Compiladores

Aula 07: Fundamentos da Análise Sintática

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Março de 2026

Sumário

Onde estamos no Compilador?

- O Analisador Léxico (Scanner) produziu um fluxo linear de **Tokens**.
- O Scanner garantiu que as "palavras" pertencem ao vocabulário, mas não sabe se a "frase" faz sentido.
- **A Analogia da Linguagem Natural:**
  - *Léxico:* Sabe que "menino", "comeu", "a" e "maçã" são palavras válidas.
  - *Sintático:* Rejeita "maçã a menino comeu o" (ordem incorreta das palavras).
- **Exemplo em C:** O scanner aceita perfeitamente `+ * = id num`, mas isso viola a sintaxe de qualquer linguagem de programação real.

As Tarefas do Parser

O Analisador Sintático (Parser) é o "coração" do front-end. Suas funções são:

1. **Verificação de Sintaxe:** Checar se o fluxo de tokens forma um programa válido segundo as regras gramaticais da linguagem.
2. **Construção da Árvore:** Produzir uma representação hierárquica em memória (geralmente uma AST - Árvore Sintática Abstrata).
3. **Tratamento de Erros:** Reportar erros de forma clara (ex: "Faltou ponto e vírgula na linha 10") e **recuperar-se** para encontrar mais de um erro por compilação.

Estratégias de Recuperação de Erros

O que o Parser faz quando encontra um erro de sintaxe?

- **Panic Mode (Modo Pânico):** A estratégia mais simples e comum. O parser descarta os tokens da entrada até encontrar um *token de sincronização* (como ; ou }).
- **Recuperação em Nível de Frase:** O parser tenta realizar correções locais. Exemplo: substituir um , por um ; ou inserir um parêntese fechando a expressão.
- **Produções de Erro:** A gramática é estendida com regras propositalmente erradas (mas comuns) para capturá-las e emitir mensagens personalizadas.

Por que não usar Expressões Regulares?

Problema: Expressões Regulares (e Autômatos Finitos) não têm "memória" de contagem irrestrita. Eles não conseguem "lembrar" quantas vezes viram um símbolo para cobrar o fechamento dele depois.

- **O Problema Clássico:** A linguagem $L = \{a^n b^n \mid n \geq 1\}$ não é regular.
- **Na Prática:** Um AFD não consegue verificar estruturas aninhadas arbitrárias.
  - Parênteses balanceados em matemática: `( ( ( a + b ) * c ) / d )`
  - Chaves aninhadas em JSON ou C: { { { } } }
  - Casamento de tags HTML: `<div> <span> </span> </div>`

A Solução: Gramáticas Livres de Contexto

Para descrever a sintaxe de linguagens de programação, subimos um degrau na Hierarquia de Chomsky: as **Gramáticas Livres de Contexto (GLC)**.

Uma GLC possui poder computacional equivalente ao de um **Autômato com Pilha** (Pushdown Automaton).

O Poder da Pilha

O uso de uma memória em formato de pilha (LIFO) permite empilhar um marcador toda vez que abrirmos um (ou { e desempilhar quando lermos um) ou }, garantindo o balanceamento perfeito.

Definição Formal de uma GLC

Uma Gramática Livre de Contexto é uma quádrupla matemática $G = (V, T, P, S)$, onde:

T (Terminais): Os símbolos básicos que formam as strings da linguagem. Na prática, são os **Tokens** retornados pelo Scanner (ex: `IF`, `NUM`, +).

V (Não-Terminais): Variáveis sintáticas que denotam conjuntos de strings (ex: *expressao*, *comando*, *bloco*).

P (Produções): Regras que ditam como não-terminais podem ser reescritos. Forma: $A \to \alpha$, onde $A \in V$ e $\alpha \in (V \cup T)^*$.

S (Símbolo Inicial): Um não-terminal especial a partir do qual todas as derivações começam.

Exemplo Prático 1: Expressões Aritméticas

Gramática G para Expressões Simples

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

- **Terminais (T):** `+`, `*`, `(`, `)`, `id`
- **Não-Terminais (V):** E
- **Símbolo Inicial (S):** E

Exemplo Prático 2: Comandos em C/Java

Como modelamos os comandos que vocês programam no dia a dia?

