Parsování v Haskellu, knihovna Parsec

IB016 Seminář z funkcionálního programování

Vladimír Štill, Martin Ukrop

Fakulta informatiky, Masarykova univerzita

Jaro 2015

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
- 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`

Datový typ Parser

```
myParser :: Parser a
```

`myParser` je parser, který zpracuje vstup na hodnotu typu `a`

Datový typ Parser

`myParser :: Parser a`

`myParser` je parser, který zpracuje vstup na hodnotu typu `a` Typ

`Parser a` je však pouze specifický 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` a vrací výsledek typu `a`.

Aplikace parserů

Jelikož dovnitř typu `ParsecT s u m a` nevidíme, voláme parsery pomocí specializovaných funkcí.

```
parse :: Stream s Identity t =>
    Parsec s () a -> SourceName -> s ->
    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 sparsovaná hodnota nebo chyba (typu `ParseError`)

Aplikace parserů

Jelikož dovnitř typu `ParsecT s u m a` nevidíme, voláme parsery pomocí specializovaných funkcí.

```
parse :: Stream s Identity t =>
    Parsec s () a -> SourceName -> s ->
    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 sparsovaná hodnota nebo chyba (typu `ParseError`)

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

- `parseFromFile p f` spustí parser `p` na obsahu souboru `f`
- importováno z modulu `Text.Parsec.String`

Základní znakový parser

- Pro zpřehlednění typů používáme typové synonymum `Parser` z modulu `Text.Parsec.String` (zmiňované parsery jsou obecnější).
- `Parser` má instanci pro třídy `Functor`, `Applicative` i `Monad`, což nám usnadní práci s vícero parsery.

Základní znakový parser

- Pro zpřehlednění typů používáme typové synonymum `Parser` z modulu `Text.Parsec.String` (zmiňované parsery jsou obecnější).
- `Parser` má instanci pro třídy `Functor`, `Applicative` i `Monad`, což nám usnadní práci s vícero parsery.

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

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

Základní znakový parser

- Pro zpřehlednění typů používáme typové synonymum `Parser` z modulu `Text.Parsec.String` (zmiňované parsery jsou obecnější).
- `Parser` má instanci pro třídy `Functor`, `Applicative` i `Monad`, což nám usnadní práci s vícero parsery.

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

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

`char`, `digit`, `letter`, `anyChar`, `space`, `oneOf`, `noneOf`, ...

Kombinace parserů – sekvence

Využijeme instanci `Parser` pro třídu `Monad`:

- `return x` je parser, který nic nečte a vrátí hodnotu `x`
- operátor >>= předá sparsovanou hodnotu zleva doprava

Kombinace parserů – sekvence

Využijeme instanci `Parser` pro třídu `Monad`:

- `return x` je parser, který nic nečte a vrátí hodnotu `x`
- operátor `>>=` předá sparsovanou hodnotu zleva doprava

Parser pro 2 libovolné znaky:

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

Kombinace parserů – sekvence

Využijeme instanci `Parser` pro třídu `Monad`:

- `return x` je parser, který nic nečte a vrátí hodnotu `x`
- operátor `>>=` předá sparsovanou hodnotu zleva doprava

Parser pro 2 libovolné znaky:

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

`string`, `between`, `count`, ...

Kombinace parserů – alternativa

```
<|> :: 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`

Kombinace parserů – alternativa

```
<|> :: 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`

Parser pro pozdrav:

```
greeting = string "hello" <|> string "ahoj"
```

Kombinace parserů – alternativa

```
<|> :: 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`

Parser pro pozdrav:

```
greeting = string "hello" <|> string "ahoj"
```

`many`, `many1`, `option`, `optional`, `sepBy`, `sepBy1`, ...

Kombinace parserů – alternativa

```
greeting2 = string "hello" <|> string "how-are-you"
```

Kombinace parserů – alternativa

```
greeting2 = string "hello" <|> string "how-are-you"
```

`greeting2` na řetězci `"how-are-you"` selže – levá alternativa sice selhala, ale spotřebovala vstup!

Kombinace parserů – alternativa

```
greeting2 = string "hello" <|> string "how-are-you"
```

`greeting2` na řetězci `"how-are-you"` selže – levá alternativa sice selhala, ale spotřebovala vstup!

```
try :: ParsecT s u m a -> ParsecT s u m 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*.)

Instance pro Applicative, Monad

- `>>`

```
spaces >> identifier
```

Instance pro Applicative, Monad

- `>>`

  ```
  spaces >> identifier
  ```

- `liftAX`

  ```
  liftA2 (+) number number
  ```

Instance pro Applicative, Monad

- `>>`

  ```
  spaces >> identifier
  ```

- `liftAX`

  ```
  liftA2 (+) number number
  ```

- `<$>`

  ```
  (++"!") <$> greeting
  ```

Instance pro Applicative, Monad

- `>>`

  ```
  spaces >> identifier
  ```

- `liftAX`

  ```
  liftA2 (+) number number
  ```

- `<$>`

  ```
  (++"!") <$> greeting
  ```

- `<$`

  ```
  "greeting here" <$ greeting
  ```

Instance pro Applicative, Monad

- `>>`

  ```
  spaces >> identifier
  ```

- `liftAX`

  ```
  liftA2 (+) number number
  ```

- `<$>`

  ```
  (++"!") <$> greeting
  ```

- `<$`

  ```
  "greeting here" <$ greeting
  ```

- `<*>`

  ```
  (,) <$> key <*> value
  ```

Instance pro Applicative, Monad

- `>>`

```
spaces >> identifier
```

- `liftAX`

```
liftA2 (+) number number
```

- `<$>`

```
(++"!") <$> greeting
```

- `<$`

```
"greeting here" <$ greeting
```

- `<*>`

```
(,) <$> key <*> value
```

- `<*`

```
string "Hello" <* spaces <* name
```

Instance pro Applicative, Monad

- `>>`

```
spaces >> identifier
```

- `liftAX`

```
liftA2 (+) number number
```

- `<$>`

```
(++"!") <$> greeting
```

- `<$`

```
"greeting here" <$ greeting
```

- `<*>`

```
(,) <$> key <*> value
```

- `<*`

```
string "Hello" <* spaces <* name
```

- `*>`

```
string "Hello" *> spaces *> name
```

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`