FLP druhé cvičení
Haskell: datové typy, monády (IO)

Stanislav Židek, Peter Matula

Department of Information Systems
Faculty of Information Technology
Brno University of Technology

Funkcionální a logické programování, 2014/2015

Vlastní datové typy

Datové typy (DT) můžeme tvořit třemi způsoby:

Vlastní datové typy

Datové typy (DT) můžeme tvořit třemi způsoby:

- **`type`** přejmenování existujícího DT

```
type SeznamCelychCisel = [Int]
```

Vlastní datové typy

Datové typy (DT) můžeme tvořit třemi způsoby:

- **`type`** přejmenování existujícího DT

```
type SeznamCelychCisel = [Int]
```

- **`newtype`** zabalení existujícího DT (raději zapomenout)

```
newtype Natural = Nat Int
```

Vlastní datové typy

Datové typy (DT) můžeme tvořit třemi způsoby:

- **type** přejmenování existujícího DT

```
type SeznamCelychCisel = [Int]
```

- **newtype** zabalení existujícího DT (raději zapomenout)

```
newtype Natural = Nat Int
```

- **data** fungl nový DT

```
data TypeConstructor <params>
   = DataConstructor1 <params>
   | DataConstructor2 <params>
   ...
   | DataConstructorN <params>
```

Vlastní datové typy

Datové typy (DT) můžeme tvořit třemi způsoby:

- **type** přejmenování existujícího DT

```
type SeznamCelychCisel = [Int]
```

- **newtype** zabalení existujícího DT (raději zapomenout)

```
newtype Natural = Nat Int
```

- **data** fungl nový DT

```
data TypeConstructor <params>
  = DataConstructor1 <params>
  | DataConstructor2 <params>
  ...
  | DataConstructorN <params>
```

Poznámka: typový konstruktor $\times$ datový konstruktor

Příklad: Datový typ pro teplotu

```
data Teplota
     = Nula
     | Kelvin Float
     | Celsius Float
```

Příklad: Datový typ pro teplotu

```
data Teplota
     = Nula
     | Kelvin Float
     | Celsius Float

     deriving Show
```

Příklad: Datový typ pro teplotu

```
data Teplota
     = Nula
     | Kelvin Float
     | Celsius Float

     deriving Show
```

Příklad použití ve funkci:

```
toKelvin :: Teplota -> Float

toKelvin Nula = 273.15
toKelvin (Kelvin x) = x
toKelvin (Celsius x) = x + 273.15
```

Příklad: Datový typ pro teplotu

```
data Teplota
     = Nula
     | Kelvin Float
     | Celsius Float

     deriving Show
```

Příklad použití ve funkci:

```
toKelvin :: Teplota -> Float

toKelvin Nula = 273.15
toKelvin (Kelvin x) = x
toKelvin (Celsius x) = x + 273.15
```

Pattern matching přes konstruktory!

Typové proměnné, rekurzivní DT

Definice analogická ke standardnímu Haskellovskému seznamu může vypadat například takto:

```
data Seznam a
     = Prazdny
     | Kons a (Seznam a)
```

- Typová proměnná **a** umožňuje tvořit seznamy nad libovolným DT.
- Díky rekurzi může seznam obsahovat neomezeně prvků.

Poznámka: kinds

Úkol 1.: Vektor

Definujte vlastní datový typ pro vektor, jeho délku zjistěte pouze při konstrukci.

```
data Vector a = ...
```

Úkol 1.: Vektor

Definujte vlastní datový typ pro vektor, jeho délku zjistěte pouze při konstrukci.

```
data Vector a = Vec Int [a]
```

Úkol 1.: Vektor

Definujte vlastní datový typ pro vektor, jeho délku zjistěte pouze při konstrukci.

```
data Vector a = Vec Int [a]
```

Vytvořte funkci `initVector`, která vytvoří vektor ze seznamu, a funkci `dotProd`, která spočte skalární součin vektorů.

```
initVector :: [a] -> Vector a
initVector l = ...


dotProd :: Num a => Vector a -> Vector a -> a
dotProd ... ... =
```

Úkol 1.: Vektor

Definujte vlastní datový typ pro vektor, jeho délku zjistěte pouze při konstrukci.

```
data Vector a = Vec Int [a]
```

Vytvořte funkci `initVector`, která vytvoří vektor ze seznamu, a funkci `dotProd`, která spočte skalární součin vektorů.

```
initVector :: [a] -> Vector a
initVector l = Vec (length l) l


dotProd :: Num a => Vector a -> Vector a -> a
dotProd ... ... =
```

