

Monády

IB016 Seminář z funkcionálního programování

Fakulta informatiky, Masarykova univerzita

Připomenutí: Funktory

Kontejnerový pohled

- f a
kontejner obsahující hodnoty typu a (např. **Maybe** a , **BinTree** a , $[a]$)
- $fmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$
funkce, která aplikuje zadanou funkci na každý prvek kontejneru, ale zachová strukturu
- např. pro $toUpper :: Char \rightarrow Char$
 - $fmap\ toUpper :: Maybe\ Char \rightarrow Maybe\ Char$, ale taky
 - $fmap\ toUpper :: BinTree\ Char \rightarrow BinTree\ Char$, ale taky
 - $fmap\ toUpper :: [Char] \rightarrow [Char]$,
 - obecně $fmap\ toUpper :: Functor\ f \Rightarrow f\ Char \rightarrow f\ Char$

Algebraický pohled

- $f :: * \rightarrow *$
typová funkce, která pro typ vrátí typ (**Maybe**, **BinTree**)
- $fmap$
jakákoliv funkce splňující axiomy třídy **Functor**

Monády

Co je to monáda?

- zastaralý výraz pro prvoka
- základní substance, ze které se skládá vesmír¹
- termín z teorie kategorií
- funktor, ke kterému přidáme ještě nějaké operace kromě `fmap`
- abstrakce výpočtů a jejich řetězení

¹[https://en.wikipedia.org/wiki/Monad_\(philosophy\)](https://en.wikipedia.org/wiki/Monad_(philosophy))

Co je to monáda?

- zastaralý výraz pro prvok
- základní substance, ze které se skládá vesmír¹
- termín z teorie kategorií
- funktor, ke kterému přidáme ještě nějaké operace kromě `fmap`
- abstrakce výpočtů a jejich řetězení

Navrhovaná strategie pochopení monád:

- dívat se na různé funktoxy „výpočtovým pohledem“
- zkoumat u každého **zvlášť**, co znamená „řetěžit výpočet“
- abstrakce vyplyne časem sama

¹[https://en.wikipedia.org/wiki/Monad_\(philosophy\)](https://en.wikipedia.org/wiki/Monad_(philosophy))

Výpočty a jejich výsledky

výpočet s výsledkem typu $a \approx$ může dělat jiné věci (selhat, měnit stav systému, ...), má výsledek typu a

$x :: \mathbf{Int}$

$x = 19$, ale i

$x = \text{succ } \$ \text{ succ } (2 * 4) * 2$

Výpočty a jejich výsledky

výpočet s výsledkem typu $a \approx$ může dělat jiné věci (selhat, měnit stav systému, ...), má výsledek typu a

```
x :: Int
```

```
x = 19, ale i
```

```
x = succ $ succ (2 * 4) * 2
```

```
y :: Maybe String
```

```
y = Just "adamat", ale i
```

```
y = Nothing, ale i
```

```
y = lookup 445763 (getStudents db)
```

(Výsledkem výpočtu je "adamat", nebo žádný neexistuje.)

Maybe jako výpočet

Maybe a je výpočet s možností selhání (parciální funkce):

- \rightsquigarrow^* **Just** 42
- \rightsquigarrow^* **Nothing**

výsledkem je 42

výsledek není, výpočet selhal

Maybe jako výpočet

Maybe a je výpočet s možností selhání (parciální funkce):

- \rightsquigarrow^* **Just** 42
- \rightsquigarrow^* **Nothing**

výsledkem je 42

výsledek není, výpočet selhal

Máme parciální výpočty

- $x, y :: \text{Maybe Int}$, které vrátí číslo nebo selžou,
- $f :: \text{Int} \rightarrow \text{Maybe Int}$, která z čísla vypočítá číslo nebo selže,
- $g :: \text{String} \rightarrow \text{Maybe Int}$, která z řetězce vypočítá číslo nebo selže.

Maybe jako výpočet

Maybe a je výpočet s možností selhání (parciální funkce):

- \rightsquigarrow^* **Just** 42
- \rightsquigarrow^* **Nothing**

výsledkem je 42

výsledek není, výpočet selhal

Máme parciální výpočty

- $x, y :: \text{Maybe Int}$, které vrátí číslo nebo selžou,
- $f :: \text{Int} \rightarrow \text{Maybe Int}$, která z čísla vypočítá číslo nebo selže,
- $g :: \text{String} \rightarrow \text{Maybe Int}$, která z řetězce vypočítá číslo nebo selže.

