Záznamy, vlastní moduly, pokročilá syntaxe, funktory

IB016 Seminář z funkcionálního programování

Fakulta informatiky, Masarykova univerzita

Záznamy

Záznamy

Připomenutí: psaní getterů přes pattern-matching

```
data Student = Student Int String Int String
getUCO (Person u _ _ _) = u
getName (Person _ n _ _) = n
getAge (Person _ _ a _) = a
getCity (Person _ _ _ c) = c
```

Záznamy

Připomenutí: psaní getterů přes pattern-matching

```
data Student = Student Int String Int String
getUCO (Person u _ _ _) = u
getName (Person _ n _ _) = n
getAge (Person _ _ a _) = a
getCity (Person _ _ _ c) = c
```

Záznamy (*records*): pojmenování hodnot v definici

```
data Student = Student { uco :: Int,
                         name :: String,
                         age :: Int,
                         city :: String }
```

Vytvoří automaticky odpovídající *selektory*:

- `uco :: Student -> Int`
- `name :: Student -> String`
- `age :: Student -> Int`
- `city :: Student -> String`

Záznamy: konstrukce

Klasickým voláním konstruktoru:

```
s1 = Student 123 "Ales" 19 "Praha"
```

Pomocí jmen polí:

```
s2 = Student { uco = 359542,
               name = "Martin",
               age = 32,
               city = "Brno" }
```

Z jiné hodnoty úpravou polí:

```
s3 = s2 { uco = 42, city = "Olomouc" }
```

Záznamy: použití

Pomocí selektorů:

```
isOlderThan20 s = age s > 20
```

V definici podle vzoru

```
isOlderThan20 (Student { age = a }) = a > 20
```

V definici podle vzoru zkráceně (rozšíření `NamedFieldPuns`[1])

```
isOlderThan20 (Student { age }) = age > 20
```

V definici podle vzoru ještě zkráceněji (rozšíření `RecordWildCards`)

```
isOlderThan20 (Student { .. }) = age > 20
```

[1] `{-# LANGUAGE NamedFieldPuns #-}` úplně na vrchu souboru nebo ghci -XNamedFieldPuns

Záznamy: Více konstruktorů

Lze použít i na datové typy s více hodnotovými konstruktory:

```
data BinTree a = Empty | Node { btValue :: a,
                                btLeft  :: BinTree a,
                                btRight :: BinTree a
                              }

valueOrDefault :: a -> BinTree a -> a
valueOrDefault def Empty = def
valueOrDefault _ (Node { btValue = v }) = v
```

Vlastní moduly

Vlastní moduly: účel

Logické členění většího kódu

- související funkce zabalené do jednoho „jmenného prostoru“
- program rozdělený na nezávislé a znovupoužitelné části

Zapouzdření

- můžou být veřejné (exportované) jen vybrané funkce
- datový typ nemusí mít veřejný hodnotový konstruktor; uživatel může dostat možnost hodnoty vytvářet jen pomocí na to určených funkcí („chytré konstruktory“)

Vlastní moduly: syntaxe

Veřejné všechny funkce:

```
module Foo where
-- zbytek souboru, definice funkcí a typů
```

Veřejné jen funkce `foo` a `bar`:

```
module Foo (foo, bar) where
```

Veřejná funkce `foo` a datový typ `Tree`:

```
module Foo (foo, Tree) where
```

Veřejná funkce `foo`, datový typ `Tree` a jeho hodnotové konstruktory `Empty` a `Node`:

```
module Foo (foo, Tree(Empty, Node)) where
```

Veřejná funkce `foo`, datový typ `Tree` a všechny jeho hodnotové konstruktory:

```
module Foo (foo, Tree(..)) where
```

Vlastní moduly: příklady

Napište modul **Stack**, který exportuje datový typ **Stack** a a funkce

- `empty :: Stack a`,
- `push :: a -> Stack a -> Stack a`,
- `top :: Stack a -> Maybe a`,
- `pop :: Stack a -> Stack a`.

Napište modul **Queue**, který exportuje datový typ **Queue** a a funkce

- `empty :: Queue a`,
- `enqueue :: a -> Queue a -> Queue a`,
- `first :: Queue a -> Maybe a`,
- `dequeue :: Queue a -> Queue a`.

