# Monadické parsování (Parsec) a nástroje

## IB016 Seminář z funkcionálního programování

Adam Matoušek, Henrieta Micheľová

(původní autoři slajdů Vladimír Štill, Martin Ukrop)

Fakulta informatiky, Masarykova univerzita

Jaro 2021

# HLint

Nástroj hlásící návrhy na zlepšení kódu („linter“)

- samostatný balík z Hackage, nutno doinstalovat
  - často dostupný přímo v repozitářích distribuce
  - FI: na nymfách nainstalovaný z repozitářů Ubuntu
  - více podrobností v samostatném návodu v ISu
- `aisa$ cabal new-install hlint`
- `hlint [--hint` *`soubor_definic_navíc`*`]` *`zdroják`*
- možno integrovat přímo do GHCi, vizte návod v ISu
- soubor s dodatečnými definicemi v ISu

# 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'
```

- syntaxe komentářů pro automatické zpracování
- generování HTML dokumentace programem `haddock`
  - `haddock file.hs --html -o doc`
- na FI: součástí modulu `ghc`
- více informací v oficiální dokumentaci

# Parsování: Regulární výrazy

Haskell má několik balíků pro práci s regulárními výrazy:

- většina se nachází v modulech `Text.Regex.*`
- „základní“ posixová implementace v balíku `regex-posix`
- pěkný přehled možností různých balíku najdete třeba na `https://wiki.haskell.org/Regular_expressions`

Vesměs stejný princip jako u jiných jazyků.

# Monadické parsování - Parsec

# Syntaktické analyzátory (parsery)

Základní regulární výrazy často nestačí.
→ využijeme syntaktické analyzátory (parsery)

- lexikální analyzátor **Alex** + syntaktický analyzátor **Happy** (podobné kombinaci *lex*/*flex* + *bison*/*yacc*)
- **Parsec** – knihovna založena na parserových kombinátorech, zvládá i tvorbu lexikálních analyzátorů, přívětivější chybové hlášky
- **Attoparsec** – další kombinátorová knihovna, rychlejší než Parsec – hlavně pro síťové protokoly a komplikované textové/binární formáty, chybí *Optimus Prime*
- A mnoho dalších – Polyparse, Megaparsec, GLL...

# Parsec

Myšlenka:

- definujme parser pomocí většího množství menších/jednodušších parserů

Technicky:

- balík `parsec` (není součástí standardní distribuce)
- nejpoužívanější funkce přímo v modulu `Text.Parsec`
- pro naše účely hlavně funkce z modulů `Text.Parsec.Char` a `Text.Parsec.Combinators`
- pro zjednodušení typování importujte i modul `Text.Parsec.String`

# Motivace

Chceme parsovat jednotlivé díly seriálu:

*The–Witcher S01E05: Bottled Appetites*

- Jak by mohl vypadat typ po parser?
  (Čím by mohl být parametrizován?)
- Jaký typ by byl vhodný pro „parsovací funkci“?

# Datový typ Parser

```
myParser :: Parser a
```

- parser, který zpracuje vstup na hodnotu typu `a`
- instance pro třídy `Functor`, `Applicative` i `Monad`
- ve skutečnosti je to pouze typové synonymum pro obecnější parser, viz později

# Aplikace parserů

Na aplikaci parserů používáme specializované funkce:[1]

```
parse :: Parser a -> SourceName -> String
      -> Either ParseError a
```

- `parse p name input` spustí parser `p` na vstupu `input`, `name` se bude používat pouze na identifikaci v chybových hláškách
- výsledkem je buď zparsovaná hodnota nebo chyba typu `ParseError`

```
parseFromFile :: Parser a -> String
              -> IO (Either ParseError a)
```

- `parseFromFile p f` spustí parser `p` na obsahu souboru `f`

[1] typ je ve skutečnosti obecnější, viz později

# Základní znakový parser

```
satisfy :: (Char -> Bool) -> Parser Char
```

- parser `satisfy p` uspěje, jestliže načtený znak splňuje zadaný predikát (pak vrací tento znak)

Existující užitečné parsery:

```
char :: Char -> Parser Char
digit, letter, anyChar, space :: Parser Char
oneOf, noneOf :: [Char] -> Parser Char
```

Příklad: *The–Witcher S01E05: Bottled Appetites*

- Které znakové parsery použít?
- Jak napsat znakové parsery pomocí `satisfy`?

# Sekvence parserů

Zamyšlení: Jak budou fungovat instance `Functor`, `Monad`?

- `fmap :: (a -> b) -> Parser a -> Parser b`
  `fmap (: "!") (char 'A')`
  `fmap` aplikuje funkci na vrácenou hodnotu
- `pure :: a -> Parser a`
  `pure 42`
  `pure x` nic nečte a vrátí hodnotu `x`
- ```
  twoChars = do
      char1 <- anyChar
      char2 <- anyChar
      pure [char1, char2]
  ```
  operátor `>>=` předá zparsovanou hodnotu zleva doprava
  ```

