Compiladores

Aula 08: Gramáticas Livres de Contexto

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Março de 2026

Sumário

Revisão: O Limite das Expressões Regulares

Na Análise Léxica, usamos Expressões Regulares (ER) e Autômatos Finitos.

- ERs são excelentes para padrões lineares e repetitivos (identificadores, números).
- **A Limitação:** Autômatos Finitos não têm "memória" de contagem ilimitada.
- Eles não conseguem garantir o balanceamento de estruturas aninhadas arbitrárias, como parênteses matemáticos ou blocos { }.

Para descrever a sintaxe de linguagens de programação, precisamos de um formalismo mais poderoso: as **Gramáticas Livres de Contexto (GLC)**.

Os 4 Elementos de uma GLC

Matematicamente, uma GLC é uma quádrupla $G = (V, T, P, S)$, onde:

1. T **(Terminais):** O conjunto de símbolos básicos que formam as sentenças da linguagem. Na prática do compilador, são os **Tokens** retornados pelo scanner (ex: `if`, `id`, `+`).
2. V **(Não-Terminais ou Variáveis):** Símbolos sintáticos que representam conjuntos de cadeias (ex: *expr*, *comando*, *bloco*).
3. P **(Produções):** O conjunto de regras de reescrita da forma $A \to \alpha$, onde $A \in V$ e $\alpha \in (V \cup T)^*$.
4. S **(Símbolo Inicial):** Um não-terminal especial de V a partir do qual todas as derivações começam.

Por que "Livre de Contexto"?

A regra fundamental das produções em uma GLC é que o lado esquerdo de cada regra $A \rightarrow \alpha$ consiste de um **único não-terminal** (A).

- Isso significa que podemos substituir o símbolo A por α **independentemente do contexto** em que A aparece.
- Não importa se A está cercado por variáveis ou números; a regra se aplica da mesma forma.
- Gramáticas sensíveis ao contexto teriam regras como $\alpha A \beta \rightarrow \alpha \gamma \beta$.

O Processo de Derivação

Derivar significa tratar uma produção como uma regra de reescrita. Começamos com o Símbolo Inicial (S) e substituímos não-terminais até obtermos uma string contendo apenas terminais.

Seja a gramática para expressões simples:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \textbf{id}$$

Exemplo de Derivação ($\Rightarrow$) para a string `id + id`:

$$E \Rightarrow E + E \Rightarrow \textbf{id} + E \Rightarrow \textbf{id} + \textbf{id}$$

Estratégias de Derivação

Em cada passo de uma derivação, se houver mais de um não-terminal na expressão, qual devemos substituir primeiro?

Derivação Mais à Esquerda (Leftmost)

Sempre substitui o não-terminal mais à esquerda. É a base dos parsers **Top-Down**.

Derivação Mais à Direita (Rightmost ou Canônica)

Sempre substitui o não-terminal mais à direita. É a base dos parsers **Bottom-Up**.

A Árvore de Derivação (Parse Tree)

Uma derivação pode ser visualizada graficamente como uma árvore.

- **A Raiz:** É rotulada pelo símbolo inicial S.
- **Nós Internos:** São rotulados por não-terminais (V).
- **Folhas:** São rotuladas por terminais (T) ou pela palavra vazia (ϵ).
- **Filhos de um nó:** Se um nó interno A tem filhos $X_1, X_2, \ldots, X_n$, então deve existir uma produção $A \to X_1 X_2 \ldots X_n$ em P.

O Que é Ambiguidade?

Definição: Uma gramática é **ambígua** se ela pode gerar *mais de uma* árvore de derivação para a **mesma** cadeia de terminais.

- Em termos de derivação: Existe mais de uma derivação *mais à esquerda* (ou mais à direita) para a mesma sentença.
- **Consequência Prática:** Se o compilador puder gerar duas árvores diferentes, o programa terá comportamentos diferentes e imprevisíveis. Compiladores exigem gramáticas não-ambíguas.

Exemplo Clássico: Precedência de Operadores

Considere a cadeia: `2 + 3 * 4` e a gramática: $E \rightarrow E + E \mid E * E \mid \textbf{num}$

Árvore 1 (Soma primeiro)

- Avalia como `(2 + 3) * 4`
- Resultado Semântico: 20

Árvore 2 (Multiplicação primeiro)

- Avalia como `2 + (3 * 4)`
- Resultado Semântico: 14

Nota: É essencial desenhar as duas árvores no quadro para demonstrar estruturalmente como a multiplicação deve ficar mais abaixo na árvore para ser avaliada primeiro.

Resolvendo Ambiguidade: Reescrevendo a Gramática

Para forçar a **precedência**, quebramos a variável E em uma hierarquia (Expressão, Termo, Fator). Operações com maior precedência ficam mais "longe" do símbolo inicial.

Gramática Não-Ambígua para Aritmética

$E \rightarrow E + T \mid T$ (Adição: Precedência Menor)
$T \rightarrow T * F \mid F$ (Multiplicação: Precedência Maior)
$F \rightarrow (E) \mid \textbf{num}$ (Parênteses e Números: Precedência Máxima)

Esta gramática também resolve a **associatividade** à esquerda para a adição e multiplicação.

O Problema Clássico do "Dangling-Else"

Outra fonte comum de ambiguidade nas linguagens imperativas é o comando `if`.

Cadeia: `if (E1) then if (E2) then C1 else C2`

O `else C2` pertence ao primeiro ou ao segundo `if`?

- Sem chaves {}, a gramática permite que o `else` se ligue a qualquer um dos `if`s anteriores.
- **Solução Comum:** A linguagem C e Java definem na semântica que o `else` sempre pertence ao `if` não resolvido mais próximo (o segundo `if`).

Resumo

- GLCs (V, T, P, S) fornecem a fundação matemática para analisar estruturas aninhadas em linguagens de programação.
- O processo de *parsing* é essencialmente a busca por uma Derivação (ou a construção de uma Árvore de Análise) para a string de entrada.
- A ambiguidade é um defeito fatal em gramáticas projetadas para compiladores.
- Podemos eliminar ambiguidades reescrevendo as produções para impor precedência e associatividade estruturalmente na árvore.

Referências

- **AHO, A. V. et al.** *Compiladores: Princípios, Técnicas e Ferramentas*. 2ª Edição. Cap. 4.
- **COOPER, K.; TORCZON, L.** *Engineering a Compiler*. 2ª Edição. Cap. 3.

Dúvidas?

Obrigado!

Email: alessio@cefetmg.br
Web: alessiojr.com