Jak vyrobit výpočty, které by intuitivně odpovídaly

- $x + y :: \text{Maybe Int}$
- $f\ x :: \text{Maybe Int}$
- $f . g :: \text{String} \rightarrow \text{Maybe Int}$
- $f\ (g\ "12") :: \text{Maybe Int}$

Ani jeden z těchto výrazů není typově korektní. ☹

Ideální abstrakce

- umožňuje kombinování výsledků více výpočtů do jednoho
- umožňuje skládání více výpočtů za sebe

Abstrakce nebude fungovat jen pro **Maybe** a, ale na libovolné výpočty.

Kombinace výpočtů v Maybe

```
liftM2 :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
```

Kombinace výpočtů pomocí (čisté²) funkce

- provedeme výpočet „operandů“ (podvýpočty),
- pokud některý selže, i kombinace selže (**Nothing**),
- jinak se výsledkem (v **Just**) kombinace podvýsledků.

Kombinací dostaneme zase výpočet (tj. něco v **Maybe**).

²ve smyslu „není sama monadickým výpočtem“³

³konkrétně zde „nebalí výsledek do Maybe“

Kombinace výpočtů v Maybe

```
liftM2 :: (a -> b -> c) -> Maybe a -> Maybe b -> Maybe c
```

Kombinace výpočtů pomocí (čisté²) funkce

- provedeme výpočet „operandů“ (podvýpočty),
- pokud některý selže, i kombinace selže (**Nothing**),
- jinak se výsledkem (v **Just**) kombinace podvýsledků.

Kombinací dostaneme zase výpočet (tj. něco v **Maybe**).

Příklad:

```
liftM2 (+) (Just 10) (Just 2) ~>* Just 12
```

```
liftM2 (+) (Nothing) (Just 2) ~>* Nothing
```

²ve smyslu „není sama monadickým výpočtem“³

³konkrétně zde „nebalí výsledek do Maybe“

$(\gg=) :: \text{Maybe } a \rightarrow (a \rightarrow \text{Maybe } b) \rightarrow \text{Maybe } b$

Řetězení výpočtů

- provedeme výpočet mezivýsledku,
- pokud se povedl (**Just** x), pokračujeme dál s hodnotou x
- pokud selhal (**Nothing**), nepokračujeme (\leadsto^* **Nothing**)

Řetězení výpočtů v Maybe

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

Řetězení výpočtů

- provedeme výpočet mezivýsledku,
- pokud se povedl (**Just** x), pokračujeme dál s hodnotou x
- pokud selhal (**Nothing**), nepokračujeme (\rightsquigarrow^* **Nothing**)

Příklad:

```
tryRead "3" >>= \x -> safeDiv 10 (x - 1)  $\rightsquigarrow^*$  Just 5
```

```
tryRead "a" >>= \x -> safeDiv 10 (x - 1)  $\rightsquigarrow^*$  Nothing
```

```
tryRead "1" >>= \x -> safeDiv 10 (x - 1)  $\rightsquigarrow^*$  Nothing
```


- Na totální funkci můžeme nahlížet jako na parciální.
- S neselhávajícím výpočtem můžeme pracovat, jako by selhání umožňoval.
- Pro hodnotu umíme vyrobit výpočet, který ji prostě vrací.

- Na totální funkci můžeme nahlížet jako na parciální.
- S neselhávajícím výpočtem můžeme pracovat, jako by selhání umožňoval.
- Pro hodnotu umíme vyrobit výpočet, který ji prostě vrací.
- `pure :: a -> Maybe a`
`pure x = Just x`

Maybe je monáda

- 1 Umíme řetězit výpočty:

$(\gg=) :: \text{Maybe } a \rightarrow (a \rightarrow \text{Maybe } b) \rightarrow \text{Maybe } b$

- 2 Umíme vyrobit triviální výpočet hodnoty:

