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

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

Monads, Arrows, and Idioms

Philip Wadler, https://homepages.inf.ed.ac.uk/wadler/topics/monads.html

Články Phila Wadlera na stránke

- Monads for functional programming
- The essence of functional programming
- Comprehending monads
- The arrow calculus
- Monadic constraint programming
- Idioms are oblivious, arrows are meticulous, monads are promiscuous
- The marriage of effects and monads
- How to declare an imperative
- Imperative functional programming

What the hell are Monads?

Noel Winstanley,
https://www-users.mat.uni.torun.pl//~fly/materialy/fp/haskell-doc/Monads.html

Obsah:

- Maybe
- State
- The Monad Class
- Do notation
- Monadic IO
- Programming in the IO Monad

All About Monads

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

Obsah:

Part I - Understanding Monads

What is a monad? Meet the Monads. Doing it with class Monad support in Haskell

Part II - A Catalog of Standard Monads

Introduction. The Identity monad. The Maybe monad. The Error monad. The List monad. The IO monad. The State monad. The Reader monad. The Writer monad. The Continuation monad.

Part III - Monads in the Real

Combining monads the hard way. Monad transformers. Standard monad transformers. Anatomy of a monad transformer. More examples with monad transformers. Managing the transformer.

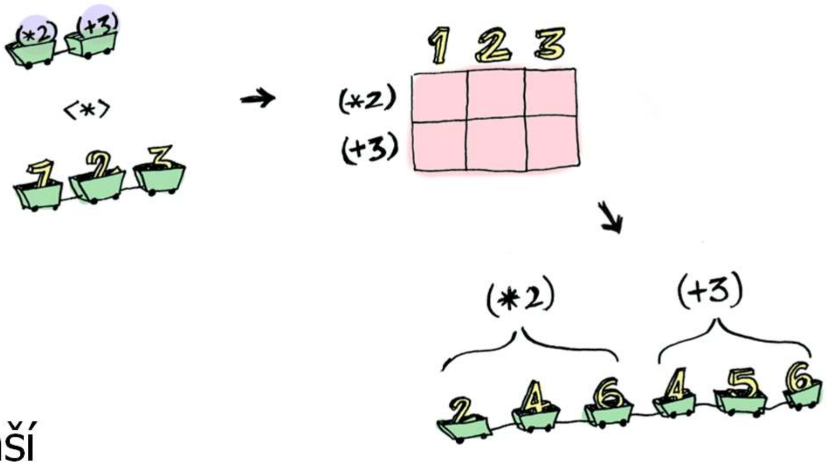
Roadmap

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

Relevantné triedy v Haskelli:

- Functor
- Applicatives
- Monad
- MonadPlus
- ...

Takže monáda nie je najjednoduchší typ v tejto hierarchii



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

Functor

prvotná idea

```
double :: [Int] -> [Int]
double [] = []
double (x:xs) = (x+x):double xs
```

```
sqr :: [Int] -> [Int]
sqr [] = []
sqr (x:xs) = x*x: sqr xs
```

```
map :: (a->b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

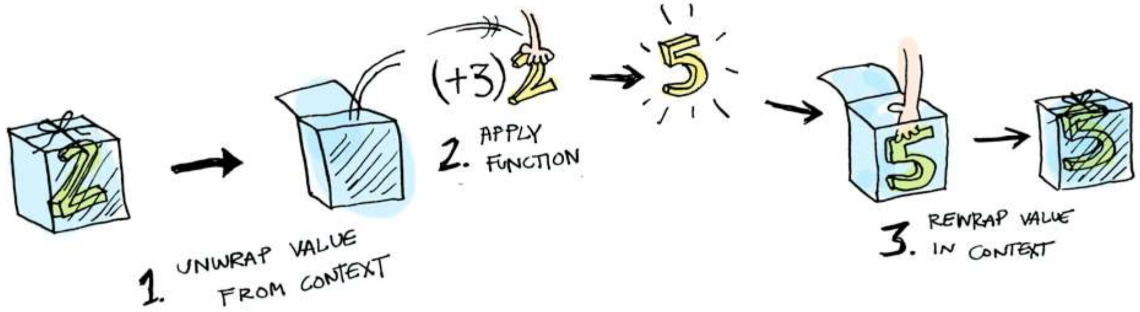
```
map (*2)
```

