Práce se soubory, zpracování chyb a výjimek

IB016 Seminář z funkcionálního programování

Fakulta informatiky, Masarykova univerzita

Práce se soubory

```haskell
main = do
       putStrLn "Enter filename:"
       filename <- getLine
       content  <- readFile filename
       putStrLn $ filename ++ " capitalized:"
       putStrLn $ map toUpper content
```

- **FilePath** je jen typový alias pro **String**
- základní funkce pro práci se soubory:

```haskell
readFile   :: FilePath -> IO String
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

Pokročilejší práce se soubory

Modul `System.IO`: multiplatformní práce se soubory

- `type FilePath = String`
- `data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode`
- základní funkce `readFile`, `writeFile`, `appendFile`
- datový typ `Handle` pro práci s proudy
  - `stdin`, `stdout`, `stderr :: Handle`
  - `openFile :: FilePath -> IOMode -> IO Handle`
  - `hClose :: Handle -> IO ()`
  - `hGetContents :: Handle -> IO String`
  - `hPutStr :: Handle -> String -> IO ()`
  - `withFile ::`
    `FilePath -> IOMode -> (Handle -> IO r) -> IO r`
    pro automatické uzavření handle po dokončení IO operace
  - a další, vizte Hoogle/dokumentaci

Práce se soubory a lenost

Všechny z funkcí `readFile`, `getContents`, `hGetContents` jsou líné! Ze souboru se čte, až když je obsah skutečně potřeba.

Existují i striktní varianty `readFile'`, `getContents'`, `hGetContents'`.

Kdy byste preferovali kterou variantu? Jaké má která výhody a nevýhody?

Utility pro práci s cestami a jmény souborů

Modul `System.FilePath`: utility pro práci s cestami a jmény souborů

- platformně nezávislé

Utility pro práci s cestami a jmény souborů

Modul **System.FilePath**: utility pro práci s cestami a jmény souborů

- platformně nezávislé
- `"some_dir" </> "some_file" <.> "ext"`

Utility pro práci s cestami a jmény souborů

Modul **System.FilePath**: utility pro práci s cestami a jmény souborů

- platformně nezávislé
- `"some_dir" </> "some_file" <.> "ext"`
- `isAbsolute "/foo"` $\rightsquigarrow^*$ **`True`**, `isRelative "bar"` $\rightsquigarrow^*$ **`True`**

Utility pro práci s cestami a jmény souborů

Modul **System.FilePath**: utility pro práci s cestami a jmény souborů

- platformně nezávislé
- `"some_dir" </> "some_file" <.> "ext"`
- `isAbsolute "/foo"` $\rightsquigarrow^*$ **`True`**, `isRelative "bar"` $\rightsquigarrow^*$ **`True`**
- `takeExtension "a.out"` $\rightsquigarrow^*$ `".out"`

Utility pro práci s cestami a jmény souborů

Modul **`System.FilePath`**: utility pro práci s cestami a jmény souborů

- platformně nezávislé
- `"some_dir" </> "some_file" <.> "ext"`
- `isAbsolute "/foo"` $\rightsquigarrow^*$ **`True`**, `isRelative "bar"` $\rightsquigarrow^*$ **`True`**
- `takeExtension "a.out"` $\rightsquigarrow^*$ `".out"`
- a spousta dalších
- importujeme **`System.FilePath`**, ten podle systému importuje buď POSIX nebo Windows variantu
- výrazně vhodnější než pracovat s cestami ručně

Práce s adresářovou strukturou

Modul `System.Directory`: multiplatformní manipulace s adresářovou strukturou, kopírování a odstraňování souborů, základní práce s právy.

Práce s adresářovou strukturou

Modul **System.Directory**: multiplatformní manipulace s adresářovou strukturou, kopírování a odstraňování souborů, základní práce s právy.

- `getCurrentDirectory`, `listDirectory`
- `doesFileExist "/etc/passwd"` $\rightsquigarrow^*$ **`True`**
- `doesDirectoryExist "/etc"` $\rightsquigarrow^*$ **`True`**
- `getTemporaryDirectory` $\rightsquigarrow^*$ `"/tmp"`
- `getPermissions`, `setPermissions` (jen základní práva, POSIX práva v POSIX-specifických modulech)
- `executable <$> getPermissions "/bin/ls"` $\rightsquigarrow^*$ **`True`**
- `getFileSize`
- `findFile`
- ...

Všechny předchozí funkce vrací **IO** akce.

Kombinování akcí IO, utility

Modul **Control.Monad**; funguje nejen pro **IO**, ale pro všechny monády.

- `mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]`
  `filterM :: Applicative m => (a -> m Bool) -> [a] -> m [a]`

  obdoba běžných funkcí, volané funkce jsou akce IO
  - `forM`: jako `mapM`, převrácené argumenty
  - varianty `mapM_`, `forM_` ignorují výsledek

Kombinování akcí IO, utility

Modul **Control.Monad**; funguje nejen pro **IO**, ale pro všechny monády.

- `mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]`
  `filterM :: Applicative m => (a -> m Bool) -> [a] -> m [a]`

  obdoba běžných funkcí, volané funkce jsou akce IO
  - `forM`: jako `mapM`, převrácené argumenty
  - varianty `mapM_`, `forM_` ignorují výsledek
- `sequence :: (Monad m) => [m a] -> m [a]`

  spuštění seznamu akcí IO, obdobně `sequence_`

Kombinování akcí IO, utility

Modul **Control.Monad**; funguje nejen pro **IO**, ale pro všechny monády.

- `mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]`
  `filterM :: Applicative m => (a -> m Bool) -> [a] -> m [a]`

  obdoba běžných funkcí, volané funkce jsou akce IO
  - `forM`: jako `mapM`, převrácené argumenty
  - varianty `mapM_`, `forM_` ignorují výsledek
- `sequence :: (Monad m) => [m a] -> m [a]`

  spuštění seznamu akcí IO, obdobně `sequence_`
- `void :: Functor f => f a -> f ()`
- a další, vizte dokumentaci

Lze používat i aplikativní kombinátory `<*>`, `*>`, `<*`, , `<$>`, atd. z minula.

Užitečné funkce pro interakci a argumenty příkazové řádky

Modul `System.Environment` poskytuje funkce pro práci s argumenty příkazové řádky a proměnnými prostředí.

- `getArgs :: IO [String]`
- `getProgName :: IO String`
- `getEnvironment :: IO [(String, String)]` a další

Užitečné funkce pro interakci a argumenty příkazové řádky

Modul `System.Environment` poskytuje funkce pro práci s argumenty příkazové řádky a proměnnými prostředí.

- `getArgs :: IO [String]`
- `getProgName :: IO String`
- `getEnvironment :: IO [(String, String)]` a další

Při interakci (nejen) s uživatelem

- `when :: Applicative f => Bool -> f () -> f ()`
- `forever :: Applicative f => f a -> f b`
- `interact :: (String -> String) -> IO ()`

Pozor na buffering! Více v dokumentaci.

Řetězce v Haskellu

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

- výchozí, jednoduché, automaticky funguje se všemi seznamovými funkcemi
- pomalé a neefektivní: cca 4.4 GB na načtení 100MB souboru do paměti
- vhodné pro líné zpracování, vhodné pro krátké řetězce

Řetězce v Haskellu