Úkol 1.: Vektor

Definujte vlastní datový typ pro vektor, jeho délku zjistěte pouze při konstrukci.

```
data Vector a = Vec Int [a]
```

Vytvořte funkci `initVector`, která vytvoří vektor ze seznamu, a funkci `dotProd`, která spočte skalární součin vektorů.

```
initVector :: [a] -> Vector a
initVector l = Vec (length l) l


dotProd :: Num a => Vector a -> Vector a -> a
dotProd (Vec l1 v1) (Vec l2 v2) =
```

Úkol 1.: Vektor

Definujte vlastní datový typ pro vektor, jeho délku zjistěte pouze při konstrukci.

```
data Vector a = Vec Int [a]
```

Vytvořte funkci `initVector`, která vytvoří vektor ze seznamu, a funkci `dotProd`, která spočte skalární součin vektorů.

```
initVector :: [a] -> Vector a
initVector l = Vec (length l) l


dotProd :: Num a => Vector a -> Vector a -> a
dotProd (Vec l1 v1) (Vec l2 v2) =
  if l1 /= l2
    then error "Nesouhlasi delka."
    else sum $ zipWith (*) v1 v2
```

Record syntax

```
data Student =
    Student
        { jmeno :: String
        , stip :: Int
        , phd :: Bool
        } deriving Show
```

Intuitivní přístup k položkám:

```
jmeno :: Student -> String
stip :: Student -> Int
phd :: Student -> Bool
```

Vytvořte si svého oblíbeného doktoranda:

```
Student { jmeno="S. Zidek", stip=7800, phd=True }
```

$\sim$

```
Student "S. Zidek" 7800 True
```

Úkol 2.: Doktorandi dostanou přidáno

Naprogramujte funkci, která doktorandům zdvojnásobí stipendium.

```
pdhPayRise :: [Student] -> [Student]
phdPayRise = map ...
```

Úkol 2.: Doktorandi dostanou přidáno

Naprogramujte funkci, která doktorandům zdvojnásobí stipendium.

```
pdhPayRise :: [Student] -> [Student]
phdPayRise = map  ppr where
   ppr s = ...
```

Úkol 2.: Doktorandi dostanou přidáno

Naprogramujte funkci, která doktorandům zdvojnásobí stipendium.

```
pdhPayRise :: [Student] -> [Student]
phdPayRise = map  ppr where
   ppr  s =
      if phd s
        then ...
        else ...
```

Úkol 2.: Doktorandi dostanou přidáno

Naprogramujte funkci, která doktorandům zdvojnásobí stipendium.

```
pdhPayRise :: [Student] -> [Student]
phdPayRise = map  ppr where
   ppr s =
      if phd s
        then  Student (jmeno s) (2*stip s) True
        else ...
```

Úkol 2.: Doktorandi dostanou přidáno

Naprogramujte funkci, která doktorandům zdvojnásobí stipendium.

```
pdhPayRise :: [Student] -> [Student]
phdPayRise = map  ppr where
   ppr s =
      if phd s
        then  Student (jmeno s) (2*stip s) True
        else  s
```

Úkol 3.: Lambda výrazy

Vytvořte DT pro výrazy v lambda kalkulu, proměnné reprezentujte řetězci nebo genericky pro jakýkoliv DT.

```
data LExp = ...
   ...
   ...
   deriving Show
```

Úkol 3.: Lambda výrazy

Vytvořte DT pro výrazy v lambda kalkulu, proměnné reprezentujte řetězci nebo genericky pro jakýkoliv DT.

```
data LExp = LVar String
   ...
   ...
   deriving Show
```

Úkol 3.: Lambda výrazy

Vytvořte DT pro výrazy v lambda kalkulu, proměnné reprezentujte řetězci nebo genericky pro jakýkoliv DT.

```
data LExp = LVar String
    | LApp LExp LExp
   ...
   deriving Show
```

Úkol 3.: Lambda výrazy

Vytvořte DT pro výrazy v lambda kalkulu, proměnné reprezentujte řetězci nebo genericky pro jakýkoliv DT.

```
data LExp = LVar String
    | LApp LExp LExp
    | LAbs String LExp
   deriving Show
```

Úkol 3.: Lambda výrazy

Vytvořte DT pro výrazy v lambda kalkulu, proměnné reprezentujte řetězci nebo genericky pro jakýkoliv DT.

```
data LExp = LVar String
    | LApp LExp LExp
    | LAbs String LExp
   deriving Show
```

Vytvořte funkci `freeVars :: LExp -> [String]`, která vrátí seznam všech volných proměnných.

```
freeVars le =
```