Pokročilá syntaxe

Pokročilejší syntaxe: case

Připomenutí: **case**: používání vzorů uvnitř funkce

```
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe _ []     = []
mapMaybe f (x:xs) = case f x of
                      Just v  -> v : mapMaybe f xs
                      Nothing ->     mapMaybe f xs
```

Pokročilejší syntaxe: case

Připomenutí: **case**: používání vzorů uvnitř funkce

```
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe _ []     = []
mapMaybe f (x:xs) = case f x of
                      Just v  -> v : mapMaybe f xs
                      Nothing ->     mapMaybe f xs
```

Rozšíření LambdaCase pro zkrácení `\x -> case x of`

```
{-# LANGUAGE LambdaCase #-}
defaultedMap :: (a -> b) -> b -> [Maybe a] -> [b]
defaultedMap f d = map $ \case
                           Just v  -> f v
                           Nothing -> d
```

Pokročilejší syntaxe: stráže (guards)

Připomenutí: vyhnutí se násobným **if**ům

```haskell
data Ordering = LT | EQ | GT deriving Show

compare :: Ord a => a -> a -> Ordering
compare x y
  | x < y     = LT
  | x == y    = EQ
  | otherwise = GT
```

Pokročilejší syntaxe: stráže (guards)

Připomenutí: vyhnutí se násobným **if**ům

```haskell
data Ordering = LT | EQ | GT deriving Show

compare :: Ord a => a -> a -> Ordering
compare x y
  | x < y     = LT
  | x == y    = EQ
  | otherwise = GT
```

Výchozí rozšíření PatternGuards umožňuje použít ve strážích i vzory

```haskell
-- lookup :: Eq a => a -> [(a, b)] -> Maybe b
defaultedLookup :: [(Int, a)] -> a -> Int -> a
defaultedLookup db def key
    | key >= 0, Just val <- lookup key db = val
    | otherwise                           = def
```

Pokročilejší syntaxe: $

Připomenutí: zbavujeme se závorek dolarem

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Operátor aplikace funkce (priorita 0, asociativita zprava)

- `f $ g $ x` ≡ `f (g x)`
- `f . g $ h x` ≡ `(f . g) (h x)`

Pokročilejší syntaxe: $

Připomenutí: zbavujeme se závorek dolarem

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Operátor aplikace funkce (priorita 0, asociativita zprava)

- `f $ g $ x` ≡ `f (g x)`
- `f . g $ h x` ≡ `(f . g) (h x)`

Zamyšlení:
mezera jako operátor aplikace (priorita 10, asociativita zleva)

- `f g x` ≡ `(f g) x`

Pokročilejší syntaxe: $

Připomenutí: zbavujeme se závorek dolarem

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Operátor aplikace funkce (priorita 0, asociativita zprava)

- `f $ g $ x` ≡ `f (g x)`
- `f . g $ h x` ≡ `(f . g) (h x)`

Zamyšlení:
mezera jako operátor aplikace (priorita 10, asociativita zleva)

- `f g x` ≡ `(f g) x`

```
filter (>10) ( map (^2)   ( filter even [1..10] ) )
```

Pokročilejší syntaxe: $

Připomenutí: zbavujeme se závorek dolarem

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Operátor aplikace funkce (priorita 0, asociativita zprava)

- `f $ g $ x` ≡ `f (g x)`
- `f . g $ h x` ≡ `(f . g) (h x)`

Zamyšlení:
mezera jako operátor aplikace (priorita 10, asociativita zleva)

- `f g x` ≡ `(f g) x`

```
  filter (>10) ( map (^2)   ( filter even [1..10] ) )
( filter (>10) . map (^2) ) ( filter even [1..10] )
```

Pokročilejší syntaxe: $

Připomenutí: zbavujeme se závorek dolarem

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Operátor aplikace funkce (priorita 0, asociativita zprava)

- `f $ g $ x` ≡ `f (g x)`
- `f . g $ h x` ≡ `(f . g) (h x)`

Zamyšlení:
mezera jako operátor aplikace (priorita 10, asociativita zleva)

- `f g x` ≡ `(f g) x`

```
  filter (>10) ( map (^2)   ( filter even [1..10] ) )
( filter (>10) . map (^2) ) ( filter even [1..10] )
  filter (>10) . map (^2)  $  filter even [1..10]
```