```
type String = [Char]
```

- výchozí, jednoduché, automaticky funguje se všemi seznamovými funkcemi
- pomalé a neefektivní: cca 4.4 GB na načtení 100MB souboru do paměti
- vhodné pro líné zpracování, vhodné pro krátké řetězce

`Text` (`Data.Text`, balík *text*)

- pro unicode text, interně používá UTF-16 (2 B / znak)
- dostupné ve striktní i líné variantě
- cca 5.6 × více paměti pro načtení souvislého bloku

Řetězce v Haskellu

```
type String = [Char]
```

- výchozí, jednoduché, automaticky funguje se všemi seznamovými funkcemi
- pomalé a neefektivní: cca 4.4 GB na načtení 100MB souboru do paměti
- vhodné pro líné zpracování, vhodné pro krátké řetězce

`Text` (**`Data.Text`**, balík *text*)

- pro unicode text, interně používá UTF-16 (2 B / znak)
- dostupné ve striktní i líné variantě
- cca 5.6 × více paměti pro načtení souvislého bloku

`ByteString` (**`Data.ByteString`**, balík *bytestring*)

- především pro binární data; síťová komunikace, volání do C
- striktní a líná varianta
- cca 2 × více paměti pro načtení souvislého bloku

Text

```
import Data.Text import qualified Data.Text as T
```

- definuje mnoho funkcí stejného jména jako **Prelude**
- líná varianta v **Data.Text.Lazy**
- lze využít optimalizací – operace nemusí vždy vytvářet mezilehlé hodnoty (*fusion*)
- užitečné funkce
  - `pack :: String -> Text`
  - `unpack :: Text -> String`
  - `append :: Text -> Text -> Text`
  - `cons`, `snoc`, ...

IO v Text

