Monády – úvod

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

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

- Noel Winstanley: What the hell are Monads?, 1999
  http://www-users.mat.uni.torun.pl/~fly/materialy/fp/haskell-doc/Monads.html
- Jeff Newbern's: All About Monads https://www.cs.rit.edu/~swm/cs561/All_About_Monads.pdf
- A Gentle Introduction to Haskell,
  https://www.haskell.org/tutorial/monads.html
  https://wiki.haskell.org/All_About_Monads
- Sujit Kamthe: Understanding Functor and Monad With a Bag of Peanuts
  https://medium.com/beingprofessional/understanding-functor-and-monad-with-a-bag-of-peanuts-8fa702b3f69e
- Functors, Applicatives, And Monads In Pictures
- http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

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

```
    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 odvádzalo pozeornosť

```
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
```

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

```
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)
```

```
> 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 **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)
```

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

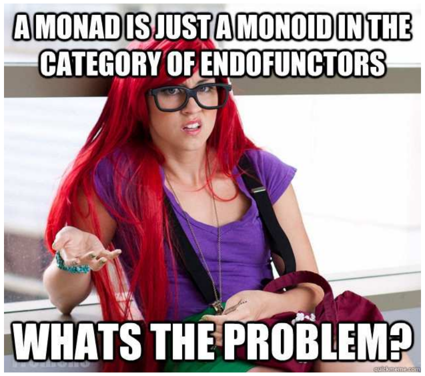
evalOut :: Term -> **M_3 Int**

```
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"
```

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/des.pattern nazývaný monáda

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

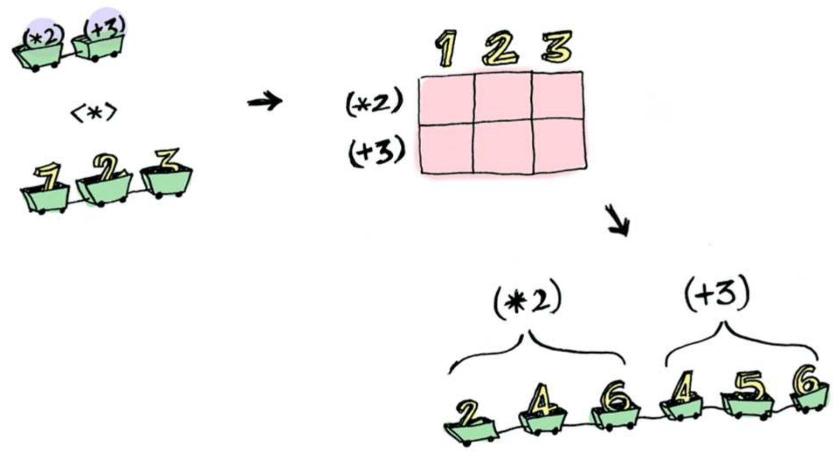
- a potrebujeme pochopiť, čo je inštancia triedy (implementácia interface):

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

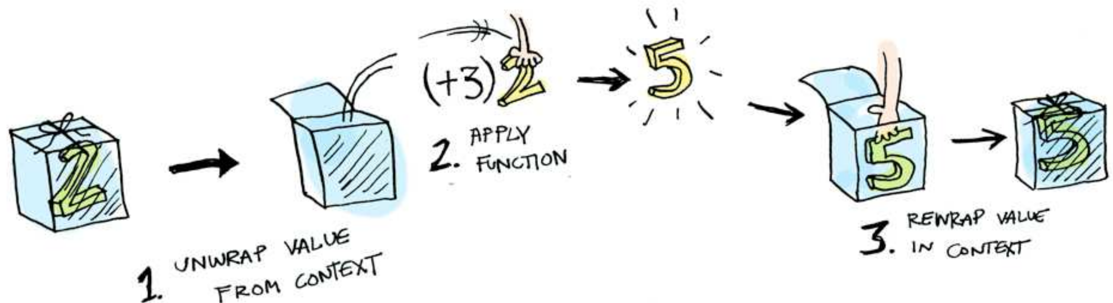
Roadmap

- Haskell má triedy, ale sú to vlastne interface (Java)
- Haskell má podtriedy, čo je vlastne dedenie na interface (Java)
- dedenie na interface ste určite v Jave videli, napr. na kolekciách
- Functor
- Applicatives
- Monad
- MonadPlus



Alternatívny prístup:
Functors, Applicatives, And Monads In Pictures
http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

Functor



http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

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