$\text{pure} :: a \rightarrow \text{Maybe } a$

= podstata monády⁴

⁴obě funkce ještě musí splňovat nějaká pravidla, uvidíme později

Seznam = nedeterministické výpočty

[a] je výpočet s možností více výsledků (nedeterministická funkce):

- \rightsquigarrow^* [42]
- \rightsquigarrow^* [42, 12]
- \rightsquigarrow^* []

výsledkem je 42

výsledkem je 42 nebo 12

výsledek není žádný

Seznam = nedeterministické výpočty

[a] je výpočet s možností více výsledků (nedeterministická funkce):

- \rightsquigarrow^* [42]
- \rightsquigarrow^* [42, 12]
- \rightsquigarrow^* []

výsledkem je 42

výsledkem je 42 nebo 12

výsledek není žádný

Máme nedeterministické výpočty

- $x, y :: [\text{Int}]$, které vrátí několik možných čísel,
- $f :: \text{Int} \rightarrow [\text{Int}]$, která z čísla vypočítá několik možných čísel
- $g :: \text{String} \rightarrow [\text{Int}]$, která z řetězce vypočítá několik možných čísel

Seznam = nedeterministické výpočty

[a] je výpočet s možností více výsledků (nedeterministická funkce):

- \rightsquigarrow^* [42]
- \rightsquigarrow^* [42, 12]
- \rightsquigarrow^* []

výsledkem je 42

výsledkem je 42 nebo 12

výsledek není žádný

Máme nedeterministické výpočty

- $x, y :: [\text{Int}]$, které vrátí několik možných čísel,
- $f :: \text{Int} \rightarrow [\text{Int}]$, která z čísla vypočítá několik možných čísel
- $g :: \text{String} \rightarrow [\text{Int}]$, která z řetězce vypočítá několik možných čísel

Pomocí monády [a] jde vyrobit výpočty, které odpovídají

- $x + y :: [\text{Int}]$
- $f\ x :: [\text{Int}]$
- $f . g :: \text{String} \rightarrow [\text{Int}]$

Seznam = nedeterministické výpočty

Chceme používat funkce, které mohou vracet seznam více hodnot stejného typu. Například:

- chceme všechny komplexní druhé/třetí odmocniny čísla
- máme různé varianty n-té otázky v odpovědníku
- získáváme obsah adresáře
- volíme směr v bludišti

Seznam = nedeterministické výpočty

Chceme používat funkce, které mohou vracet seznam více hodnot stejného typu. Například:

- chceme všechny komplexní druhé/třetí odmocniny čísla
- máme různé varianty n-té otázky v odpovědníku
- získáváme obsah adresáře
- volíme směr v bludišti

S každým mezivýsledkem má smysl zkusit pokračovat. Řetěžením takových funkcí tak můžeme například:

- zjistit všechny komplexní šesté odmocniny čísla
- generovat různé varianty celého odpovědníku
- rekurzivně získat všechny soubory v domovském adresáři
- backtrackingem najít východ z bludiště

Seznam = nedeterministické výpočty

Příklad:

- máme k dispozici funkce

```
-- 2. odmocniny
```

```
root2 :: Complex Float -> [Complex Float]
```

```
-- 3. odmocniny
```

```
root3 :: Complex Float -> [Complex Float]
```

- chceme funkci `root6` vracející všechny 6. odmocniny

Seznam = nedeterministické výpočty

Příklad:

- máme k dispozici funkce

```
-- 2. odmocniny
```

```
root2 :: Complex Float -> [Complex Float]
```

```
-- 3. odmocniny
```

```
root3 :: Complex Float -> [Complex Float]
```

- chceme funkci `root6` vracející všechny 6. odmocniny

```
root6 :: Complex Float -> [Complex Float]
```

```
root6 c = concat $ map root3 (root2 c)
```

Seznam = nedeterministické výpočty

Jak vypadá řetězení a `pure` pro nedeterministické výpočty?

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

`pure` `:: a -> [a]`

Seznam = nedeterministické výpočty

Jak vypadá řetězení a `pure` pro nedeterministické výpočty?