Pokročilejší syntaxe: $

Připomenutí: zbavujeme se závorek dolarem

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Operátor aplikace funkce (priorita 0, asociativita zprava)

- `f $ g $ x` ≡ `f (g x)`
- `f . g $ h x` ≡ `(f . g) (h x)`

Zamyšlení:
mezera jako operátor aplikace (priorita 10, asociativita zleva)

- `f g x` ≡ `(f g) x`

```
  filter (>10) ( map (^2)   ( filter even [1..10] ) )
( filter (>10) . map (^2) ) ( filter even [1..10] )
  filter (>10) . map (^2)  $  filter even [1..10]
  filter (>10) $ map (^2)  $  filter even [1..10]
```

Pokročilejší syntaxe: @

Někdy chceme u vstupního argumentu funkce použít vzory, ale zároveň pak použít argument jako celek.

```
foo ((Person sex age weight) : ps)
    = if {- ... -} then (Person sex age weight) : foo ps else foo ps
```

Pokročilejší syntaxe: @

Někdy chceme u vstupního argumentu funkce použít vzory, ale zároveň pak použít argument jako celek.

```
foo ((Person sex age weight) : ps)
    = if {- ... -} then (Person sex age weight) : foo ps else foo ps
```

Můžeme použít *as-patterns*:

```
foo (p@(Person sex age weight) : ps)
       = if {- ... -} then p : foo ps else foo ps
```

type a newtype

```haskell
type String = [Char]
type Matrix a = [[a]]
```

- typový alias: jen nové pojmenování, zaměnitelné s původním typem
- výjimka: nelze použít při instanciování typových tříd
- pouze pro přehlednost kódu, pro překladač transparentní

type a newtype

```
type String = [Char]
type Matrix a = [[a]]
```

- typový alias: jen nové pojmenování, zaměnitelné s původním typem
- výjimka: nelze použít při instanciování typových tříd
- pouze pro přehlednost kódu, pro překladač transparentní

```
newtype Matrix a = M { unM :: [[a]] } deriving Show
```

- nový typ (jako **data**) ⇒ typová kontrola překladače
- musí mít právě jeden unární hodnotový konstruktor
- nutnost „rozbalování“ a „balení“

```
M (map (map (+4)) (unM matrix))
```

- časté využití: psaní jiné instance pro týž typ
- rychlejší než **data**, další rozdíly s rozšířeními GHC
  (nad rámec kurzu: `-XGeneralisedNewtypeDeriving`)

Typované díry I.

Otypování samostatného výrazu:

- přes povel interpretu `:t`

Typované díry I.

Otypování samostatného výrazu:

- přes povel interpretu `:t`

Otypování výrazu v kódu:

- použijeme typovanou díru: `[ "a", "b", _ ]`
- vygeneruje typovou chybu obsahující požadovaný typ

```
> [ "a", "b", _ ]
<interactive>:1:12: error:
    * Found hole: _ :: [Char]
    * In the expression: ["a", "b", _]
    ...
```

Typované díry II.

Typované díry:

- názvy začínají podtržítkem
- ladění typových chyb ve složitějším kódu
- prototypování programu (namísto `undefined`)
- chyby lze odložit až do okamžiku volání pomocí přepínače GHC/GHCi `-fdefer-typed-holes`
- pozor: i stejně pojmenované díry jsou *různé* díry
- více v dokumentaci GHC a na GHC wiki

Užitečné datové typy

Datové typy `Set` a `Map`

Množiny a asociativní mapy (též *slovníky*):

- **`Set`** a
  - a je typ hodnoty, musí být v **`Ord`**
- **`Map`** k v
  - k je typ klíče, musí být **`Ord`**
  - v je typ hodnoty
- moduly `Data.Set` a `Data.Map`
  - balík `containers`, součást běžné distribuce GHC
  - **`Map`** dvou typů (*lazy* a *strict*)
  - vhodný kvalifikovaný import
- logaritmické vkládání, odstraňování, zjišťování minima a maxima

Typový konstruktor Either

```haskell
data Either a b = Left a
                | Right b
                deriving (Eq, Ord, Read, Show)
```

