Úvod, opakování, pokročilejší syntax; moduly, typové třídy

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2017

Vývojové prostředí

3 možnosti instalace:

- **samostatná instalace**
  - kompilátor GHC, balíčkovací systém cabal (viz dále)
  - možnost instalovat přímo z repozitářů distribuce

Vývojové prostředí

3 možnosti instalace:

- **samostatná instalace**
  - kompilátor GHC, balíčkovací systém cabal (viz dále)
  - možnost instalovat přímo z repozitářů distribuce
- **Haskell Platform**
  - instalace „všechno v jednom“
  - cross-platform
  - možná zbytečně velká

Vývojové prostředí

3 možnosti instalace:

- **samostatná instalace**
  - kompilátor GHC, balíčkovací systém cabal (viz dále)
  - možnost instalovat přímo z repozitářů distribuce
- **Haskell Platform**
  - instalace „všechno v jednom“
  - cross-platform
  - možná zbytečně velká
- **The Haskell Tool Stack**
  - cross-platform systém pro pokročilý vývoj
  - umožňuje vyvýjet různé projekty najednou
  - determinističnost
  - složitější prostředí

Verze GHC

Standard pro IB016 je GHC verze alespoň 7.10!

Verze GHC

Standard pro IB016 je GHC verze alespoň 7.10!

Významné změny v GHC 7.10 oproti 7.8:

- využití typové třídy `Foldable` v `Prelude`
  - zobecnění konceptu seznamů, přes které jde „foldovat“ na podobné datové struktury (pole, množiny, ...)
  - `length :: Foldable t => t a -> Int`
  - `foldr :: Foldable t => (a->b->b) -> b -> t a -> b`
- změny týkající se typové třídy `Applicative`
  - nyní přímo v `Prelude`
  - nově je nadtřídou typové třídy `Monad` (viz později)
- Důsledek:
  Pro funkčnost kódu z 7.10 v 7.8 je často třeba doplnit importy (`Control.Applicative`, `Data.Foldable`)!
- nymfe mají GHC 7.10.3, případně 8.0.2 v modulu (funguje i na aise)

Opakování: funkce vyšších řádů, lambda funkce, ...

```
import Data.Maybe

lookBy :: (a -> Maybe b) -> [a] -> Maybe b
lookBy _   []     = Nothing
lookBy lfn (x:xs) = let mv = lfn x
                    in if isJust mv
                          then mv
                          else lookBy lfn xs

look' :: Eq a => a -> [(a, b)] -> Maybe b
look' x = lookBy (\ (k, v) ->
            if k == x then Just v else Nothing)
```

Pokročilejší syntax: case

Používání vzorů uvnitř funkce

- stejně jako u vzorů funkcí se prochází odshora

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

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 syntax: stráže (guards)

Definice pomocí alternativ

- použije se tělo u první podmínky, která se vyhodnotí na `True`
- v `Prelude`: `otherwise = True`

```
look :: Eq a => [(a,b)] -> a -> Maybe b
look []           _ = Nothing
look ((k,v) : s) x
    | k == x    = Just v
    | otherwise = look s x
```

Pokročilejší syntax: stráže (guards)

```
data BinTree a = BNode a (BinTree a) (BinTree a)
               | BEmpty
               deriving (Eq, Show, Read)

lookSTree :: Ord k => k -> BinTree (k, v) -> Maybe v
lookSTree _ BEmpty = Nothing
lookSTree x (BNode (k, v) l r)
    | k == x    = Just v
    | x < k     = lookSTree x l
    | otherwise = lookSTree x r

testlst n = lookSTree n (BNode (10, "a")
                (BNode (1, "b") BEmpty BEmpty)
                (BNode (100, "c")
                    (BNode (42, "?") BEmpty BEmpty)
                    BEmpty))
```

Pokročilejší syntax: $

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

Operátor aplikace funkce

- aplikace funce s nízkou prioritou a asociativitou zprava
  - `f $ g $ x` ≡ `f (g x)`
  - `f . g $ h x` ≡ `(f . g) (h x)`
- mezeru lze vnímat jako operátor aplikace s nejvyšší prioritou a asociativitou zleva
  - `f g x` ≡ `(f g) x`

Pokročilejší syntax: $

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

Operátor aplikace funkce

- aplikace funce s nízkou prioritou a asociativitou zprava
  - `f $ g $ x` ≡ `f (g x)`
  - `f . g $ h x` ≡ `(f . g) (h x)`
- mezeru lze vnímat jako operátor aplikace s nejvyšší prioritou a asociativitou zleva
  - `f g x` ≡ `(f g) x`

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

Moduly a balíky

Moduly (`Data.Char`, `Data.Vector.Mutable`, ...)