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

```
list >>= f = concat $ map f list
```

```
pure :: a -> [a]
```

```
pure x = [x]
```

Pozorování: `[]` \approx **Nothing** (neexistuje žádný výsledek)

Either jako výpočet

Either a b je výpočet s možností více chybových stavů

- \rightsquigarrow^* **Right** 42
- \rightsquigarrow^* **Left** "out of memory"
- \rightsquigarrow^* **Left** "invalid input"

výsledkem je 42

výsledek není, výpočet selhal

výsledek není, výpočet selhal

Skládání

- při chybovém stavu výpočet dál nepokračuje
- vrátíme výsledek nebo první chybový stav, který nastal

- chceme pracovat s „náhodnými“ hodnotami
- „náhodné“ hodnoty závisí na `seed` → potřebujeme `seed` propagovat a aktualizovat napříč funkcemi
- nemáme k dispozici globální stavové proměnné
- nutné předávat jako argument v celém výpočtu

- chceme pracovat s „náhodnými“ hodnotami
- „náhodné“ hodnoty závisí na `seed` → potřebujeme `seed` propagovat a aktualizovat napříč funkcemi
- nemáme k dispozici globální stavové proměnné
- nutné předávat jako argument v celém výpočtu
- náhodné hodnoty můžeme reprezentovat jako funkce:
 - vstup je hodnota `seed`
 - výstup:
 - „náhodná“ hodnota závislá na `seed`
 - nová hodnota `seed` pro další „náhodnou“ proměnnou
 - **Typ:**

- chceme pracovat s „náhodnými“ hodnotami
- „náhodné“ hodnoty závisí na `seed` → potřebujeme `seed` propagovat a aktualizovat napříč funkcemi
- nemáme k dispozici globální stavové proměnné
- nutné předávat jako argument v celém výpočtu
- náhodné hodnoty můžeme reprezentovat jako funkce:
 - vstup je hodnota `seed`
 - výstup:
 - „náhodná“ hodnota závislá na `seed`
 - nová hodnota `seed'` pro další „náhodnou“ proměnnou
 - **Typ:** `Int` → (a, `Int`)

Pro přehlednost tento typ pojmenujme

```
newtype Rand a = Rand { runRand :: Int -> (a, Int) }
```

Opět chceme řetězit výpočty.

Příklad:

- `rollDie :: Rand Int` vrátí náhodné číslo z intervalu [1, 6]
- `roll2Dice :: Rand Int`

Výpočet s náhodou

Pro přehlednost tento typ pojmenujme

```
newtype Rand a = Rand { runRand :: Int -> (a, Int) }
```

Opět chceme řetězit výpočty.

Příklad:

- `rollDie :: Rand Int` vrátí náhodné číslo z intervalu [1, 6]
- `roll2Dice :: Rand Int`
`roll2Dice = Rand $ \seed ->`
 let
 (d1, seed') = runRand rollDie seed
 (d2, seed'') = runRand rollDie seed'
 in
 (d1 + d2, seed'')

```
newtype Rand a = Rand { runRand :: Int -> (a, Int) }
```

- Princip řetězení náhodných výpočtů zobecníme.
- Levá strana bere `seed` → vytváří nový `seed'` → použije se jako vstup pro náhodné číslo na pravé straně.

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

```
pure :: a -> Rand a
```

```
newtype Rand a = Rand { runRand :: Int -> (a, Int) }
```

- Princip řetězení náhodných výpočtů zobecníme.
- Levá strana bere `seed` → vytváří nový `seed'` → použije se jako vstup pro náhodné číslo na pravé straně.

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

```
x >>= f = Rand $ \seed ->
```

```
  let (val, seed') = runRand x seed
```

```
  in runRand (f val) seed'
```

```
pure :: a -> Rand a
```

```
pure x = Rand $ \seed -> (x, seed)
```

Výpočet s náhodou

Řešení roll2Dice:

```
roll2Dice :: Rand Int
```

```
roll2Dice =
```

```
  rollDie >>= (\d1 ->
```

```
    rollDie >>= (\d2 ->
```

```
      pure (d1 + d2)))
```

Výpočet s náhodou

Řešení roll2Dice:

```
roll2Dice :: Rand Int
roll2Dice =
  rollDie >>= (\d1 ->
    rollDie >>= (\d2 ->
      pure (d1 + d2)))
```