Úkol 3.: Lambda výrazy

Vytvořte DT pro výrazy v lambda kalkulu, proměnné reprezentujte řetězci nebo genericky pro jakýkoliv DT.

```
data LExp = LVar String
   | LApp LExp LExp
   | LAbs String LExp
  deriving Show
```

Vytvořte funkci `freeVars :: LExp -> [String]`, která vrátí seznam všech volných proměnných.

```
freeVars le = fv le [] where
  fv (LVar v) l =
  fv (LApp e1 e2) l =
  fv (LAbs v e) l =
```

Úkol 3.: Lambda výrazy

Vytvořte DT pro výrazy v lambda kalkulu, proměnné reprezentujte řetězci nebo genericky pro jakýkoliv DT.

```
data LExp = LVar String
   | LApp LExp LExp
   | LAbs String LExp
  deriving Show
```

Vytvořte funkci `freeVars :: LExp -> [String]`, která vrátí seznam všech volných proměnných.

```
freeVars le = fv le [] where
  fv (LVar v) l =
  fv (LApp e1 e2) l =
  fv (LAbs v e) l =  fv e (v:l)
```

Úkol 3.: Lambda výrazy

Vytvořte DT pro výrazy v lambda kalkulu, proměnné reprezentujte řetězci nebo genericky pro jakýkoliv DT.

```
data LExp = LVar String
   | LApp LExp LExp
   | LAbs String LExp
  deriving Show
```

Vytvořte funkci `freeVars :: LExp -> [String]`, která vrátí seznam všech volných proměnných.

```
freeVars le = fv le [] where
  fv (LVar v) l =
  fv (LApp e1 e2) l =  fv e1 l ++ fv e2 l
  fv (LAbs v e) l =  fv e (v:l)
```

Úkol 3.: Lambda výrazy

Vytvořte DT pro výrazy v lambda kalkulu, proměnné reprezentujte řetězci nebo genericky pro jakýkoliv DT.

```
data LExp = LVar String
   | LApp LExp LExp
   | LAbs String LExp
  deriving Show
```

Vytvořte funkci `freeVars :: LExp -> [String]`, která vrátí seznam všech volných proměnných.

```
freeVars le = fv le [] where
  fv (LVar v) l =  if elem v l then [] else [v]
  fv (LApp e1 e2) l =  fv e1 l ++ fv e2 l
  fv (LAbs v e) l =  fv e (v:l)
```

Úkol 3.: Lambda výrazy

Vytvořte DT pro výrazy v lambda kalkulu, proměnné reprezentujte řetězci nebo genericky pro jakýkoliv DT.

```
data LExp = LVar String
   | LApp LExp LExp
   | LAbs String LExp
  deriving Show
```

Vytvořte funkci `freeVars :: LExp -> [String]`, která vrátí seznam všech volných proměnných.

```
freeVars le = fv le [] where
  fv (LVar v) l =  if elem v l then [] else [v]
  fv (LApp e1 e2) l =  fv e1 l ++ fv e2 l
  fv (LAbs v e) l =  fv e (v:l)
```

Bylo by dost hloupé nezkusit si naimplementovat obdobné operace s lambda výrazy (redukce, . . .)

Úkrok stranou: konstrukce `case ... of`

- alternativní způsob definice funkcí
- umí pattern matching
- výhoda: je možné vnořování

Příklad:

```
delka seznam =
    case seznam of
        [] -> 0
        _:xs -> 1 + delka xs
```

Monády: motivace

- Zatím jsme se zabývali čistými (pure) funkcemi, tj. při stejném vstupu vracejí stejný výsledek.
- Potřebujeme vedlejší efekty =(
  - změna stavu
  - vstup/výstup
  - generování náhodných čísel
  - ...

Řešení: monády

Monády

Q: Co to je?
A: Abstraktní konstrukt z teorie kategorií...
(člověk hned ví...)

Monádu si můžeme představit jako model výpočtu.

Q: Co to umí?
A: Spoustu věcí:

- řízení toku programu
- stav výpočtu
- vedlejší efekty
- výjimky
- ...