- skupina funkcí ve stejném souboru a „jmenném prostoru“
- jméno modulu vždy velkými písmeny, hierarchie (tečky)
- základní modul `Prelude`, další načítáme pomocí `import`
- hlavička `module ModuleName (fn1, fn2, fn3) where`

Moduly a balíky

Moduly (`Data.Char`, `Data.Vector.Mutable`, ...)

- skupina funkcí ve stejném souboru a „jmenném prostoru“
- jméno modulu vždy velkými písmeny, hierarchie (tečky)
- základní modul `Prelude`, další načítáme pomocí `import`
- hlavička `module ModuleName (fn1, fn2, fn3) where`

Balíky (`containers-0.5.10.1`, `brainfuck-0.1.0.3`, ...)

- skupina modulů, která se instaluje spolu
- název většinou malými písmeny, nemá hierarchii
- balík `base` (základní knihovna modulů)
- spravuje typicky balíčkovací systém `cabal` (viz později)

Moduly v praxi

Informace o modulech/balících:

- databáze balíků Hackage
- vyhledávač Hayoo (hledání podle funkcí, typů, balíků, . . .)

Moduly v praxi

Informace o modulech/balících:

- databáze balíků Hackage
- vyhledávač Hayoo (hledání podle funkcí, typů, balíků, . . .)

Použití při programování

- nainstalovat balík:
  - `cabal update && cabal install <package>`
  - komplikovanější, detaily příště
- import v kódu: `import <module>`
- přidání modulu v GHCi: `:m + <module>`

Dokumentace v Haskellu: Haddock

```haskell
-- | The 'square' function squares an integer.
square :: Int -> Int
square x = x * x

data T a b
    = C1 a b  -- ^ info about constructor 'C1'
    | C2 a b  -- ^ info about constructor 'C2'
```

- syntax komentářů pro automatické zpracování
- generování HTML dokumentace programem `haddock`
  - `mkdir -p doc && haddock file.hs --html -o doc`
- více info viz oficiální dokumentace

Testování pomocí balíku HUnit

Základní v Haskellu obecně:

- mnoho příspupů i balíků
- dnes `HUnit`, ideovo vycházející z `JUnit`
- v průběhu semestru ještě `QuickCheck`

Testování pomocí balíku HUnit

Základní v Haskellu obecně:

- mnoho příspupů i balíků
- dnes `HUnit`, ideovo vycházející z `JUnit`
- v průběhu semestru ještě `QuickCheck`

Balík `HUnit` ve skratce:

- jednoduché unit testování (hierarchie testů)
- v balíku `HUnit`
  instalace: `cabal update && cabal install hunit`
- kompletní dokumentace na Hackage

HUnit: základní syntax

Operátory pro konstrukci testů:

- `(~?) :: (...) => t -> String -> Test`
- `(~=?) :: (...) => a -> a -> Test`
- `(~?=) :: (...) => a -> a -> Test`

HUnit: základní syntax

Operátory pro konstrukci testů:

- `(~?) :: (...) => t -> String -> Test`
- `(~=?) :: (...) => a -> a -> Test`
- `(~?=) :: (...) => a -> a -> Test`

Hierarchie a pojmenovávání testů:

```
data Test = TestCase Assertion
          | TestList [Test]
          | TestLabel String Test
```

HUnit: základní syntax

Operátory pro konstrukci testů:

- `(~?) :: (...) => t -> String -> Test`
- `(~=?) :: (...) => a -> a -> Test`
- `(~?=) :: (...) => a -> a -> Test`

Hierarchie a pojmenovávání testů:

```
data Test = TestCase Assertion
          | TestList [Test]
          | TestLabel String Test
```

Spuštění testů:

- `runTestTT :: Test -> IO Counts`

HUnit: ukázka

```
module HUnitExample (fact, runTests) where
import Test.HUnit

fact :: Integer -> Integer
fact n = product [1..n]

runTests :: IO Counts
runTests = runTestTT $ TestList [testSet1, testSet2]

testSet1 :: Test
testSet1 = TestLabel "Factorials" $
  TestList [ fact 4 ~?= 25, fact 0 ~?= 1 ]

testSet2 :: Test
testSet2 = TestLabel "Numerical functions" $
  TestList [ even 4 ~? "4 even?", odd 4 ~? "4 odd?" ]
```

Samostatný úkol

Dle vlastního uvážení si vyberte z následujícího:

- práce se zlomky: **6.1.8** se zbírky IB015
- dokumentovat, vygenerovat HTML dokumentaci
- napsat si pěkné testy pomocí HUnit
- Hanoiské věže: **8.3.1** se zbírky IB015
  (opět nezapomenout na dokumentaci a testy)