Análise sintática

Construção de compiladores I

Objetivos

Objetivos

- Apresentar a importância da etapa de análise sintática.
- Revisar conceitos de gramáticas e linguagens livres de contexto

Objetivos

- Apresentar como representar programas como árvores de sintaxe.
  - Como árvores de sintaxe podem ser codificadas como tipos em Haskell.
- Apresentar a técnica de análise sintática descendente recursiva.

Análise sintática

Análise sintática

- Responsável por determinar se o programa atende as restrições sintáticas da linguagem.

Análise sintática

- Regras sintáticas de uma linguagem são expressas utilizando gramáticas livres de contexto.

Análise sintática

- Porque utilizar GLCs e não ERs?
  - ERs não são capazes de representar construções simples de linguagens.

Análise sintática

- Vamos considerar um fragmento de expressões formado por variáveis, constantes inteiras adição, multiplicação.

Análise sintática

- A seguinte ER representa essa linguagem:

$$base = [a..z]([a..z]|[0..9])^*$$
$$base((+|*)base)^*$$

Análise sintática

- A ER anterior aceita palavras como $a * b + c$.
- Porém, como determinar a precedência entre operadores?

Análise sintática

- Podemos usar a precedência usual da aritmética.
- Porém, não é possível impor uma ordem diferente de avaliação.
  - Para isso, precisamos de parêntesis.

Análise sintática

- Ao incluir parêntesis, temos um problema:
  - Como expressar usando ER que parêntesis estão corretos?

Análise sintática

- Pode-se provar que a linguagem de parêntesis balanceados não é regular.
  - Usando o lema do bombeamento.
  - Estrutura similar a $\{0^n 1^n \mid n \geq 0\}$.

Análise sintática

- Dessa forma, precisamos utilizar GLCs para representar a estrutura sintática de linguagens.

Análise sintática

- Antes de apresentar técnicas de análise sintática, vamos revisar alguns conceitos sobre GLCs.

Gramáticas Livres de Contexto

Gramáticas livres de contexto

- Uma GLC é $G = (V, \Sigma, R, P)$, em que
  - V: conjunto de variáveis (não terminais)
  - Σ: alfabeto (terminais)
  - $R \subseteq V \times (V \cup \Sigma)^*$: regras (produções).
  - $P \in V$: variável de partida.

Gramáticas livres de contexto

- Gramática de expressões

$$E \quad \rightarrow \quad (E) \,|\, E + E \,|\, E * E \,|\, num \,|\, var$$

Gramáticas livres de contexto

- $V = \{E\}$
- $\Sigma = \{num, var, (,), *, +\}$
- R: conjunto de regras da gramática.

Gramáticas livres de contexto

- Determinamos se uma palavra pertence ou não à linguagem de uma gramática construindo uma **derivação**

Gramáticas livres de contexto

- Exemplo: Derivação de $num + num * num$.

$$E \quad \Rightarrow$$

Gramáticas livres de contexto

- Exemplo: Derivação de $num + num * num$.

$$\begin{array}{ll} E & \Rightarrow \quad \textbf{regra } E \to E + E \\ E + E & \end{array}$$

Gramáticas livres de contexto

- Exemplo: Derivação de $num + num * num$.

$$\begin{array}{ll} E & \Rightarrow \quad \textbf{regra } E \to E + E \\ E + E & \Rightarrow \quad \textbf{regra } E \to num \end{array}$$

Gramáticas livres de contexto

- Exemplo: Derivação de $num + num * num$.

$$\begin{array}{ll} E & \Rightarrow \quad \textbf{regra } E \to E + E \\ E + E & \Rightarrow \quad \textbf{regra } E \to num \\ num + E & \end{array}$$

Gramáticas livres de contexto

- Exemplo: Derivação de $num + num * num$.

$$\begin{array}{ll} E & \Rightarrow \quad \textbf{regra } E \to E + E \\ E + E & \Rightarrow \quad \textbf{regra } E \to num \\ num + E & \Rightarrow \quad \textbf{regra } E \to E * E \\ num + E * E & \end{array}$$

Gramáticas livres de contexto

- Exemplo: Derivação de $num + num * num$.

$$\begin{array}{ll} E & \Rightarrow \quad \textbf{regra } E \to E + E \\ E + E & \Rightarrow \quad \textbf{regra } E \to num \\ num + E & \Rightarrow \quad \textbf{regra } E \to E * E \\ num + E * E & \Rightarrow \quad \textbf{regra } E \to num \end{array}$$

Gramáticas livres de contexto

- Exemplo: Derivação de $num + num * num$.

$$
\begin{array}{lcl}
E & \Rightarrow & \textbf{regra } E \rightarrow E + E \\
E + E & \Rightarrow & \textbf{regra } E \rightarrow num \\
num + E & \Rightarrow & \textbf{regra } E \rightarrow E * E \\
num + E * E & \Rightarrow & \textbf{regra } E \rightarrow num \\
num + num * E & &
\end{array}
$$

Gramáticas livres de contexto

- Exemplo: Derivação de $num + num * num$.

$$
\begin{array}{lcl}
E & \Rightarrow & \textbf{regra } E \rightarrow E + E \\
E + E & \Rightarrow & \textbf{regra } E \rightarrow num \\
num + E & \Rightarrow & \textbf{regra } E \rightarrow E * E \\
num + E * E & \Rightarrow & \textbf{regra } E \rightarrow num \\
num + num * E & \Rightarrow & \textbf{regra } E \rightarrow num \\
num + num * num & &
\end{array}
$$

Gramáticas livres de contexto

- O exemplo anterior foi de uma **derivação mais à esquerda**
  - Expande-se o não terminal mais a esquerda.