```
map (^2)
```

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

fmap aplikuje funkciu f na hodnoty zabalené do typu, ktorý implementuje Interface Functor

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 implementuje 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

Cvičenie1: Príklad inštancie pre **data M$_1$ a = Raise String | Return a**, overte, že platia obe sémantické pravidlá:

```
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

Definícia:
fmap **f** (Raise str) = Raise str
fmap **f** (Return x) = Return (**f** x)

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

Cvičenie1 (pokrač.):

- fmap id =? id
  - fmap id (Raise str) = Raise str
  - fmap id (Return x) = Return (id x) = Return x
- fmap (p.q) =? (fmap p) . (fmap q)
  - Prípad Raise error:
  - 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
  - Prípad Return hodnota:
  - 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)))…. q.e.d.

Functor

Maybe, List

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

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

Cvičenie2: Definujte 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]

a pochopíte, ako je Functor definovaný pre štandardné typy Maybe a [].

```
> fmap (even) (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 (show) (Cons 1 (Cons 2 Null))                -- f : Int->String
Cons "1" (Cons "2" Null)

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

Maybe, List

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

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

Cvičenie2 (pokrač.): Definujte 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 even [1,2,3]
[False,True,False]
> fmap (*2) [1,2,3]
[2,4,6]
> fmap (show) [1,2,3]
["1","2","3"]
> fmap (\x->x++x) $ fmap (show) [1,2,3]
["11","22","33"]
> fmap ((\x->x++x). show) [1,2,3]
["11","22","33"]
```

```
instance Functor MyMaybe  where
      fmap f MyNothing     =  MyNothing
      fmap f (MyJust x)    =  MyJust (f x)

instance  Functor MyList  where
      fmap f Null = Null
      fmap f (Cons x xs) = Cons (f x) (fmap f xs)

instance  Functor []  where
      fmap = map
```

... chýba dôkaz platnosti vlastností ... q.e.d

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

Functor – strom

Cvičenie3: Binárny strom:

data LExp a = ID a | APL (LExp a) (LExp a) | ABS a (LExp a) deriving (Show)

instance Functor LExp where

fmap **f** (ID x) = ID (**f** x)

fmap **f** (APP left right) = APP (fmap f left) (fmap f right)

fmap **f** (Abs x body) = ABS (**f** x) (fmap f body)

```
omega = ABS "x" (APP (ID "x") (ID "x"))
> fmap (\t -> t++t) omega
ABS "xx" (APP (ID "xx") (ID "xx"))
> fmap (\t -> (length t)) omega
ABS 1 (APP (ID 1) (ID 1))
```

Cvičenie4: Ľubovoľne n-árny strom (prezývaný RoseTree alias Rhododendron):

data RoseTree a = Node a [RoseTree a]