- používá se, když může mít výpočet dva typy výsledků
- často se používá jako zobecnění **Maybe**
  - **Left** a označuje chybný výpočet, hodnota specifikuje chybu
  - **Right** b označuje korektní výpočet, hodnota je výsledkem

```haskell
myDiv :: Int -> Int -> Either String Int
myDiv x 0 | x < 0  = Left "-inf"
          | x > 0  = Left "+inf"
          | x == 0 = Left "NaN"
myDiv x y = Right (div x y)
```

Připomenutí: Typové třídy

Typové třídy

Typová třída = kolekce jmen funkcí + typů

```
class MakesSound a where
    doSound :: a -> String
```

Typové třídy

Typová třída = kolekce jmen funkcí + typů

```
class MakesSound a where
    doSound :: a -> String
```

Instance typové třídy = implementace požadovaných funkcí pro daný typ

```
data Animal = Dog | Cat
data MoneyPrinter = MP { amount :: Int }

instance MakesSound Animal where
    doSound Dog = "haf"
    doSound Cat = "mnau"

instance MakesSound MoneyPrinter where
    doSound mp = "b" ++ replicate (amount mp) 'r'
```

Funktory

Druhy aneb typování typů

- všechny konkrétní typy jsou druhu `*`

```
Integer :: *
Maybe Int :: *
Either String Int :: *
BinTree (Int, [Int]) :: *
```

Druhy aneb typování typů

- všechny konkrétní typy jsou druhu `*`

```
Integer :: *
Maybe Int :: *
Either String Int :: *
BinTree (Int, [Int]) :: *
```

- typové konstruktory vyšší arity jsou vlastně „typové funkce“

```
[] :: * -> *
Maybe :: * -> *
(,) :: * -> * -> *
Either :: * -> * -> *
```

Druhy aneb typování typů

- všechny konkrétní typy jsou druhu `*`

```
Integer :: *
Maybe Int :: *
Either String Int :: *
BinTree (Int, [Int]) :: *
```

- typové konstruktory vyšší arity jsou vlastně „typové funkce“

```
[] :: * -> *
Maybe :: * -> *
(,) :: * -> * -> *
Either :: * -> * -> *
```

- opět platí princip částečné aplikace

```
Either String :: * -> *
```

- GHCi definuje povel `:k` na určení druhu

Motivace: map

- Funkce `map` na seznamech:

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

Motivace: map

- Funkce `map` na seznamech:

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

- Funkce `map` na binárních stromech:

```
data BinTree a = Node a (BinTree a) (BinTree a) | Empty

treeMap :: (a -> b) -> BinTree a -> BinTree b
treeMap _ Empty = Empty
treeMap f (Node v l r) = Node (f v) (treeMap f l) (treeMap f r)
```

Motivace: map

- Funkce `map` na seznamech:

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

- Funkce `map` na binárních stromech:

```
data BinTree a = Node a (BinTree a) (BinTree a) | Empty

treeMap :: (a -> b) -> BinTree a -> BinTree b
treeMap _ Empty = Empty
treeMap f (Node v l r) = Node (f v) (treeMap f l) (treeMap f r)
```

- Nedalo by se to zobecnit? Jaký je obecný typ funkce `map`?

Typová třída Functor

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

- pro typy, přes které se dá „mapovat“
- třída pro „unární typové funkce“, tedy věci druhu `* -> *`
  - instance pro `[]`, `BinTree`, `Maybe`
  - ne pro konkrétní typy (`[String]`, `BinTree a`, `Maybe Int`)

Typová třída Functor

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

- pro typy, přes které se dá „mapovat“
- třída pro „unární typové funkce“, tedy věci druhu `* -> *`
  - instance pro `[]`, `BinTree`, `Maybe`
  - ne pro konkrétní typy (`[String]`, `BinTree a`, `Maybe Int`)
- jiný pohled: funktory tvoří kontext/kontejner pro typy (obalují je další strukturou); `fmap` tento kontext nemění

Typová třída Functor

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

- pro typy, přes které se dá „mapovat“
- třída pro „unární typové funkce“, tedy věci druhu `* -> *`
  - instance pro `[]`, `BinTree`, `Maybe`
  - ne pro konkrétní typy (`[String]`, `BinTree a`, `Maybe Int`)
- jiný pohled: funktory tvoří kontext/kontejner pro typy (obalují je další strukturou); `fmap` tento kontext nemění
- operátor `<$>` ≡ infixový `fmap`

```
recip  $  2        ⇝  0.5
recip <$> Just 2   ⇝  Just 0.5
```

