Análise sintática

Construção de compiladores I

Objetivos

Objetivos

- Introduzir a segunda etapa do front-end: a análise sintática.
- Introduzir o conceito de analisadores top-down e bottom-up.

Objetivos

- 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:

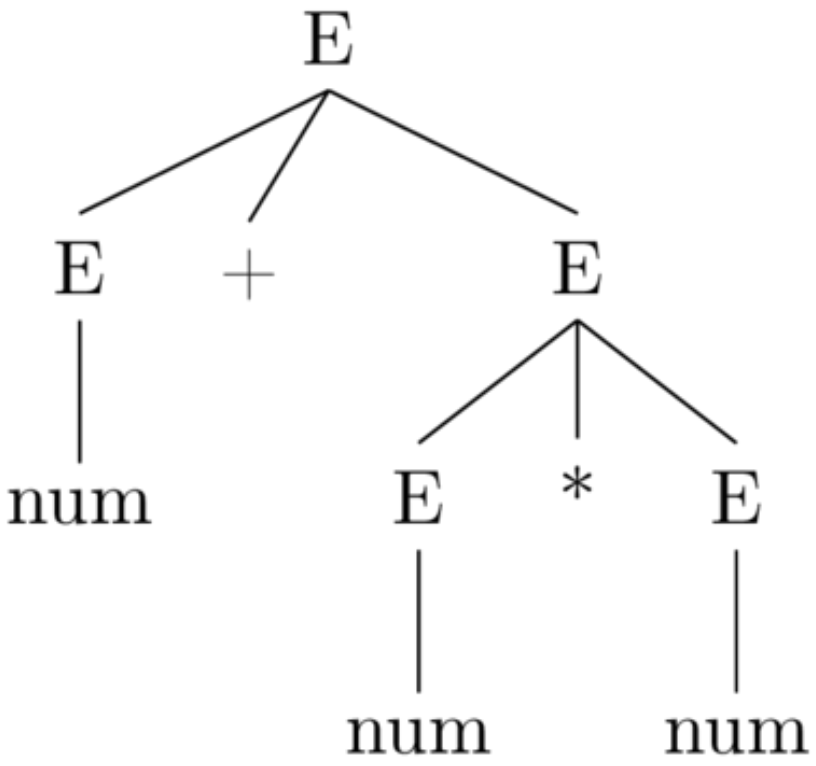
$$
\begin{array}{lcl}
E & \Rightarrow & \textbf{regra } E \rightarrow E * E \\
E * E & &
\end{array}
$$

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

- Árvores de derivação: representação hierárquica da derivação.



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:

$$A \rightarrow xz \mid xy \mid v$$

- pode ser transformada em:

$$\begin{array}{lcl} A & \rightarrow & xZ \mid v \\ Z & \rightarrow & z \mid y \end{array}$$

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.

$$E_i \rightarrow E_{i+1} \mid E_i Op_i E_{i+1}$$

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

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

Transformações de gramáticas

- Exemplo:
  - $*$ tem prioridade maior que $+$

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

Transformações de gramáticas

- Exemplo:
  - $*$ tem prioridade maior que $+$

$$\begin{aligned} E_1 &\rightarrow E_2 \,|\, E_1 + E_2 \\ E_2 &\rightarrow E_3 \,|\, E_2 * E_3 \\ E_3 &\rightarrow num \,|\, var \,|\, (E_1) \end{aligned}$$

Transformação de gramáticas

- Eliminar recursão a esquerda.
  - Resolução no quadro

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

Análise sintática

Análise sintática

- Dois tipos de algoritmos:
  - Análise sintática top-down
  - Análise sintática bottom-up

Análise sintática

- Analisador sintático top-down
  - Inicia a partir do símbolo de partida da gramática.
  - A cada passo, escolhe um não terminal para expandir até
    - Obter o programa de entrada.
    - Encontrar um erro.

Análise sintática

- Analisador sintático bottom-up
  - Inicia a partir das folhas
  - Encontra substrings da palavra e encontra um lado direito de regra correspondente.

Análise sintática

- Nesta aula, vamos focar em uma técnica top-down.
  - Analisador sintático descendente recursivo.

Análise descendente

Análise descendente

- Técnica simples para codificação manual de analisadores sintáticos.
- Gramáticas são representadas por um conjunto de funções
  - Uma função para cada não terminal.

Análise descendente

- Analisador para $A \rightarrow X_1 ... X_n$:
  - Para $i = 1$ até n
    - Se X_i for uma variável, chame a função correspondente a X_i.
    - Se X_i for um terminal, verifique se ele é igual ao primeiro token da entrada.