Gramática para Fluxo de Controle

```
Stmt     -> if ( Expr ) Stmt
          | if ( Expr ) Stmt else Stmt
          | while ( Expr ) Stmt
          | { StmtList }
          | id = Expr ;
StmtList -> Stmt StmtList | vazio
```

Note a natureza recursiva: um comando pode conter outros comandos dentro dele.

O Processo de Derivação

A derivação é o processo de substituir não-terminais pelas suas produções até chegarmos a uma string contendo apenas terminais (código fonte puro).

Exemplo para gerar a string `id + id`:

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

Estratégias de derivação (importantes para os algoritmos de parser):

- **Mais à Esquerda (Leftmost):** Substitui o não-terminal mais à esquerda em cada passo.
- **Mais à Direita (Rightmost):** Substitui o não-terminal mais à direita em cada passo.

Parse Tree vs. AST: Exemplo Prático

A derivação pode ser visualizada como uma árvore. Mas há uma grande diferença entre a árvore concreta e a abstrata. **Considere a expressão:** `5 + (2 * x)`

Parse Tree (Árvore Concreta)

- Contém todos os ruídos visuais.
- Raiz: `Expressao`
- Filhos: `Termo`, `+`, `Termo`
- Desce por parênteses e produções vazias.
- Útil para a prova matemática, mas pesada em memória.

AST (Árvore Abstrata)

- Foca apenas no que importa.
- Nó Operador: `+`
- Filho Esq: `5`
- Filho Dir: Nó Operador `*`
- Sub-filhos do `*`: `2` e `x`
- *Os parênteses desapareceram! A hierarquia da árvore já garante a precedência.*

Gramáticas Ambíguas

Uma gramática é **ambígua** se ela pode produzir mais de uma árvore de análise (Parse Tree) válida para a **mesma** string de tokens.

Por que isso é um desastre?

- Diferentes árvores geram diferentes códigos assembly (ordem de execução).
- **Matemática ambígua:** Na string `2 + 3 * 4`, se a árvore avaliar a soma primeiro, o resultado é 20. Se avaliar a multiplicação primeiro, é 14. O compilador não pode ter "dúvidas".

O Exemplo Clássico: Dangling Else

Outro problema clássico de gramáticas reais é o "Else Balançante".

```
if (x > 0) if (y > 0) x = 1; else y = 1;
```

A quem pertence o `else`?

1. Ao primeiro `if (x > 0)`?
2. Ao segundo `if (y > 0)`?

A gramática padrão permite ambas as árvores. Para resolver, a linguagem C, Java e Pascal estabeleceram uma regra semântica: *"O else sempre se liga ao if mais próximo (mais interno) não resolvido"*.

1. Parsing Top-Down (Descendente)

- Constrói a árvore da **raiz** até as **folhas**.
- Processa tentando "adivinhar" qual produção aplicar olhando o próximo token (*Lookahead*).
- Vantagem: Fácil de programar manualmente em C/Java (*Recursive Descent*). Compiladores modernos como GCC e Clang (C++) usam parsers Top-Down escritos à mão.
- Restrição: Não tolera recursão à esquerda (ex: $E \rightarrow E + T$).

2. Parsing Bottom-Up (Ascendente)

- Constrói a árvore das **folhas** empilhando tokens até chegar na **raiz**.
- Realiza operações de "Shift" (empilhar) e "Reduce" (transformar lado direito no esquerdo).
- Vantagem: Extremamente poderoso, suporta uma classe maior de gramáticas e lida bem com recursão à esquerda.
- Restrição: Difícil de codificar à mão. Exige ferramentas geradoras baseadas em matrizes complexas de estados (ex: Yacc, Bison, CUP).

Resumo da Aula

- O Parser garante a estrutura (sintaxe) da linguagem, enquanto o Scanner garante o vocabulário (léxico).
- Precisamos de Gramáticas Livres de Contexto (GLC) porque Expressões Regulares não lidam com estruturas aninhadas e memória empilhada.
- A AST (Árvore Sintática Abstrata) é a estrutura de dados mais importante gerada nesta fase para o restante do compilador.
- Gramáticas ambíguas são inaceitáveis. Problemas de precedência e o *Dangling Else* devem ser resolvidos na elaboração das regras.

Referências para Estudo

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

Dúvidas?

Obrigado!

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