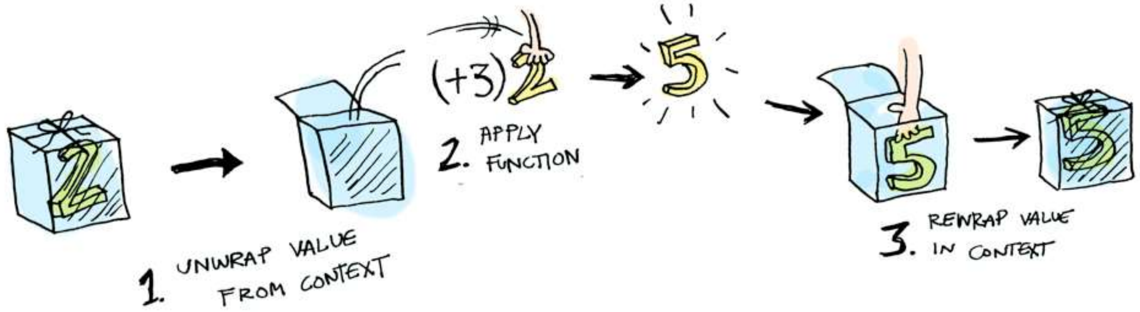
instance Functor RoseTree where

fmap f (Node a bs) = Node (f a) (map (fmap f) bs)

... chýba dôkaz platnosti vlastností pre Cvičenie 3 aj 4 ... q.e.d

Functor

zhrnutie



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

```
instance Functor [] where
  -- fmap :: (a->b) -> [a] -> [b]
   fmap = map
```

```
instance Functor Maybe where
   -- fmap :: (a->b) -> Maybe a -> Maybe b
   fmap _ Nothing = Nothing
   fmap g (Just x) = Just (g x)
```

```
instance Functor IO where
  -- fmap :: (a->b) -> IO a -> IO b
   fmap g mx = do { x<-mx; return (g x) }
```

```
infixl 4 <$>
<$>  = fmap
main :: IO ()
main = do  res <- words <$> getLine
           res <- fmap words getLine
           putStrLn $ show res
```

```
double :: Functor f => f Int -> f Int
double = fmap (*2)

sqr :: Functor f => f Int -> f Int
sqr = fmap (^2)
```

```
double (Just 7) = Just 14
double [1,2,3,4] = [2,4,6,8]

double (Branch (Leaf 7) (Leaf 9)) =
          Branch (Leaf 14) (Leaf 18)

double (Rose 3 [Rose 5 [], Rose 7 []]) =
          Rose 6 [Rose 10 [],Rose 14 []]
```

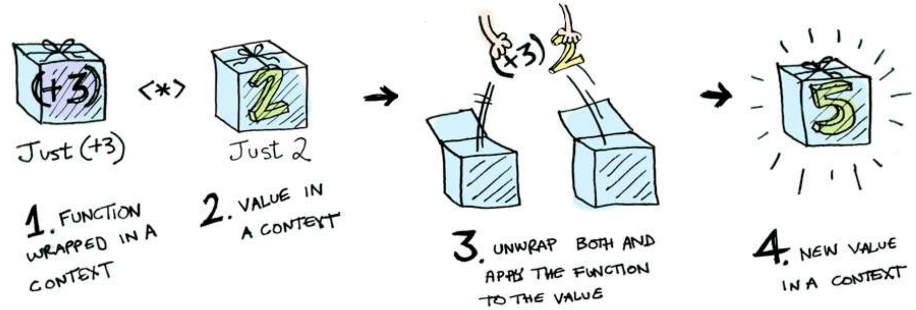
Applicative

prvotná idea

- Functor predstavuje abstrakciu aplikácie **unárnej funkcie** na každý prvok Functor-like dátovej štruktúry, nech je akákoľvek komplikovaná
- Čo, ak by sme mali funkcie s veľa argumentami (nie len unárne):
  - fmap0 :: a -> f a
  - fmap1 :: (a->b) -> f a -> f b -- fmap
  - fmap2 :: (a->b->c) -> f a -> f b -> f c
  - fmap3 :: (a->b->c->d) -> f a -> f b -> f c -> f d -- ☹
- riešenie = Currying – transformácia funkcie s mnohými argumentami na unárnu, ktorá vráti inú funkciu, ktorá skonzumuje ďalšie argumenty
  - **pure :: a -> f a**
  - **(<*>) :: f (a->b) -> f a -> f b** **infixl**