```haskell
import qualified Data.Text.IO as T
```

- závislé na nastavení locale (v systému, nebo v GHC)
- `readFile :: FilePath -> IO Text`
- `appendFile`, `writeFile`, `getLine`, `putStr`...
- funguje i práce s `Handle`
- existuje i líná varianta `Data.Text.Lazy.IO`

ByteString

```
import Data.ByteString ( ByteString )
import qualified Data.ByteString as BS
```

- kompaktní reprezentace binárních dat (resp. 8bitových znaků)
- striktní verze (v podstatě C-like *char*$*$ + hlavička)
- líná verze (seznam striktních bloků)

ByteString

```
import Data.ByteString ( ByteString )
import qualified Data.ByteString as BS
```

- kompaktní reprezentace binárních dat (resp. 8bitových znaků)
- striktní verze (v podstatě C-like *char∗* + hlavička)
- líná verze (seznam striktních bloků)

- binární `Data.ByteString` nebo
  znakový `Data.ByteString.Char8` pohled na data
  - `pack :: [Word8] -> ByteString`
  - `pack :: String -> ByteString`
  - stejný typ, jiná metoda přístupu
  - **znaková verze bere jen spodních 8 bitů z `Char`**

ByteString

```
import Data.ByteString ( ByteString )
import qualified Data.ByteString as BS
```

- kompaktní reprezentace binárních dat (resp. 8bitových znaků)
- striktní verze (v podstatě C-like *char∗* + hlavička)
- líná verze (seznam striktních bloků)

- binární `Data.ByteString` nebo
  znakový `Data.ByteString.Char8` pohled na data
  - `pack :: [Word8] -> ByteString`
  - `pack :: String -> ByteString`
  - stejný typ, jiná metoda přístupu
  - **znaková verze bere jen spodních 8 bitů z `Char`**
- opět (re)definuje vlastní `IO` funkce, podporuje *fusion*

Konverze mezi Text a ByteString

- například binární data obsahující UTF-8 znaky
- `decodeUtf8 :: ByteString -> Text`
  - může vyhodit výjimku pokud vstup není validní UTF-8
- `decodeUtf8'` vrací `Either UnicodeException Text`
- i varianty pro UTF-16, UTF-32 v big endian/little endian
- `encodeUtf8 :: Text -> ByteString`

Vstup a výstup

- standardní IO (`System.IO`) má mnoho problémů
  - někdy neintuitivní líné vyhodnocování
  - závislost na nastaveném locale při práci se soubory
  - vyhazuje výjimky při nevalidním kódování (např. neplatný UTF-8 znak)
- typický bývá nejvhodnější použít striktní `ByteString` a dekódování do `Text`
- více třeba na `www.snoyman.com/blog/2016/12/beware-of-readfile`

Výjimky

Řešení chyb v čistém kódu

- `Maybe`, `Either`

1

Řešení chyb v čistém kódu

- **Maybe**, **Either**
- využití toho, že obojí jsou funktory/monády
  - `fmap (+ 2) (lookup "a" [...])`
  - `lookup "a" [...] >>= foo`

1

Řešení chyb v čistém kódu

- **Maybe**, **Either**
- využití toho, že obojí jsou funktory/monády
  - `fmap (+ 2) (lookup "a" [...])`
  - `lookup "a" [...] >>= foo`
- `maybe`[1] `:: b -> (a -> b) -> Maybe a -> b`
  `fromMaybe :: a -> Maybe a -> a`
  (z **Data.Maybe**)

[1] Povšimněte si, že jde o katamorfismus

Řešení chyb v čistém kódu

- **Maybe**, **Either**
- využití toho, že obojí jsou funktory/monády
  - `fmap (+ 2) (lookup "a" [...])`
  - `lookup "a" [...] >>= foo`
- `maybe`[1] `:: b -> (a -> b) -> Maybe a -> b`
  `fromMaybe :: a -> Maybe a -> a`
  (z `Data.Maybe`)
- `either`[1] `:: (a -> c) -> (b -> c) -> Either a b -> c`

[1] Povšimněte si, že jde o katamorfismy

Chyby v IO kódu

Věci, kde je selhání běžné: **Maybe**, **Either**

- `findExecutable :: String -> IO (Maybe FilePath)`
- zabalení v **IO** komplikuje práci

Chyby v IO kódu

Věci, kde je selhání běžné: **`Maybe`**, **`Either`**

- `findExecutable :: String -> IO (Maybe FilePath)`
- zabalení v **`IO`** komplikuje práci

Věci, kde je selhání neočekávané/závažná chyba: **výjimky**

- podobně jako v imperativních jazycích (C++, C#, Java) používáme pro výjimečné případy
- pokud nechceme chybu ošetřovat hned, jak nastane
- o něco komplikovanější než v imperativních jazycích
- typicky neočekávaný stav souborového systému, nedostatečná práva,...

Výjimky v Haskellu I

- lze chytat jen v `IO`
- do funkcionálního paradigmatu se nehodí
- modul `Control.Exception` pro obecnou práci s výjimkami
- modul `System.IO.Error` obsahuje další funkce pro práci s výjimkami z `IO` operací

Výjimky v Haskellu I

- lze chytat jen v **`IO`**
- do funkcionálního paradigmatu se nehodí
- modul **`Control.Exception`** pro obecnou práci s výjimkami
- modul **`System.IO.Error`** obsahuje další funkce pro práci s výjimkami z **`IO`** operací
- typová třída **`Exception`**, vyžaduje **`Show`**
  - každý typ výjimky je instancí

Výjimky v Haskellu I

- lze chytat jen v `IO`
- do funkcionálního paradigmatu se nehodí
- modul `Control.Exception` pro obecnou práci s výjimkami
- modul `System.IO.Error` obsahuje další funkce pro práci s výjimkami z `IO` operací
- typová třída `Exception`, vyžaduje `Show`
  - každý typ výjimky je instancí
- při zachytávání nutné, aby byl jednoznačně určen typ výjimky
  - aby se odvodilo, zda má být chycena
  - často vyžaduje explicitní otypování
  - `SomeException` pro libovolnou

Výjimky v Haskellu I

- lze chytat jen v **IO**
- do funkcionálního paradigmatu se nehodí
- modul **Control.Exception** pro obecnou práci s výjimkami
- modul **System.IO.Error** obsahuje další funkce pro práci s výjimkami z **IO** operací
- typová třída **Exception**, vyžaduje **Show**
  - každý typ výjimky je instancí
- při zachytávání nutné, aby byl jednoznačně určen typ výjimky
  - aby se odvodilo, zda má být chycena
  - často vyžaduje explicitní otypování
  - **SomeException** pro libovolnou
- `catch :: Exception e => IO a -> (e -> IO a) -> IO a`

Výjimky v Haskellu I

- lze chytat jen v **IO**
- do funkcionálního paradigmatu se nehodí
- modul **Control.Exception** pro obecnou práci s výjimkami
- modul **System.IO.Error** obsahuje další funkce pro práci s výjimkami z **IO** operací
- typová třída **Exception**, vyžaduje **Show**
  - každý typ výjimky je instancí
- při zachytávání nutné, aby byl jednoznačně určen typ výjimky
  - aby se odvodilo, zda má být chycena
  - často vyžaduje explicitní otypování
  - **SomeException** pro libovolnou
- `catch :: Exception e => IO a -> (e -> IO a) -> IO a`
  Použití:

  ```
  catch expr (\ex -> print (ex :: IOException) >> handleIOExc)
  ```
- `try :: Exception e => IO a -> IO (Either e a)`

Výjimky v Haskellu II

- správa zdrojů – `bracket`:

```
bracket :: IO a         -- ^ získání zdroje
        -> (a -> IO b) -- ^ uvolnění zdroje
        -> (a -> IO c) -- ^ operace se zdrojem
        -> IO c

withFile name mode = bracket (openFile name mode) hClose
```

- `throwIO :: Exception e => e -> IO a`
- vyhazování výjimek je v čistém kódu možné, ale silně nevhodné

Výjimky v Haskellu III

Interakce mezi výjimkami a leností je problém

```
λ> catch (pure (4 `div` 0)) (\ex -> print (ex :: SomeException) >> pure 0)
```

Výjimky v Haskellu III

Interakce mezi výjimkami a leností je problém

```
λ> catch (pure (4 `div` 0)) (\ex -> print (ex :: SomeException) >> pure 0)
*** Exception: divide by zero
```

- výjimka nechycena!

Výjimky v Haskellu III

Interakce mezi výjimkami a leností je problém

```
λ> catch (pure (4 `div` 0)) (\ex -> print (ex :: SomeException) >> pure 0)
*** Exception: divide by zero
```

- výjimka nechycena!
- `pure` vrací nevyhodnocenou věc, až při vyhodnocení se vyhodí výjimka
- `catch` dříve než vyhodnocení
- pokud výjimku vyhazuje **IO**-funkce, problém typicky není

Výjimky v Haskellu IV

Při špatné interakci s leností je třeba vynutit vyhodnocení

- `evaluate :: a -> IO a` vyhodnotí po vnější datový konstruktor
- vrací **IO**-akci, která vyhodí výjimku, pokud vyhodnocení vyhodilo výjimku