Gramáticas livres de contexto

- Note que essa gramática de expressões permite:

$$
\begin{array}{lcl}
E & \Rightarrow & \textbf{regra } E \rightarrow E * E \\
E * E & &
\end{array}
$$

Gramáticas livres de contexto

- Com isso temos **duas** derivações distintas para a mesma palavra.
- Isso torna a gramática de exemplo **ambígua**.

Gramáticas livres de contexto

- Em algumas situações é necessário modificar regras de uma gramática para usar certas técnicas de análise sintática.
- Veremos algumas dessas técnicas.

Transformações de gramáticas

Transformações de gramáticas

- Fatoração à esquerda: Evitar mais de uma regra com o mesmo prefixo

Transformações de gramáticas

- Exemplo:

$$A \rightarrow xz \,|\, xy \,|\, v$$

- pode ser transformada em:

$$\begin{aligned} A &\rightarrow xZ \,|\, v \\ Z &\rightarrow z \,|\, y \end{aligned}$$

Transformações de gramáticas

- Introdução de prioridades.
  - Problema comum em linguagens de programação com operadores.
  - Impor ordem de precedência na ausência de parêntesis.

Transformações de gramáticas

- Forma geral para introduzir prioridades:
  - E_i: expressões com precedência de nível i.
  - Maior precedência: mais profundo.

$$E_i \rightarrow E_{i+1} \,|\, E_i Op_i E_{i+1}$$

Transformação de gramáticas

- Exemplo:
  - Multiplicação com predência maior que adição.

$$E \rightarrow n \mid E + E \mid E * E$$

Transformação de gramáticas

- Exemplo

$$\begin{aligned} E_1 &\rightarrow E_1 + E_2 \mid E_2 \\ E_2 &\rightarrow E_2 * E_3 \mid E_3 \\ E_3 &\rightarrow n \end{aligned}$$

Transformações de gramáticas

- Eliminar recursão à esquerda
  - Transformar em recursão à direita.

$$A \rightarrow Ay_1 \mid ... \mid Ay_n \mid w_1 \mid ... \mid w_k$$

Transformações de gramáticas

- Pode ser transformada em

$$\begin{aligned} A &\rightarrow w_1 Z \mid ... \mid w_k Z \mid w_1 ... \mid w_k \\ Z &\rightarrow y_1 Z \mid ... \mid y_n Z \mid y_1 ... \mid y_n \end{aligned}$$

Transformação de gramáticas

- Eliminar recursão a esquerda.
  - Resolução no quadro

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \mid \lambda \end{aligned}$$

Árvores de sintaxe

Árvores de sintaxe

- Em teoria de linguagens, representamos derivações de uma gramática por **árvores de derivação**.
- Uma árvore de sintaxe deve representar a estrutura da derivação de um programa.

Árvores de sintaxe

- Estratégia para definir árvores de sintaxe
  - Um tipo para cada não terminal da gramática.
  - Cada regra de um não terminal, é um construtor do tipo.

Árvores de sintaxe

- Qual a estrutura da árvore de sintaxe:

$$E \quad \rightarrow \quad num \,|\, var \,|\, (E) \,|\, E + E \,|\, E * E$$

Árvores de sintaxe

- Árvore de sintaxe

```
data Exp = Const Int
         | Var String
         | Add Exp Exp
         | Mul Exp Exp
```

Árvores de sintaxe

- Porque não uma construção para parêntesis?
  - São usados apenas para determinar precedência
  - A rigor, parêntesis não tem significado após análise sintática.

Árvores de sintaxe

- O tipo anterior é um exemplo de sintaxe **abstrata**
  - Elimina detalhes irrelevantes para o significado do programa.
- Código escrito do programa usa a sintaxe **concreta**.

Árvores de sintaxe

- Considere a seguinte gramática:

$$S \rightarrow S\,S \mid s$$

Árvores de sintaxe

- Representando a árvore de sintaxe

```
data S = Rule1 S S | Rule2 Char
```

Árvores de sintaxe

- Considere a tarefa de produzir a string representada pela árvore

```
pprS :: S -> String
pprS (Rule1 s1 s2) = pprS s1 ++ pprS s2
pprS (Rule2 _) = "s"
```

Árvores de sintaxe

- Note que o construtor `Rule2 Char` não usa o caracter que armazena
  - Sempre vamos produzir o caractere `s`.