# Sekvence parserů

Existující užitečné parsery:

```
string :: String -> Parser String
between :: Parser open -> Parser close
        -> Parser a -> Parser a
count :: Int -> Parser a -> Parser [a]
many, many1 :: Parser a -> Parser [a]
sepBy, sepBy1 :: Parser a -> Parser sep -> Parser [a]
```

- Kolik vstupu spracuje `many`?
  `parse (many digit) "input name" "hello"`
- Jak vyparsovat užitečné věci z názvu seriálu?
  *The–Witcher S01E05: Bottled Appetites*

# Příklad kombinace parserů v do-bloku

Příklad: *The–Witcher S01E05: Bottled Appetites*

```
series :: Parser (String, Int, Int, String)
series = do
  series <- many (noneOf " ")
  spaces
  string "S"
  season <- many1 digit
  string "E"
  episode <- many1 digit
  string ":"
  spaces
  epName <- many anyChar
  pure (series, read season, read episode, epName)
```

# Alternativa parserů

```
<|> :: Parser a -> Parser a -> Parser a
```

- `p <|> q` se pokusí nejdříve použít parser `p` a jestli uspěje, vrátí jeho výsledek
- jestli `p` selže bez toho, aby spotřeboval nějaký vstup, použije se parser `q`
- Příklad: `string "spring" <|> string "autumn"`

Existující užitečné parsery:

```
option :: a -> Parser a -> Parser a
optionMaybe :: Parser a -> Parser (Maybe a)
optional :: Parser a -> Parser ()
```

# Alternativa parserů

```
season = string "spring" <|> string "summer"
```

`season` na řetězci `"summer"` selže – levá alternativa sice selhala, ale spotřebovala vstup!

```
try :: Parser a -> Parser a
```

- `try p` se chová stejně jako parser `p`, ale když `p` selže, vrátí se ve vstupu, jako by žádný nebyl parserem `p` spotřebován
- Pozor: používání `try` zvyšuje složitost parsování! (Ale na druhou stranu nám umožňuje mít libovolný *lookahead*.)

# Další užitečné operátory

- `>>`
- `liftAX`
- `<$>`
- `<$`
- `<*>`
- `<*`
- `*>`

```
spaces >> identifier
liftA2 (+) number number
(++ "!") <$> greeting
"greeting here" <$ greeting
(,) <$> key <*> value
string "Hello" <* spaces <* name
string "Hello" *> spaces *> name
```

# Užitečné chybové hlášky

Nevýhoda kombinátorových parserů: Ladění není snadné.
(Který parser selhal?)

Zlepšení: Doplňme do parseru popis, co dělá:

```
"your input:" (line 1, column 1):
unexpected "L"
expecting digit
```

```
(<?>) :: Parser a -> String -> Parser a
```

- mění chybovou hlášku („expected“)
- obzvlášť vhodné pro alternativy nebo `try`
- parser nesmí spotřebovat vstup

# Komplexní příklad

Příklad: *The–Witcher S01E05: Bottled Appetites*

```haskell
data TVSeries = Series String Episode String deriving Show
data Episode  = Ep Int Int deriving Show -- Season, No.

episode :: Parser Episode
episode = liftM2 Ep
  ((string "s" <|> string "S") *> fmap read (many1 digit))
  ((string "e" <|> string "E") *> fmap read (many1 digit))
    <?> "Season/Episode spec"

series :: Parser TVSeries
series = liftM3 Series
  (many (noneOf " ") <* spaces)
  (episode <* string ":" <* spaces)
  (many anyChar)
    <?> "series spec"
```

# Ve skutečnosti je to obecnější...

Typ `Parser a` je pouze speciální případ parseru:

```
type Parser a = Parsec String () a
type Parsec s u a = ParsecT s u Identity a
```

`ParsecT s u m a` představuje parser, který

- zpracovává typ `s`
- udržuje si stav typu `u`
- pracuje v monádě `m`
- vrací výsledek typu `a`

Typ funkce `parse` je pak následovný:

```
runParser :: Stream s Identity t =>
    Parsec s u a -> u -> SourceName -> s
    -> Either ParseError a
```

# Příklady k procvičení

Napište parser pro:

- datum – `"2015-04-14"`
  rozpoznaný řetězec číslic můžete konvertovat funkcí `read`, výsledek ať je vámi definovaného typu `Date`, rozsah kontrolovat nemusíte
- čísla která můžou mít desetinnou část – `"12"`, `"12.54"`
- seznam čísel – `"[ 12, 12.54, 42, 66, 3.14 ]"`
  využijte parser z předchozího cvičení
- výraz s přirozenými čísly a operacemi – `(2+5)*2`
  precedenci operátorů neřešte, parser ať výraz rovnou vyhodnotí, tj. ať je typu `Parser Int`