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])
```

queens.hs

Kôň

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




```
type KnightPos = (Int,Int)
-- jeden krok koňa na šachovnici
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]) ;  -- stále na ploche
                       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 (1,1) = 45016
length $ nub $ ink 7 (1,1) = 32
```

knights.hs

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 ()
```

https://hackage.haskell.org/package/base-4.6.0.1/docs/src/Control-Monad.html

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
```

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 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]
???
```

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

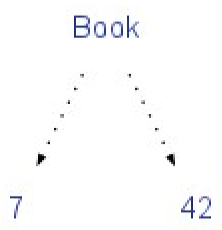
type State s a = s -> (a,s)

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

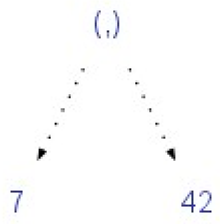
State s a je reprezentovaná krabicou State s pointrom na s -> (a,s)

Príklad:

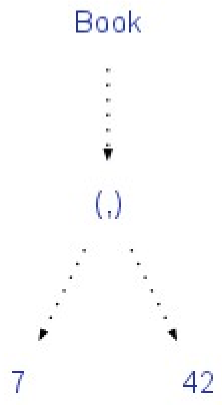
data Book = Book Int Int



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
(basics-1)

```
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

(basics-2)

```
return :: a -> State s a          -- monáda s výsledkom x::a, stavom s
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 monády x, výsledok je nezaujímavý
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
(basics-3)

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

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

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

evalState (gets (+1)) 77  = 78        -- vráti výsledok state monády pre
                                         vstupný stav s, po aplikovaný funkcie

execState (gets (+1)) 77  = 77        -- vráti výsledný stav state monády pre
                                         vstupný stav s, a ten sa nezmenil

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

State Stack

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

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

výsledok

stav

```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 vráti výslednú hodnotu

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

execState vráti výsledný stav

```
> 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) []
[]
```

súbor:stack.hs

Preorder so stavom
(Control.Monad.State)

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

stav

```
preorder :: Tree a -> State [a] ()                -- stav a výstupná hodnota
preorder Nil                          = return ()
preorder (Node value left right)      =
                                      do {             str :: [a]
                                           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) []  -- stav
["b","a","c"]

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

súbor:tree.hs

stav
výsledok

Prečíslovanie binárneho stromu

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

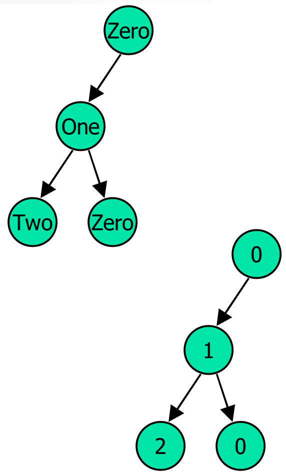
```
> e'
Node "d" (Node "c" (Node "a" Nil Nil) (Node "b" Nil Nil)) (Node "c" (Node "a" Nil
    Nil) (Node "b" Nil Nil))

> evalState (reindex 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 (reindex e') 0
7
```

súbor:tree.hs

stav výsledok

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 (addNode x table)
                                         put newTable
                                         return newPos

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



súbor:tree.hs

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
```

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"]
```

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

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

State Stack

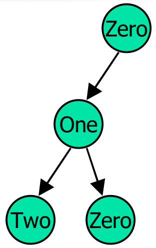
```haskell
type Stack = [Int]

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) []
[]
```

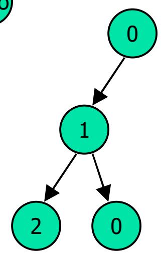
Prečíslovanie binárneho stromu

```
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
```