  Napr. nech g chce "tri argumenty"
  - pure g <*> x <*> y <*> z = ((pure g <*> x) <*> y) <*> z
- Hierarchia
  - pmap0 g0 = pure g0 -- g0 je konštanta
  - fmap1 g1 x = pure g1 <*> x -- g1 :: a->b, pure g1::f (a->b)
  - fmap2 g2 x y = pure g2 <*> x <*> y -- g2 :: a->b->c, pure g2::f (a->b->c)
  - fmap3 g3 x y z = pure g3 <*> x <*> y <*> z

Applicative



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

class Functor f => Applicative f where -- Applicative je podtrieda Functor

  pure :: a -> f a

  (<*>) :: f (a -> b) -> f a -> f b (infixl 4)

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

- pure id <*> v = v -- identita
- pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- kompozícia
- pure f <*> pure x = pure (f x) -- homomorfizmus
- u <*> pure y = pure ($ y) <*> u = pure (\g->g y) <*> u -- výmena

Príklad (pre M1):

Return id <*> Return 4 = Return 4

Return (.) <*> Return (+1) <*> Return (+2) <*> Return 4 = Return 7

Return (+1) <*> (Return (+2) <*> Return 4) = Return 7

Return (+4) <*> Return 3 = Return 7 -- pure f <*> pure x = pure (f x)

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

class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

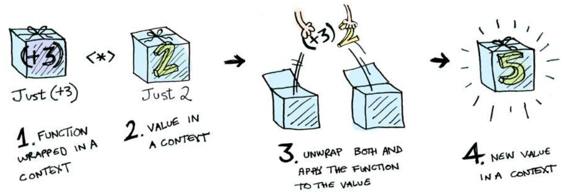
pure id <*> v = v -- identita
pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- kompozícia
pure f <*> pure x = pure (f x) -- homomorfizmus
u <*> pure y = pure ($ y) <*> u = pure (\g->g y) <*> u

Applicative

Cvičenie5: definujte inštanciu M1 pre triedu Applicative a overte 4 pravidlá:

instance Applicative M1 where
    pure a = Return a
    (Raise e) <*> _ = Raise e -- e:: String, Raise e::M1 a
    (Return f) <*> a = fmap f a -- f::a->b, Return f :: M1(a->b)

Príklad:



1) Return id <*> Return 4 = Return 4

2) L.S. = Return (.) <*> Return (+1) <*> Return (+2) <*> Return 4 = Return 7
   P.S. = Return (+1) <*> (Return (+2) <*> Return 4) = Return 7

3) Return (+4) <*> Return 3 = Return 7 -- pure f <*> pure x = pure (f x)

4) Return (+2) <*> Return 7 = Return 9 = Return ($ 7) <*> Return (+2)

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

Applicative

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

pure id <*> v = v                                 -- identita
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)      -- kompozícia
pure f <*> pure x = pure (f x)                    -- homomorfizmus
u <*> pure y = pure ($ y) <*> u = pure (\g->g y) <*> u
```

Cvičenie5 (pokrač.): definujte inštanciu M1 pre Applicative a overte pravidlá:

```
instance Applicative M1 where
    pure a              = Return a
    (Raise e)  <*> _    = Raise e        -- e:: String, Raise e::M1 a
    (Return f) <*> a    = fmap f a       -- f::a->b, Return f :: M1(a->b)
```

Dôkaz:

1) (Return id) <*> v = fmap id v = v   pravidlo identity pre Functors

2) (Return (.)) <*> (Return fu) <*> (Return fv) <*> (Return fw) =
   (Return (.) fu) <*> (Return fv) <*> (Return fw) =
   (Return ((.) fu) fv) <*> (Return fw) = (Return (fu . fv)) <*> (Return fw) =
   (Return ((fu . fv) fw)) = Return (fu (fv (fw)))

3) pure f <*> pure x = (Return f) <*> (Return x) = fmap f (Return x) = Return (f x) = pure (f x)

4) L.S. = (Return f) <*> (Return y) = fmap f (Return y) = (Return (f y))
   P.S. = (Return ($ y)) <*> (Return f) = fmap ($ y) (Return f) = Return (($ y) f) = Return (f y)

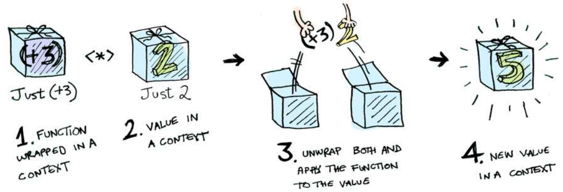
```
data M1 a    =   Raise String | Return a  deriving(Show, Read, Eq)
```

Applicative

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

pure id <*> v = v                              -- identita
pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- kompozícia
pure f <*> pure x = pure (f x)                 -- homomorfizmus
u <*> pure y = pure ($ y) <*> u = pure (\g->g y) <*> u
```