Nebo pomocí **do**-notace:

```
roll2Dice :: Rand Int
roll2Dice = do
  d1 <- rollDie
  d2 <- rollDie
  return (d1 + d2)
```

Výpočet s náhodou

Řešení `roll2Dice`:

```
roll2Dice :: Rand Int
roll2Dice =
  rollDie >>= (\d1 ->
    rollDie >>= (\d2 ->
      pure (d1 + d2)))
```

Nebo pomocí `do`-notace:

```
roll2Dice :: Rand Int
roll2Dice = do
  d1 <- rollDie
  d2 <- rollDie
  return (d1 + d2)
```

Nebo pomocí `liftM2`:

```
roll2Dice :: Rand Int
roll2Dice = liftM2 (+) rollDie rollDie
```

Rand jako monáda s vnitřním stavem

Obecněji: monády s vnitřním stavem.

- **Rand**
- **Turtle**
- **Parser**
- **State s**
- ★ **IO**

(Některé uvidíme později během semestru.)

stav je sémě generátoru

poloha a směr želvy

stav obsahuje nepřečtenou část textu

vlastní libovolný stav typu `s`

stav je kouzelný⁵

⁵Pro zájemce: https://wiki.haskell.org/IO_inside

Typová třída Monad

Typová třída Monad

```
class Applicative m => Monad m where  
  (>>=) :: m a -> (a -> m b) -> m b  -- čti „bind“  
  return :: a -> m a                  -- = pure
```

Typová třída Monad

```
class Applicative m => Monad m where  
  (>>=) :: m a -> (a -> m b) -> m b  -- čti „bind“  
  return :: a -> m a                  -- = pure
```

```
(>>) :: m a -> m b -> m b  
x >> y = x >>= \_ -> y
```

- umožňuje řetězení výpočtů
 - předem definujeme, jak se toto řetězení chová
 - `bind` navenek pracuje s výsledkem = „rozbalenou“ hodnotou
- definice třídy v `Prelude`, více v `Control.Monad`
- prozatím si nevímejme nadtřídy **Applicative...**

Instance třídy Monad I

```
instance Monad Maybe where  
  return :: a -> Maybe a
```

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

Instance třídy Monad I

```
instance Monad Maybe where
```

```
  return :: a -> Maybe a
```

```
  return = Just
```

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

```
  Nothing >>= f = Nothing
```

```
  Just x   >>= f = f x
```

- „Parciální“ funkce (výpočet může selhat).
- >>=: výsledek se předává, pokud existuje.

Instance třídy Monad II

```
instance Monad [] where  
  return :: a -> [a]
```

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

```
instance Monad [] where
```

```
  return :: a -> [a]
```

```
  return x = [x]
```

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

```
xs >>= f = concat (map f xs)
```

- „Nedeterministické“ funkce/výpočty
($a \rightarrow [b]$ = jeden vstup, libovolný počet výsledků).
- $\gg=$: výpočet probíhá nad každou hodnotou vstupu

Instance třídy Monad III

```
instance Monad (r -> _) where -- funkce z r  
  return :: a -> (r -> a)
```

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


Instance třídy Monad III

```
instance Monad (r -> _) where -- funkce z r
  return :: a -> (r -> a)
  return x _ = x

  (>>=) :: (r -> a) -> (a -> (r -> b)) -> (r -> b)
  (>>=) rx f ctx = let x = rx ctx
                   ry = f x
                   in ry ctx
```

- Výpočty s read-only kontextem (např. konfigurace).
- >>=: všem výpočtům předá též kontext
- Monáda čtenáře. Uvidíme ještě později v semestru.

Instance třídy Monad IV

```
instance Monad ([], _) where – dvojice se seznamem  
  return :: a -> ([], a)
```

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

```
instance Monad ([l], _) where – dvojice se seznamem  
  return :: a -> ([l], a)  
  return x = ([], x)  
  
  (>>=) :: ([l], a) -> (a -> ([l], b)) -> ([l], b)  
  (list, x) >>= f = let (list', y) = f x  
                   in (list ++ list', y)
```

- Výpočty s „logováním“.
- >>=: udržuje se společný log celého výpočtu.
- Obecněji: monáda píšeře. Uvidíme později v semestru.