Árvores de sintaxe

- Podemos refinar a árvore para

```
data SA = Rule1 SA SA | Rule2
```

Árvores de sintaxe

- Refinando a função de impressão

```
pprS :: SA -> String
pprS (Rule1 s1 s2) = pprS s1 ++ pprS s2
pprS Rule2 = "s"
```

Análise descendente

Análise descendente

- Na apresentação do compilador de expressões, implementamos funções simples para um analisador descendente
- Apesar de possuir uma implementação simples:
  - Não é eficiente
  - Não permite uma maneira adequada para lidar com erros de análise sintática.

Análise descendente

- Vamos utilizar a biblioteca `megaparsec`
  - Permite a construção de analisadores descendentes eficientes.
  - Bom suporte a mensagens de erro.

Análise descendente

- Excelente documentação disponível on-line:

`https://markkarpov.com/tutorial/megaparsec.html`

Análise descendente

- Vamos apresentar a implementação do parser de expressões usando `megaparsec`
- Exemplo disponível no módulo `Megaparsec.ParserExample` do repositório.

Análise descendente

- Primeiro passo: definir um tipo para os parsers e erros

```
type Parser = Parsec Void String

type ParserError = ParseErrorBundle String Void
```

Análise descendente

- Segundo passo: definir um analisador léxico.

```
slexer :: Parser ()
slexer = L.space space1
                 (L.skipLineComment "//")
                 (L.skipBlockComment "/*" "*/")
```

Análise descendente

- Definindo funções simples.

```
symbol :: String -> Parser String
symbol s = L.symbol slexer s
```

Análise descendente

- Lidando com parêntesis

```
parens :: Parser a -> Parser a
parens = between (symbol "(") (symbol ")")
```

Análise descendente

- Adicionando a capacidade de eliminar espaços e comentários em um parser qualquer.

```
lexeme :: Parser a -> Parser a
lexeme = L.lexeme slexer
```

Análise descendente

- Processando números

```
integer :: Parser Int
integer = lexeme L.decimal
```

Análise descendente

- Processando um fator

```
pFactor :: Parser Exp
pFactor = choice [ Const <$> integer
                 , parens pExp
                 ]
```

Análise descendente

- Para criar o parser de expressões, vamos usar a função `makeExprParser` que constrói o parser a partir de uma tabela de precedências.

Análise descendente

- Definindo uma função para criar a precedência de um operador binário.
  - Pode-se definir operadores unários pré-fixados (`Prefix`) e pós-fixados (`Postfix`)

```
binary :: String -> (Exp -> Exp -> Exp) -> Operator Parser Exp
binary name f = InfixL (f <$ symbol name)
```

Análise descendente

- Usando a função anterior, podemos criar a tabela de precedências.
  - Maiores precedências aparecem primeiro na tabela.

```
optable :: [[Operator Parser Exp]]
optable = [
            [binary "*" Mul]
          , [binary "+" Add]
          ]
```

Análise descendente

- Parser de expressões

```
pExp :: Parser Exp
pExp = makeExprParser pFactor optable
```

Análise descendente

- Podemos processar qualquer gramática usando analisadores descendentes?
  - Não: essa técnica aplica-se a gramáticas da classe LL(k).

Análise descendente

- Gramáticas LL(k)
  - **L** : Entrada processada da esquerda para a direita
  - **L**: Produzindo uma derivação mais a esquerda
  - **k**: tomando a decisão usando até **k** tokens da entrada.

Análise descendente

- Gramáticas LL(k)
  - Não possuem recursão à esquerda
  - Não possuem fatores comuns à esquerda
- De maneira geral, gramáticas LL(k) não possuem **ambiguidade**

Análise descendente

- Então, para determinar se uma gramática é LL(k), basta determinar se ela é ou não ámbigua...

Análise descendente

- Ambiguidade de gramáticas livres de contexto é um problema indecidível, no caso geral.
  - Pode-se reduzir o problema de correspondência de Post a ele.

Análise descendente

- Vantagens:
  - Analisadores descendentes são eficientes, para $k = 1$.
  - Simples implementação.

Análise descendente

- Desvantagens:
  - Não são capazes de lidar com gramáticas com regras recursivas à esquerda.
  - Regras não devem possuir fatores comuns à esquerda.

Análise descendente

- Algum compilador usa essa técnica?
  - Analisador sintático de Lua e Go é descendente recursivo.
  - Analisador sintático de Clang é baseado nesta técnica.

Conclusão

Conclusão

- Nesta aula:
  - Importântica da análise sintática em um compilador.
  - Revisamos conceitos de gramáticas livres de contexto e transformações sobre estas.

Conclusão

- Nesta aula:
  - Discutimos sobre sintaxe concreta e abstrata.
  - Mostramos como deduzir uma árvore de sintaxe a partir de uma gramática.

Conclusão

- Nesta aula:
  - Apresentamos a técnica de análise descendente recursiva.
  - Usamos a biblioteca `megaparsec` para construção de um analisador descendente.
  - Discutimos vantagens e desvantagens de analisadores descendentes.

Conclusão

- Próxima aula:
  - Análise sintática preditiva LL(1).