

Revisão de Haskell

Construção de compiladores I

Objetivos

Objetivos

- Revisar conceitos sobre funtores aplicativos.
- Construção de parsers usando funtores aplicativos.

Markdown

Markdown

- Na última aula, iniciamos a construção de um compilador de Markdown para HTML
 - Criação de uma EDSL para Html
 - Uso da EDSL para produzir Html a partir da AST de Markdown

Markdown

- Porém, como produzir a AST a partir de texto armazenado em arquivos?
- Como manipular argumentos de entrada para o programa?

Markdown

- Para produzir a AST, precisamos construir um **parser**.
- Parser: função de tipo **String** -> **AST**.

Markdown

- Idealmente, para especificarmos um parser, precisamos de uma gramática.

Gramáticas

Gramáticas

- Uma gramática livre de contexto $G = (V, \Sigma, R, P)$:
 - V : conjunto finito de variáveis
 - Σ : alfabeto
 - $R \subseteq V \times (V \cup \Sigma)^*$: regras
 - $P \in V$: variável de partida.

Gramáticas

- Exemplo: gramática livre de contexto para palavras com a mesma quantidade de 0s e 1s:

$$P \rightarrow 0P1P \mid 1P0P \mid \lambda$$

Gramáticas

- Na gramática anterior, temos:
 - $V = \{P\}$, $\Sigma = \{0, 1\}$
 - Três regras:

$$P \rightarrow 0P1P$$

$$P \rightarrow 1P0P$$

$$P \rightarrow \lambda$$

Gramáticas

- Representando a AST:

```
data AST
= Zero AST AST -- starts with 0
| One AST AST  -- starts with 1
| Empty        -- empty string
```

Gramáticas

- Como construir um parser para esta gramática?

```
parse :: String -> AST
```

Gramáticas

- Tentando implementar diretamente:

```
parse :: String -> AST
parse = fst . parse'
  where
    parse' [] = (Empty, [])
    parse' ('0' : rest) =
      let (t1,r1) = parse' rest
          (t2,r2) = parse' r1
      in (Zero t1 t2, r2)
    parse' ('1' : rest) =
      let (t1,r1) = parse' rest
          (t2,r2) = parse' r1
      in (One t1 t2, r2)
    parse' _ = ???
```

Gramáticas

- Problemas da implementação anterior:
 - Passagem explícita do restante da string.
 - Repetição de código.
 - Como lidar com erros?

Gramáticas

- Lidar com manipulação de erros tende a aumentar a complexidade do código.

```
parse' [] = Just (Empty, [])
parse' ('0' : rest) =
  case parse' rest of
    Just (t1,r1) ->
      case parse' r1 of
        Just (t2,r2) -> Just (Zero t1 t2, r2)
        _ -> Nothing
    _ -> Nothing
```

Gramáticas

- Para construir parsers, utilizaremos a EDSL da biblioteca `mega-parsec`.
- Em essência, utiliza os conceitos de funtores aplicativos para construir parsers.

Funtores aplicativos

Funtores aplicativos

- Faremos um pequeno “desvio” para revisar o conceito de functor aplicativo.
- Esse conceito é fundamental para desenvolvimento em Haskell.

Funtores aplicativos

- Functor: Abstração que permite aplicar uma função sobre elementos de uma estrutura.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

Funtores aplicativos

- Exemplo: `Maybe` é um Functor.

```
instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just v) = Just (f v)
```

Funtores aplicativos

- Exemplo:

```
not <$> Nothing = Nothing
not <$> (Just True) = Just (not True) = Just False
```

Funtores aplicativos

- Intuitivamente, funtores permitem que façamos chamadas de função a todos os elementos de uma estrutura de dados.

```
(2 *) <$> Nothing = Nothing
(2 *) <$> (Just 2) = Just (2 * 2) = Just 4
```

Funtores aplicativos

- Usamos funtores para aplicar funções a elementos **dentro** de uma estrutura.
- Pergunta: como realizar chamadas de funções armazenadas dentro de estrutura de dados?

Funtores aplicativos

- A abstração de functor aplicativo modela a capacidade de chamar funções dentro de uma estrutura de dados

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Funtores aplicativos

- O principal componente da abstração de functor aplicativo é o operador <*>.
 - Responsável por chamadas de função dentro de estruturas de dados.

```
(<*>) :: f (a -> b) -> f a -> f b
```

Funtores aplicativos

- O tipo Maybe é um functor aplicativo.