Cvičenie5": definujte inštanciu Maybe pre triedu Applicative a overte pravidlá:

```
instance Applicative Maybe where
    pure                :: a -> Maybe a
    pure                = Just
    (<*>)               :: Maybe (a->b) -> Maybe a -> Maybe b
    Nothing  <*> _      = Nothing
    (Just g) <*> a      = fmap g a
```



Príklad:

```
pure (*2) <*> Just 7                              = Just 14
pure (+) <*> Just 7 <*> Just 9                    = Just 16
pure (+) <*> Nothing <*> Just 9                   = Nothing
pure (\x y z->(x,y,z)) <*> Just 1 <*> Just 2 <*> Just 3   = Just (1,2,3)
```

```
data Maybe a    =    Just a | Nothing  deriving(Show, Read, Eq)
```

Applicative

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

pure id <*> v = v                              -- identita
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)   -- kompozícia
pure f <*> pure x = pure (f x)                 -- homomorfizmus
u <*> pure y = pure ($ y) <*> u = pure (\g->g y) <*> u
```

Cvičenie6: definujte inštanciu [] pre triedu Applicative a overte pravidlá:

```
instance Applicative [] where
      pure a          = [a]
      fs  <*> xs      = [ f x | f <- fs, x <- xs]
```

Príklad:

```
1) pure (+1) <*> [1..5]                              = [2,3,4,5,6]
2) pure (+) <*> [1,2] <*> [11,12]                    = [12,13,13,14]
3) pure (,) <*> [1,2] <*> [11,12]                    = [(1,11),(1,12),(2,11),(2,12)]
2) pure (.) <*> pure (+1) <*> pure (+3) <*> pure 9   = 13
   pure (.) <*> pure (+1) <*> pure (+3) <*> [9]      = [13]
   pure (.) <*> pure (+1) <*> pure (+3) <*> Just 9   = Just 13
3) pure (+1) <*> pure 7                              = 8
4) pure ($ 7) <*> pure (+1)                          = 8
```

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

Applicative

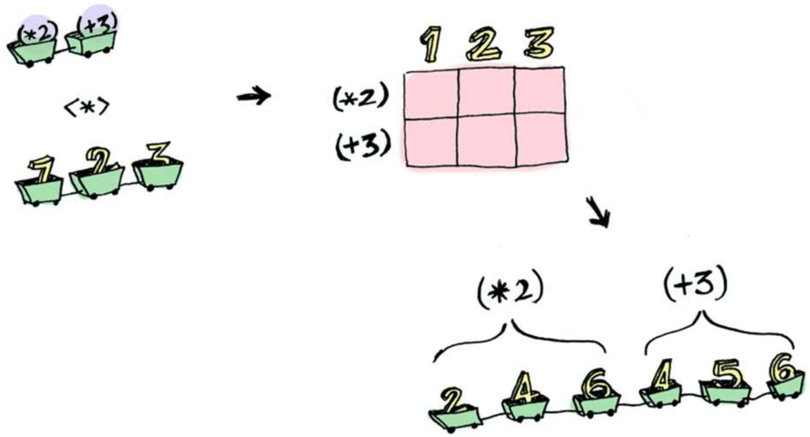
```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

pure id <*> v = v                              -- identita
pure (.) <*> u <*> v <*> w = u <*> (v <*> w) -- kompozícia
pure f <*> pure x = pure (f x)                 -- homomorfizmusu
<*> pure y = pure ($ y) <*> u = pure (\g->g y) <*> u
```

Cvičenie6: definujte inštanciu [] pre Applicative, a overte pravidlá:

```
instance Applicative [] where
    pure a          = [a]
    fs  <*> xs      = [ f x | f <- fs, x <- xs]
```

Dôkaz:

1) (Return id) <*> v = [id] <*> v = v
2)
3) pure f <*> pure x = [f] <*> [x] = [f x]
4) [f1,… fn] <*> [y] = [f1 y, … fn y]



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

Príklady:

[(*2), (+3)] <*> [1,2,3] = [2,4,6,4,5,6]

(,) <$> [1,2,3] <*> [4,5,6]
[(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)]

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

Kartézsky súčin
domáca úloha

```
module KSucin where
cart  :: [[t]] -> [[t]]
```

Príklad:

```
cart [ [1,2], [3,4], [5] ] = [[1,3,5],[1,4,5],[2,3,5],[2,4,5]]

cart [ [1,2], [3,4], [5,6,7] ] =          [[1,3,5],[1,3,6],[1,3,7],[1,4,5],[1,4,6],[1,4,7],[2,3,5],[2,3,6],
                                           [2,3,7],[2,4,5],[2,4,6],[2,4,7]]

cart [ [1,2], [3,4], [5,6] ] =            [[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],[2,4,6]]
cart [ [1,2], [3,4], [] ] = []
cart [ ["janka", "danka"], ["misko", "palko"] ] = [ ["janka","misko"],["janka","palko"],["danka","misko"],
                                                    ["danka","palko"]]
```

```
sequenceApplicative          :: Applicative f => [f a] -> f [a]
sequenceApplicative []       =  pure []
sequenceApplicative (x:xs)   =  pure (:) <*> x <*> sequenceApplicative xs

sequenceApplicative‘         :: Applicative f => [f a] -> f [a]
sequenceApplicative' xs      = foldr (\x -> \rec -> pure (:) <*> x <*> rec) (pure []) xs
```

GHC.Base

- https://hackage.haskell.org/package/base-4.15.0.0/docs/GHC-Base.html

```
▸ Applicative []                                      #
```
Since: base-2.1

```
▾ Applicative Maybe                                   #
```
Since: base-2.1

Defined in GHC.Base

Methods

```
pure :: a -> Maybe a

(<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b

liftA2 :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c

(*>) :: Maybe a -> Maybe b -> Maybe b

(<*) :: Maybe a -> Maybe b -> Maybe a
```

```
▸ Applicative IO                                      #
```
Since: base-2.1

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 >>=
```

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
```

monad.hs

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 M_1 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"
```

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

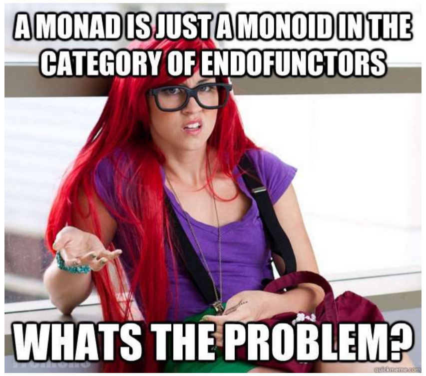
tretia verzia je interpreter výrazov, ktorý vypisuje debug.informáciu do reťazca

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

evalOut           :: Term -> M3 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"
```

```
> 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))
```

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/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

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) }
```

```
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
```

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)

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

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

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 (_,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,…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
  - $[c_1, \dots, c_n]$ >>= (\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
```