```
λ> catch (evaluate (4 `div` 0)) (\ex -> print (ex :: SomeException) >> pure 0)
divide by zero
```

Výjimky v Haskellu IV

Při špatné interakci s leností je třeba vynutit vyhodnocení

- `evaluate :: a -> IO a` vyhodnotí po vnější datový konstruktor
- vrací **IO**-akci, která vyhodí výjimku, pokud vyhodnocení vyhodilo výjimku

```
λ> catch (evaluate (4 `div` 0)) (\ex -> print (ex :: SomeException) >> pure 0)
divide by zero
```

POZOR!

```
λ> catch (evaluate [4 `div` 0]) (\ex -> print (ex :: SomeException) >> pure [])
[*** Exception: divide by zero
```

Výjimky v Haskellu IV

Při špatné interakci s leností je třeba vynutit vyhodnocení

- `evaluate :: a -> IO a` vyhodnotí po vnější datový konstruktor
- vrací **IO**-akci, která vyhodí výjimku, pokud vyhodnocení vyhodilo výjimku

```
λ> catch (evaluate (4 `div` 0)) (\ex -> print (ex :: SomeException) >> pure 0)
divide by zero
```

POZOR!

```
λ> catch (evaluate [4 `div` 0]) (\ex -> print (ex :: SomeException) >> pure [])
[*** Exception: divide by zero
```

- úplné vyhodnocení pomocí `Control.DeepSeq.force` (`evaluate (force x)`)

Výjimky v Haskellu: příklad

```haskell
import Control.Exception
import Control.Monad
import System.Environment
import System.IO

main = handle ioExpHandler $ do
    [from, to] <- getArgs
    withFile from ReadMode $ \hFrom ->
        withFile to WriteMode $ \hTo ->
            until (hIsEOF hFrom) $ do
                line <- hGetLine hFrom
                hPutStrLn hTo line
  where
    ioExpHandler :: IOException -> IO ()
    ioExpHandler e = putStrLn $ "fatal: " ++ show e
    until :: IO Bool -> IO a -> IO ()
    until cond act = cond >>= \x -> when x (act >> until cond act)
```

Samostatné programování

Hledání spustitelných souborů

Napište funkci `isExecutable :: FilePath -> IO Bool`, která o zadané cestě rozhodne, jestli na ní je spustitelný soubor.

Poté napište program, který jako argument příkazové řádky dostane cestu k adresáři a vypíše na standardní výstup cesty ke všem spustitelným souborům, které se nachází v zadaném adresáři. Nejprve zkuste variantu, která hledá spustitelné soubory jen v zadaném adresáři. Poté zkuste variantu, která rekurzivně prochází i všechny podadresáře.

Nemusíte ošetřovat výjimky.

Pokud chcete mít krátký a elegantní kód, mohly by se vám hodit funkce `mapM`, `mapM_` nebo `filterM`.

Velikost adresáře

Napište program, který jako argument příkazové řádky dostane cestu k adresáři a vypíše na standardní výstup součet velikostí všech souborů, které tento adresář obsahuje.

Opět nejprve zkuste variantu, která počítá velikost jen ze souborů, které se nachází přímo v zadaném adresáři. Poté zkuste variantu, která rekurzivně prochází i všechny podadresáře.

Nemusíte ošetřovat výjimky.

Mohly by se vám hodit funkce `mapM` nebo `foldM`.

Vyhledávání textu

Napište program, který dostane dva argumenty příkazové řádky: cestu k adresáři a slovo. Program vypíše na standardní výstup cesty všech souborů ze zadaného adresáře (i rekurzivně v podadresářích), které obsahují zadané slovo.

Např:

```
> ./wordFinder adresar espresso
adresar/kavarnyVBrne.txt
adresar/knihy/JamesHoffmann_WorldAtlasOfCoffee.txt
```

Nemusíte ošetřovat výjimky.

Výjimky: počítání slov

Napište program, který dostane jako argumenty cesty k libovolně mnoha sobourům a vypíše na standardní výstup součet počtu slov, která tyto soubory obsahují.

Např:

```
> ./wordCounter soubor1.txt soubor2.txt soubor3.txt
193
```

Ošetřete výjimky, kdy se některý soubor nepodařilo otevřít, vhodnou chybovou hláškou. I když se některý soubor nepodaří otevřít, program by stále měl spočítat slova ze všech ostatních souborů.