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  _ <*> Nothing = Nothing
  (Just f) <*> (Just x) = Just (f x)
```

Funtores aplicativos

- Exemplos

```
(Just not) <*> Nothing = Nothing
(Just not) <*> (Just False) = Just (not False) = Just True
```

Funtores aplicativos

- Exemplos

```
(Just not) <*> Nothing = Nothing
(Just not) <*> (Just False) = Just (not False) = Just True
```

Funtores aplicativos

- Podemos combinar funtores e funtores aplicativos

```
(*) <$> (Just 2) <*> (Just 3) =
```

Funtores aplicativos

- Podemos combinar funtores e funtores aplicativos

```
(*) <$> (Just 2) <*> (Just 3) =
Just (2 *) <*> (Just 3) =
```

Funtores aplicativos

- Podemos combinar funtores e funtores aplicativos

```
(*) <$> (Just 2) <*> (Just 3) =
Just (2 *) <*> (Just 3) =
Just (2 * 3) = Just 6
```

Funtores aplicativos

- No contexto de parsing, o operador <*> representa a composição sequencial de parsers.
- O operador <\$> é utilizado para construir o resultado do parsing.

Funtores aplicativos

- De volta à gramática original:

$$P \rightarrow 0P1P \mid 1P0P \mid \lambda$$

Funtores aplicativos

- E a árvore que representa o resultado do parsing.

```
data AST
  = Zero AST AST -- starts with 0
  | One AST AST  -- starts with 1
  | Empty        -- empty string
```

Funtores aplicativos

- O parsing utilizando a biblioteca mega-parsec é:

```
type Parser = Parsec Void String

sample :: Parser AST
sample = try startWithZero <|>
         try startWithOne  <|>
         pure Empty

startWithZero :: Parser AST
startWithZero
  = f <$> symbol "0" <*> sample <*> symbol "1" <*> sample
  where
    f _ t _ t' = Zero t t'

startWithOne :: Parser AST
startWithOne
  = g <$> symbol "1" <*> sample <*> symbol "0" <*> sample
  where
    g _ t _ t' = One t t'
```

Funtores aplicativos

- Concatenação (sequência) é representada pelo operador <*>.
- Alternativas são representadas pelo operador <|>.

Funtores aplicativos

- Símbolos são representados pela função `symbol`.
- Função `try` permite o backtracking.
 - Caso aconteça um erro, a entrada é passada não modificada para próxima alternativa.
- Variáveis da gramática correspondem a chamadas de função.

Parser de Markdown

Parser de Markdown

- Gramática de Markdown

$$\begin{aligned} \textit{Document} &\rightarrow \textit{Structure}^* \\ \textit{Structure} &\rightarrow \textit{UnorderedList} \\ &\quad | \textit{CodeList} \\ &\quad | \textit{Heading} \\ &\quad | \textit{Paragraph} \end{aligned}$$

Parser de Markdown

- Gramática de Markdown (continuação)

$$\begin{aligned} \textit{UnorderedList} &\rightarrow -\Sigma^* \\ \textit{CodeList} &\rightarrow >\Sigma^* \\ \textit{Paragraph} &\rightarrow \Sigma^* \\ \textit{Heading} &\rightarrow \textit{Level} \textit{Paragraph} \\ \textit{Level} &\rightarrow * | ** | *** \end{aligned}$$

Parser de Markdown

- De posse da gramática, podemos representar seu parser:
 - Parser `sc`: remove espaços em branco entre linhas.
 - Linhas são separadas por uma quebra de linha.

```
pDocument :: Parser Document
pDocument = (sc *> pLine) 'sepBy' newline
```


Parser de Markdown

- Parser de uma linha do arquivo

```
pLine :: Parser Structure
pLine = choice [ pUnorderedList
                , pCodeBlock
                , pHeading
                , pParagraph
                ]
```

Parser de Markdown

- Parser de linha não ordenada

```
pUnorderedList :: Parser Structure
pUnorderedList = (UnorderedList . wrap) <$> p
  where
    p = symbol "-" *> pString
```

Parser de Markdown

- Parser de parágrafos

```
pParagraph :: Parser Structure
pParagraph = Paragraph <$> pString

pString :: Parser String
pString = many (satisfy (flip notElem "\n*->"))
```

Parser de Markdown

- Demais casos, seguem estruturas similares.

Concluindo

Concluindo

- Nesta aula, revisamos o conceito de functor applicativo e como este é utilizado para construir parsers na biblioteca `megaparsec`.

Concluindo

- Na próxima aula veremos outro padrão importante para estruturar programas: Mônadas.

Exercícios

Exercícios

- Estenda o parser de Markdown com suporte a
 - Listas ordenadas
 - Links
 - Imagens

Exercícios

- Gramática para listas ordenadas.

$$OrderedList \rightarrow \# \Sigma^*$$

Exercícios

- Gramática para links

$$Link \rightarrow \Sigma^*$$

Exercícios

- Gramática para imagens

$$Image \rightarrow !\Sigma^*$$