Monády – úvod

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

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

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

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

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

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

Monady sú použiteľný nástroj pre programátora poskytujúci:

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

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

Základný interpreter výrazov

Princíp fungovania monád sme trochu ilustrovali na type

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

```
return          :: a->Parser a
return v        = \xs -> [(v,xs)]
bind, >>=       :: Parser a -> (a -> Parser b) -> Parser b
p >>= qf        = \xs -> concat [ (qf v) xs' | (v,xs')<-p xs])
```

… len sme nepovedali, že je to monáda

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

```
data Term = Con Int | Div Term Term | Add … | Sub … | Mult …
                              deriving(Show, Read, Eq)
eval            :: Term -> Int
eval(Con a)     = a
eval(Div t u)   = eval t `div` eval u
```

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

```
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 M_1 a = Raise String | Return a   deriving(Show, Read, Eq)

evalExc :: Term -> M_1 Int

```
evalExc(Con a)    = Return a
evalExc(Div t u)  = case evalExc t of
                       Raise e -> Raise e
                       Return a ->
                           case evalExc u of
                               Raise e -> Raise e
                               Return b ->
                                   if b == 0
                                   then Raise "div by zero"
                                   else Return (a `div` b)
```

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

Interpreter so stavom

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

naivne:

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

resp.:

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

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

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

evalCnt              :: Term -> M2 Int
evalCnt (Con a) st   = (a,st)
evalCnt (Div t u) st =  let (a,st1) = evalCnt t st in
                        let (b,st2) = evalCnt u st1 in
                        (a `div` b, st2+1)
```

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

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

Interpreter s výstupom

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

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

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

```
> putStr(fst(evalOut (Div (Div (Con 1972) (Con 2)) (Con 23)
```

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

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

Monadický interpreter
(vízia)

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

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

- a potrebujeme pochopiť, čo je inštancia triedy

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

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

Functor – definícia

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

```
class Functor t where                  -- musí mať funkciu fmap s profilom
  fmap :: (a -> b) -> t a -> t b       -- haskell class je podobne java interface
```

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

- fmap id = id -- identita
- fmap (p . q) = (fmap p) . (fmap q) -- kompozícia

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

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

Functor – príklad

Cvičenie: Skúste definovať inštanciu triedy Functor pre typy:

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

```
> fmap (\s -> even s) (Cons 1 (Cons 2 Null))                 -- f : Int->Bool
Cons False (Cons True Null)
> fmap (\s -> s+s) (Cons 1 (Cons 2 Null))                    -- f : Int->Int
Cons 2 (Cons 4 Null)
> fmap (\s -> show s) (Cons 1 (Cons 2 Null))                  -- f : Int->String
Cons "1" (Cons "2" Null)

> fmap ((\t -> t++t) . (\s -> show s)) (Cons 1 (Cons 2 Null))     -- f : (String->String).(Int->String)
Cons "11" (Cons "22" Null)
> fmap (\t -> t++t)    (fmap (\s -> show s) (Cons 1 (Cons 2 Null))) -- overenie vlastnosti kompozície
Cons "11" (Cons "22" Null)

> fmap id (Cons 1 (Cons 2 Null))                              --  overenie vlastnosti identity
Cons 1 (Cons 2 Null)
```

Functor – strom

Cvičenie: Binárny strom:

```
data LExp a = Var a | Appl (LExp a) (LExp a) | Abs a (LExp a) deriving (Show)
instance Functor LExp where
    fmap f (Var x)                 = Var (f x)
    fmap f (Appl left right)       = Appl (fmap f left) (fmap f right)
    fmap f (Abs x right)           = Abs (f x) (fmap f right)

omega = Abs "x" (Appl (Var "x") (Var "x"))
> fmap (\t -> t++t) omega
Abs "xx" (Appl (Var "xx") (Var "xx"))
```

Cvičenie:

Ľubovoľne n-árny strom (prezývaný RoseTree alias Rhododendron):

```
data RoseTree a = Node a [RoseTree a]
instance Functor RoseTree where
    fmap f (Node a bs)      = Node (f a) (map (fmap f) bs)
```

Monáda
(class Monad)

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

```
class Monad m where                   -- predpisuje tieto funckie
   return   :: a -> m a
   >>=      :: m a -> (a -> m b) -> m b          -- náš `bind`
```

ktoré spĺňajú isté (sémantické) zákony:

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

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) }
```

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

Identity monad

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

na verziu M_0 a = a sme zabudli, volá sa identity monad, resp. M_0 = 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
```

```
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
```

Exception monad

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

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

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

```
evalExceptM              :: Term -> Exception Int
evalExceptM(Con a)      = return a
evalExceptM(Div t u)    = evalExceptM t >>= \valT->
                          evalExceptM u >>= \valU ->
                          if valU == 0 then Raise "div by zero"
                                       else return(valT `div` valU)
evalExceptM (Div t u) = do valT<-evalExceptM t
                           valU<-evalExceptM u
                           if valU == 0 then Raise "div by zero"
                                        else return(valT `div` valU)
```

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

State monad

```
data M₂ = SM a =  SM(State -> (a, State)) -- funkcia obalená v konštruktore SM
                                          -- type State = Int
instance Monad SM where
  return v        = SM (\st -> (v, st))                pomôcka:
  (SM p) >>= f    = SM (\st ->let (a,st1) = p st in    p::State->(a,State)
                              let SM g = f a in        f::a->SM(State->(a,State))
                              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 ()
incState          = SM (\s -> ((),s+1))
```

do notácia

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

Problémom je, že výsledkom evalSM, resp. evalSM’, nie je stav, ale stavová monada SM Int, t.j. niečo ako SM(State->(Int,State)).

Preto si definujme pomôcku, podobne ako pri parseroch:

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

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

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

Output monad

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

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

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

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

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

```
> evalOutM (Div (Div (Con 1972) (Con 2)) (Co
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)) <
```

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

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

do { m_1 ;
m_2 ;
…
m_n ;
return ()

Kde nájsť v *praxi* monádu ?

Prvý pokus :-)

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

> sequence [evalExceptM (Div (Div (Con 1972) (Con 2)) (Con 23)),
                 evalExceptM (Div (Con 8) (Con 4)),
                 evalExceptM (Div (Con 7) (Con 0))
            ]
???                                                            == Raise "div by 0"
```

IO monáda

Druhý pokus :-)

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

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

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

echo = getChar >>= putChar

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

Interaktívny Haskell

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

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

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

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

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

Maybe MonadPlus

```
data Maybe a = Nothing | Just a

class Monad m => MonadPlus m where – podtrieda, resp. podinterface
    mzero     :: m a                              -- ø
    mplus     :: m a -> m a -> m a                -- disjunkcia

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

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

Zákony monád a 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.
```

List monad

- List monad použijeme, ak simulujeme nedeterministický výpočet

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

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

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

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

List monad

```
type List a          = [a]

instance Functor List where
     fmap = map

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

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

List monad - vlastnosti

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

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

Cvičenie: overme platnosť zákonov:

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

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
```

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)
```

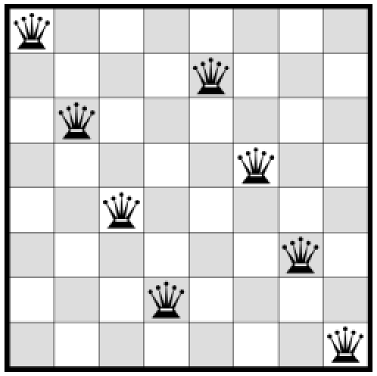
Guard
(Control.Monad)

```
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)]
```

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])
```

filterM
(Control.Monad)

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

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

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

mapM, forM
(Control.Monad)

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

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

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

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

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

foldM
(Control.Monad)

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

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

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

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

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

Error monad

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

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

```haskell
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')
```

Writer monad
(Control.Monad.Writer)

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

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)
```

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) []
[]
```

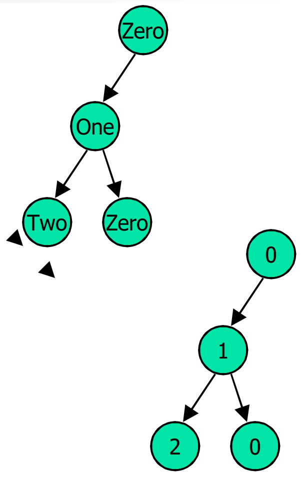
Prečíslovanie binárneho stromu

```
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
```