Pravidla pro třídu Functor

Instance třídy **`Functor`** musí splňovat dvě pravidla:

- `fmap id` $\equiv$ `id`
- `fmap (f . g)` $\equiv$ `fmap f . fmap g`

Pravidla pro třídu Functor

Instance třídy **`Functor`** musí splňovat dvě pravidla:

- `fmap id ≡ id`
- `fmap (f . g) ≡ fmap f . fmap g`

Pravidla musí platit!

- překladač se spoléhá na výše uvedená pravidla
- jejich platnost musí ověřit programátor (!)
- pro všechny knihovní instance platí

Instance třídy Functor I

- ```
  instance Functor [] where
      fmap :: (a -> b) -> [a] -> [b] -- pozn. 1
  ```

[1] Uvádět typy v instancích je dovoleno pouze s rozšířením `InstanceSigs`. Ve svých instancích typy explicitně nepište (nebo si zapněte rozšíření).
  ```

Instance třídy Functor I

- ```
  instance Functor [] where
      fmap :: (a -> b) -> [a] -> [b] -- pozn. 1
      fmap = map
  ```

- ```
  instance Functor BinTree where
      fmap :: (a -> b) -> BinTree a -> BinTree b
  ```

[1] Uvádět typy v instancích je dovoleno pouze s rozšířením `InstanceSigs`. Ve svých instancích typy explicitně nepište (nebo si zapněte rozšíření).

Instance třídy Functor I

- ```
  instance Functor [] where
      fmap :: (a -> b) -> [a] -> [b] -- pozn. 1
      fmap = map
  ```

- ```
  instance Functor BinTree where
      fmap :: (a -> b) -> BinTree a -> BinTree b
      fmap = treeMap
  ```

[1] Uvádět typy v instancích je dovoleno pouze s rozšířením `InstanceSigs`. Ve svých instancích typy explicitně nepište (nebo si zapněte rozšíření).

Instance třídy Functor II

- **Either** je binární typový konstruktor, musíme tedy udělat instanci pro jeho částečnou aplikaci (**Either** e :: * -> *)

```
instance Functor (Either e) where
    fmap :: (a -> b) -> Either e a -> Either e b
```

Instance třídy Functor II

- **Either** je binární typový konstruktor, musíme tedy udělat instanci pro jeho částečnou aplikaci (**Either** e :: * -> *)

```
instance Functor (Either e) where
    fmap :: (a -> b) -> Either e a -> Either e b
    fmap f (Right x) = Right (f x)
    fmap f (Left x) = Left x
```

- Obdobně pro typový konstruktor dvojice:

```
instance Functor ((,) w) where
    fmap :: (a -> b) -> (w, a) -> (w, b)
```

Instance třídy Functor II

- **Either** je binární typový konstruktor, musíme tedy udělat instanci pro jeho částečnou aplikaci (**Either** e :: * -> *)

```
instance Functor (Either e) where
    fmap :: (a -> b) -> Either e a -> Either e b
    fmap f (Right x) = Right (f x)
    fmap f (Left x) = Left x
```

- Obdobně pro typový konstruktor dvojice:

```
instance Functor ((,) w) where
    fmap :: (a -> b) -> (w, a) -> (w, b)
    fmap f (x, y) = (x, f y)
```

Instance třídy Functor III

- ```
  instance Functor IO where
      fmap :: (a -> b) -> IO a -> IO b
  ```
  ```

Instance třídy Functor III

```haskell
instance Functor IO where
    fmap :: (a -> b) -> IO a -> IO b
    fmap f action = do
        result <- action
        pure (f result)
    fmap' f action = action >>= (pure . f)
```

★ Instance třídy Functor IV

Funkce je binární typový konstruktor `(->) :: * -> * -> *`

★ Instance třídy Functor IV

Funkce je binární typový konstruktor `(->) :: * -> * -> *`
Částečná aplikace na jeden argument:
`(->) r :: * -> *` (tedy „funkce z `r`“)