```
                                   -- každý typ t ak je Funtor t
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 <u>dve pravidlá</u> (to je <u>sémantika</u>, 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)
```

Cvičenie

class Functor t where
  fmap :: (a -> b) -> t a -> t b

Def.
fmap **f** (Raise str) = Raise str
fmap **f** (Return x) = Return (**f** x)

Dokázať:
fmap id = id
fmap (p . q) = (fmap p) . (fmap q)

- fmap id =? id
  - fmap id (Raise str) = Raise str
  - fmap id (Return x) = Return x
- fmap (p.q) =? (fmap p) . (fmap q)
  - L.S. = fmap (p.q) (Raise str) = Raise str
  - P.S. = ((fmap p) . (fmap q)) (Raise str) = (fmap p) ((fmap q) (Raise str))
    = Raise str
  - L.S. = fmap (p.q) (Return x) = Return ((p.q) x) = (Return (p (q x)))
  - P.S. = ((fmap p) . (fmap q)) (Return x)
    = (fmap p) ((fmap q) (Return x))
    = (fmap p) (Return (q x))
    = (Return (p (q x)))

```
class Functor t where
   fmap :: (a -> b) -> t a -> t b

fmap id        = id
fmap (p . q)  = (fmap p) . (fmap q)
```

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)
```

```
class Functor t where
   fmap :: (a -> b) -> t a -> t b
fmap id         = id
fmap (p . q)  = (fmap p) . (fmap q)
```

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"))
> fmap (\t -> (length t)) omega
Abs 1 (Appl (Var 1) (Var 1))
```

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 **m** je typ implementujúci dve funkcie:

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

ktoré spĺňajú isté (sémantické) zákony:

neutrálnosť return:

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

neutrálnosť 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 >>=

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 -> 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) }
```

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 = 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
```

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)
```

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)

```
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"
```

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))                typovacia 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 (parse) 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
```

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

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)
```

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

Output monad

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

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

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

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

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

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

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

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

do { m_1 ;
m_2 ;
...
m_n ;
return ()

Kde nájsť v *praxi* monádu ?

```
> 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"
```

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 **>>=**

```
instance Monad [] where
    return x     = [x]                      :: a -> [a]
    m >>= f      =  concat (map f m)        :: [a] -> (a -> [b]) -> [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 { c<-getChar; putChar c } :: IO ()
-- do { ch <-getChar; putStr [ch,ch] }
```

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
```

Cvičenie: dokážte, že platia vlastnosti:

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     = 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
```

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, … ,c_n] >>= (\x->return x) = concatMap (\x->return x) [c_1, … ,c_n] = concat . map (\x->return x) [c_1, … ,c_n] = concat [[c_1], … ,[c_n]] = [c_1, … ,c_n]
- m1 >>= (****x->m2 >>= (****y->m3)) **=** (m1 >>= (****x->m2)) >>= (****y->m3)
  - ([c_1, … ,c_n] >>= (\x->[d_1, … ,d_m])) >>= (\y->m3) =
    (concat [[d_1[x/c_1], … ,d_m[x/ c_1]], … [d_1[x/ c_n], … ,d_m[x/ c_n]]]) >>= (\y->m3) =
    ([d_1[x/c_1], … ,d_m[x/ c_1], …, d_1[x/ c_n], … ,d_m[x/ c_n]]) >>= (\y->m3) =
    ([d_1[x/c_1], … ,d_m[x/ c_1], …, d_1[x/ c_n], … ,d_m[x/ c_n]]) >>= (\y->[e_1, … ,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
```