Monády: `return`, `>>=`

Return:

```
return :: a -> m a
```

- „zabalení“ do monády, předstíračka výpočtu

`>>=` (čteme „bind“)

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

- navázání dvou výpočtů
- „vybalení“ výsledku prvního výpočtu
- předání tohoto výsledku druhému výpočtu

Monády: do notace

```
m1 >>= (\ a -> m2 a >>= (\ _ -> m3 a
    >>= (\ b -> m4 b
    >>= (\ c -> return (f c b)))))
```

...FUJ!

```
m1 >>= \ a ->
m2 a >>= \ _ ->
m3 a >>= \ b ->
m4 b >>= \ c ->
return (f c b)}
```

...menší fuj

```
do
    a <- m1
    m2 a
    b <- m3 a
    c <- m4 b
    return (f c b)
```

Monády: `Maybe`

Modelování výpočtu, který se nemusí povést:

```
data Maybe a
     = Nothing      -- nepovedlo se
     | Just a
```

Monády: `Maybe`

Modelování výpočtu, který se nemusí povést:

```
data Maybe a
     = Nothing      -- nepovedlo se
     | Just a
```

Například:

```
odmocnina :: (Ord a, Floating a) => a -> Maybe a
odmocnina x
     | x < 0 = Nothing
     | otherwise = Just (sqrt x)
```

Úkol 4.

Mějme funkci `otec`, která vrátí otce dané osoby, který ovšem nemusí existovat (dejme tomu z databáze, ale zadrátujeme ji do kódu):

```
otec :: String -> Maybe String
otec "Karel" = Just "Evzen"
otec "Evzen" = Just "Dobromil"
otec "Dobromil" = Just "Franta"
otec _ = Nothing
```

S jejím využitím definujte funkci `otcuvOtec` a `otcovaOtceOtec`.

1. pomocí vnořených `case ... of`
2. pomocí `>>=` a lambda funkcí

Úkol 4.

```
otcuvOtec x = ...
```

```
otcovaOtceOtec x = ...
```

```
ooo x = ...
```

Úkol 4.

```
otcuvOtec x =
  case otec x of
    Nothing -> Nothing
    Just y -> otec y


otcovaOtceOtec x = ...


ooo x = ...
```

Úkol 4.

```
otcuvOtec x =
  case otec x of
    Nothing -> Nothing
    Just y -> otec y

otcovaOtceOtec x =
  case otec x of
    Nothing -> Nothing
    Just y ->
      case otec y of
        Nothing -> Nothing
        Just z -> otec z

ooo x = ...
```

Úkol 4.

```
otcuvOtec x =
  case otec x of
    Nothing -> Nothing
    Just y -> otec y

otcovaOtceOtec x =
  case otec x of
    Nothing -> Nothing
    Just y ->
      case otec y of
        Nothing -> Nothing
        Just z -> otec z

ooo x =  otec x >>= otec >>= otec
```

Monády: `IO`

Monáda pro **vstup/výstup**.

Jak rozumět IO datovým typům?

- `IO a`: V/V akce s výsledkem typu `a`
- `IO String`: V/V akce vracející řetězec (např. načtení obsahu souboru)
- `IO ()`: V/V akce bez návratové hodnoty – důležitý je jen vedlejší efekt (např. tisk řetězce)

Příklady:

```
getLine :: IO String
putStrLn :: String -> IO ()
```

Poznámka: `:b IO`

Úkol 5.: Psychiatr

Vytvořte funkci `psychiatr`, která si bude s uživatelem povídat (zopakuje jinak to, co řekl). Skončí při prázdném řádku.

```
psychiatr :: IO ()
psychiatr = do

    ...
```

Úkol 5.: Psychiatr

Vytvořte funkci `psychiatr`, která si bude s uživatelem povídat (zopakuje jinak to, co řekl). Skončí při prázdném řádku.

```
psychiatr :: IO ()
psychiatr = do

   l <- getLine
   if length l == 0
     then return ()
     else do
       putStrLn ("Vy tedy tvrdite: " ++ l)
       psychiatr
```

Úkol 6.: Počet řádků v souboru

Užitečné funkce:

```
type FilePath = [Char] -- tj. jméno souboru
openFile :: FilePath -> IOMode -> IO Handle
hIsEOF :: Handle -> IO Bool
hGetLine :: Handle -> IO String
hClose :: Handle -> IO ()
```

Definujte funkci `countLines`, která zjistí počet řádků v souboru.

```
countLines :: FilePath -> IO Int
countLines file = do
    ...
```

Tip: `:set +I`

Úkol 6.: Počet řádků v souboru

```
countLines file = do
    ...
```

Úkol 6.: Počet řádků v souboru

```
countLines file = do
   handle <- openFile file ReadMode
   num <- cl' handle 0
   hClose handle
   return num
   where
      ...
```

Úkol 6.: Počet řádků v souboru

```
countLines file = do
  handle <- openFile file ReadMode
  num <- cl' handle 0
  hClose handle
  return num
  where
    cl' handle num = do
      eof <- hIsEOF handle
      if eof
        then return num
        else do
          hGetLine handle
          cl' handle (num+1)
```