★ Instance třídy Functor IV

Funkce je binární typový konstruktor `(->) :: * -> * -> *`
Částečná aplikace na jeden argument:
`(->) r :: * -> *` (tedy „funkce z `r`“)

```
instance Functor ((->) r) where
```

★ Instance třídy Functor IV

Funkce je binární typový konstruktor `(->) :: * -> * -> *`
Částečná aplikace na jeden argument:
`(->) r :: * -> *` (tedy „funkce z `r`“)

```
instance Functor ((->) r) where
    fmap :: (a -> b) -> (r -> a) -> (r -> b)
```

★ Instance třídy Functor IV

Funkce je binární typový konstruktor `(->) :: * -> * -> *`
Částečná aplikace na jeden argument:
`(->) r :: * -> *` (tedy „funkce z `r`“)

```
instance Functor ((->) r) where
    fmap :: (a -> b) -> (r -> a) -> (r -> b)
    fmap f g = (\x -> f (g x))   -- === f . g
```

★ Instance třídy Functor IV

Funkce je binární typový konstruktor `(->) :: * -> * -> *`
Částečná aplikace na jeden argument:
`(->) r :: * -> *` (tedy „funkce z `r`“)

```
instance Functor ((->) r) where
    fmap :: (a -> b) -> (r -> a) -> (r -> b)
    fmap f g = (\x -> f (g x))   -- === f . g
```

Intuice: hodnota závisí na kontextu (typu `r`), který teprve přijde.

```
greetings = (\polite -> if polite then "Good morning"
                                  else "Hello there")
ave = (++ ", Caesar.") <$> greetings
```

Samostatné programování

Příklady: Záznamy

Uvažte datový typ **Student** definovaný na začátku dnešního semináře, tj.

```
data Student = Student { uco :: Int,
                         name :: String,
                         age :: Int,
                         city :: String }
```

Napište funkci `namesFromCity :: String -> [Student] -> [String]`, která pro zadaný název města a seznam studentů vrátí seznam jmen studentů ze zadaného města.

Zkuste napsat jak vlastní rekurzivní funkci, tak řešení pomocí `filter` a `map`. V rekurzivní funkci zkuste použít i definici podle vzoru.

Příklady: `Data.Set`

Pomocí datové struktury **`Set`** z modulu **`Data.Set`** napište funkci `containsDuplicates :: Ord a => [a] -> Bool`, která o zadaném seznamu rozhodne, jestli se v něm nějaký prvek vyskytuje alespoň dvakrát.

Pro přehled operací modulu **`Data.Set`** se nebojte konzultovat dokumentaci.

Příklady: Data.Map

Pomocí datové struktury **Map** z modulu **Data.Map** napište funkci `lengthStats :: [[a]] -> String`, která textově popíše histogram délek zadaných seznamů. Tedy například

```
lengthStats ["abc","ab","ba","ahoj","yyz","aabbcc"] =
 "delky 2: 2, delky 3: 2, delky 4: 1, delky 6: 1"
```

Zkuste napsat jak vlastní rekurzivní funkci, tak řešení pomocí funkce `map`, foldů a vhodných obdob z modulu **Data.Map**.

Nebojte se definovat si vlastní pomocné funkce.

Pro přehled operací modulu **Data.Map** se nebojte konzultovat dokumentaci.

Příklady: Moduly a záznamy

Napište modul **BST**, který obsahuje záznam **BinSearchTree** a a následující funkce pro práci s binárními vyhledávacími stromy:

- prázdný binární vyhledávací strom,
- hledání prvku v binárním vyhledávacím stromě (vrací **Bool**),
- vložení prvku do binárního vyhledávacího stromu.

Vyvažování stromu při vkládání nijak řešit nemusíte (pokud chcete, samozřejmě můžete).

Chcete z modulu **BST** exportovat konstruktor datové struktury **BinSearchTree**? Proč?

Vlastní instance třídy Functor

Mějme vlastní verzi datového typu **Maybe** a:

```
data MyMaybe a = MyNothing | MyJust a deriving (Show)
```

Napište pro **MyMaybe** instanci typové třídy **Functor**.

Příklady: IO

Pokud vám zbyde čas, zkuste se vrátit k příkladům na **IO**, které jste minule nestihli.