Korektní instance monády musí splňovat:

- *levá identita*

`return x >>= f ≡ f x`

- *pravá identita*

`mx >>= return ≡ mx`

- *asociativita*

`(mx >>= f) >>= g ≡ mx >>= (\x -> f x >>= g)`

Užitečné funkce třídy Monad

`(<=<) :: Monad m =>`
`(b -> m c) -> (a -> m b) -> (a -> m c)`

`f <=< g = (\x -> g x >>= f)`

- ekvivalent kompozice běžných funkcí (.)
- existuje i obrácená varianta `>=>`

`join :: Monad m => m (m a) -> m a`

`join mmx = mmx >>= id`

- slučování vnořených kontextů
- ★ teoretici často místo `bindu` definují `join`

Java

- `Optional<T> = Maybe T`
`o.flatMap(f) = o »= f`

Rust

- `Option<T> = Maybe T`
`o.and_then(f) = o »= f`
- `Result<T, E> = Either E T`
`r.and_then(f) = r »= f`

C++ 23

- `std::optional<T> = Maybe T`
`o.and_then(f) = o »= f`
- `std::expected<T, E> = Either E T`
`e.and_then(f) = e »= f`

... a spoustu ostatních (např. `Async/Future/Promise` monády v mnohých jazycích)

Samostatné programování

Příklad: monáda Maybe

Mějme následující typ reprezentující aritmetické výrazy:

```
data Expr = Const Int |  
           Add Expr Expr | Sub Expr Expr |  
           Mul Expr Expr | Div Expr Expr  
           deriving (Show)
```

Naprogramujte funkci `eval :: Expr -> Maybe Int`, která vyhodnotí zadaný výraz a *ošetří dělení nulou*.

Poté přidejte datovému typu `Expr` ještě if-then-else hodnotový konstruktor `IfZero Expr Expr Expr` a odpovídajícím způsobem upravte funkci `eval` .

Příklad: monáda Either

Naprogramujte pro předchozí příklad také funkci `evalVerbose :: Expr -> Either String Int`, která vyhodnotí zadaný výraz a při chybě vrátí odpovídající chybový řetězec.

Příklad: kombinace hodnot

Naprogramujte funkci

```
myLiftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
```

pomocí funkcí (`>>=`) a `return`.

Příklad: skládání více výpočtů

Na prvním semináři jste viděli funkci `sequence :: [IO a] -> IO [a]`, která vyrábí ze seznamu výpočtů jeden výpočet, který vrací seznam výsledků všech těchto výpočtů.

Knihovní funkce `sequence` je ve skutečnosti obecnější:

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

Co tato funkce dělá pro monády **Maybe** a **List**?

Až to zjistíte, naprogramujte vlastní funkci `mySequence :: Monad m => [m a] -> m [a]`.

Příklad: vlastní monáda

Mějme vlastní definici typu **Either** a b a odpovídající instance **Functor** a **Applicative**⁶.

```
import Control.Applicative
```

```
data MyEither a b = MyLeft a | MyRight b deriving Show
```

```
instance Functor (MyEither a) where
  fmap f (MyLeft l) = MyLeft l
  fmap f (MyRight r) = MyRight (f r)
```

```
instance Applicative (MyEither a) where
  pure x = MyRight x
  liftA2 f (MyRight r1) (MyRight r2) = MyRight (f r1 r2)
  liftA2 f (MyLeft l1) _ = MyLeft l1
  liftA2 f _ (MyLeft l2) = MyLeft l2
```

Naprogramujte pro **MyEither** instanci třídy **Monad**.

⁶Zatím neřešte, co je zač třída **Applicative**.

Příklad: ekvivalence definic monády

Typová třída **Monad** m vyžaduje buď definice funkcí

```
return :: m a
```

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

nebo definice funkcí

```
return :: m a
```

```
join :: m (m a) -> m a
```

Ukažte, že tyto definice jsou ekvivalentní. Tj.,

- pomocí `return` a `(>>=)` jde vyrobit `join`, a
- pomocí `return` a `join` jde vyrobit `(>>=)`.

Nápověda: Možná se vám bude hodit, že třída **Monad** je podtřídou **Functor**.