Monadické parsování (Parsec)

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2018

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.

Syntaktické analyzátory (parsery)

Základní regulární výrazy často nestačí.
$\rightarrow$ 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
- **Attoparsec** – další kombinátorová knihovna, hlavně pro síťové protokoly a komplikované textové/binární formáty
- **Polyparse** – alternativní kombinátorová parsovací knihovna

Parsec

Myšlenka:

- definujme parser pomocí většího množství menších/jednodušších parserů

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 názvy přednášek o Haskelli:
Lecture10 Monadic parsing; spring 2018

Motivace

Chceme parsovat názvy přednášek o Haskelli:
Lecture10 Monadic parsing; spring 2018

- 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 parsrů 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`

[1] typ je ve skutečnosti obecnější, viz později

Aplikace parserů

Na aplikaci parsrů 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 f` uspěje, jestliže načtený znak splňuje zadaný predikát (pak vrací tento znak)

Základní znakový parser

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

- parser `satisfy f` 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
```

Základní znakový parser

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

- parser `satisfy f` 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: *Lecture10 Monadic parsing; spring 2018*

- 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")
  ```
  ```

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
  ```

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` (`return x`) nic nečte a vrátí hodnotu `x`

- ```
  twoChars = do
      char1 <- anyChar
      char2 <- anyChar
      pure [char1, char2]
  ```
  ```

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` (`return 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 :: Parsec a -> Parsec [a]
sepBy, sepBy1 :: Parsec a -> Parsec sep -> Parsec [a]
```

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 :: Parsec a -> Parsec [a]
sepBy, sepBy1 :: Parsec a -> Parsec sep -> Parsec [a]
```

- Kolik vstupu spracuje `many`?
  `parse (many digit) "input name" "hello"`
- Jak vyparsovat užitečné věci z názvu přednášky?
  Lecture10 Monadic parsing; spring 2018

Příklad kombinace parsrů v do-bloku

Příklad: *Lecture10 Monadic parsing; spring 2018*

```
lecture :: Parser (Int, String, String, Int)
lecture = do
  string "Lecture"
  id <- count 2 digit
  spaces
  name <- many (noneOf [';'])
  string "; "
  semester <- many letter
  spaces
  year <- count 4 digit
  pure (read id, name, semester, read year)
```

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"`

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"
```

Alternativa parserů

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

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

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

- `>>`

```
spaces >> identifier
```

Další užitečné operátory

- `>>` `spaces >> identifier`
- `liftAX` `liftA2 (+) number number`

Další užitečné operátory

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

```
spaces >> identifier
liftA2 (+) number number
(++"!") <$> greeting
```

Další užitečné operátory

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

```
spaces >> identifier
liftA2 (+) number number
(++"!") <$> greeting
"greeting here" <$ greeting
```

Další užitečné operátory

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

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

Další užitečné operátory

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

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

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 parsrů: Ladění není snadné.
(Který parser selhal?)

Užitečné chybové hlášky

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

Zlepšení: Doplňme do parsru popis, co dělá:

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

Užitečné chybové hlášky

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

Zlepšení: Doplňme do parsru 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`

Komplexní příklad

Příklad: *Lecture10 Monadic parsing; spring 2018*

```
data Lecture = Lecture Int String Semester deriving Show
data Semester = Semester String Int deriving Show

semester2 :: Parser Semester
semester2 = liftM2 Semester
  (string "spring" <|> string "autumn")
  (spaces *> fmap read (count 4 digit))
    <?> "semester spec"

lecture2 :: Parser Lecture
lecture2 = liftM3 Lecture
  (string "Lecture" *> fmap read (count 2 digit <* space))
  (many (noneOf [';']) <* string "; ")
  (semester2)
    <?> "lecture 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`

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
```

Samostatná práce

Napište parser pro:

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