Análise descendente

- Vantagem:
  - Codificação simples e de fácil compreensão.
- Problemas:
  - Não permite gramática com recursão à esquerda.

Análise descendente

- Em Haskell, analisadores descendentes são codificados por combinadores.

Análise descendente

- Combinadores são uma EDSL para expressar gramáticas.
  - Dúzias de bibliotecas: `megaparsec`, `parser-combinators`, `uu-parselib`, etc...

Análise descendente

- Veremos uma implementação simples de uma EDSL para gramáticas.

Análise descendente

- Representação de parsers

```
newtype Parser s a =
  Parser { runParser :: [s] -> [(a,[s])] }
```

Análise descendente

- Parsers são functores

```
instance Functor (Parser s) where
  fmap f p = Parser (\ s -> [(f x, s') | (x,s') <- runParser p s])
```

Análise descendente

- Parsers são applicatives

```
instance Applicative (Parser s) where
  pure  a = Parser (\ts -> [(a, ts)])
  p1 <*> p2 = Parser (\ s -> [(f x, s2) | (f, s1) <- runParser p1 s,
                                          (x, s2) <- runParser p2 s1])
```

Análise descendente

- Parsers são alternatives

```
instance Alternative (Parser t) where
  empty = Parser (\ _ -> [])
  p1 <|> p2 = Parser (\ s -> runParser p1 s ++ runParser p2 s)
```

Análise descendente

- Parsers são mônadas

```
instance Monad (Parser t) where
  return =  pure
  p >>= f  = Parser (\ts -> concat [ runParser (f a) cs' |
                                     (a,cs') <- runParser p ts ])
```

Análise descendente

- Processando o primeiro token da entrada

```
item :: Parser t t
item = Parser (\ ts ->
                 case ts of
                   []     -> []
                   (c:cs) -> [(c,cs)])
```

Análise descendente

- Processando um token que satisfaz uma condição.

```
sat :: (t -> Bool) -> Parser t t
sat p = do
          t <- item
          if p t then return t else mzero
```

Análise descendente

- Processando um certo token.

```
symbol :: Eq s => s -> Parser s s
symbol c = sat (c ==)
```

Análise descendente

- Devido a lazy evaluation, podemos entender recursão à esquerda em expressões como uma lista de separadores.
- Ideia sumarizada pela ER: $E(op\, E)^*$

Análise descendente

- Função `chainl`

```
chainl  ::  Parser s a -> Parser s (a -> a -> a) -> Parser s a
chainl pe po  =  h <$> pe <*> many (j <$> po <*> pe)
  where j op x  =  (`op` x)
        h x fs  =  foldl (flip ($)) x fs
```

Análise descendente

- Automatizando parsers de expressões

```
type Op s a = (s, a -> a -> a)

gen :: Eq s => [Op s a] -> Parser s a -> Parser s a
gen ops p = chainl p (choice (map f ops))
  where
    f (s,c) = const c <$> symbol s
```

Análise descendente

- Tipo do token

```
data Token
  = Id String
  | Number Int
  | Add
  | Mult
  | LParen
  | RParen
  deriving (Eq, Show)
```

Análise descendente

- Tipo de expressões

```
data Expr
  = Var String
  | Lit Int
  | Expr :+: Expr
  | Expr :*: Expr
  deriving (Eq, Show)
```

Análise descendente

- Criando o parser de expressões

```
exprParser :: Parser Token Expr
exprParser = addtable `gen` termParser
  where
    addtable = [(Add, (:+:))]
```

Análise descendente

- Parser para termos

```
termParser :: Parser Token Expr
termParser = multable `gen` factParser
  where
    multable = [(Mult, (:*:))]
```

Análise descendente

- Parser para fatores

```
factParser :: Parser Token Expr
factParser = numParser  <|>
             varParser  <|>
             parenExpr
  where
    parenExpr = pack lparen exprParser rparen
    lparen = symbol LParen
    rparen = symbol RParen
```

Concluindo

Concluindo

- Apresentamos uma introdução à análise sintática.
- Revisamos sobre GLCs e transformações sobre estas.

Concluindo

- Apresentamos a técnica de análise sintática descendente recursiva.
- Em Haskell, esta técnica é representada por combinadores.

Concluindo

- Próxima aula: análise sintática LL(1).

Exercícios

Exercícios

- Construa um analisador sintático descendente recursivo para seguinte linguagem de fórmulas da lógica proposicional.

$$F \;\rightarrow\; \textbf{true} \,|\, \textbf{false} \,|\, \textbf{not}\ F \,|\, F \ /\ F \,|\, F \ /\ F$$