Compiladores

Aula 05: Análise Léxica, Expressões Regulares e Autômatos

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br
CEFET-MG - Campus Timóteo

Fevereiro de 2026

1 Objetivos

- Compreender a posição da análise léxica na estrutura do compilador (Front-end).
- Definir e diferenciar os conceitos fundamentais: Tokens, Padrões e Lexemas.
- Entender o gargalo de I/O na compilação e aplicar técnicas de Bufferização e Sentinelas.
- Conhecer as Expressões Regulares como formalismo matemático para especificação de padrões.
- Compreender o pipeline de conversão de Expressões Regulares para Autômatos Finitos (Thompson e Subconjuntos).
- Diferenciar os estilos de arquitetura de Scanners (Ad-hoc, Dirigido por Tabela e Direto) e o algoritmo de *Maximal Munch*.

2 O Papel do Analisador Léxico

O processo de compilação é tradicionalmente dividido em duas grandes fases: o **Front-end** (análise da estrutura da linguagem fonte) e o **Back-end** (síntese e geração de código alvo). O primeiro passo de todo esse processo é a **Análise Léxica**, frequentemente chamada de *Scanning*.

A separação da Análise Léxica da Análise Sintática (Parser) não é por acaso. Ela simplifica o design geral, garante maior eficiência (já que as técnicas léxicas baseadas em autômatos são muito mais rápidas do que técnicas sintáticas baseadas em pilhas) e isola detalhes de portabilidade, como o conjunto de caracteres do código-fonte (ASCII, UTF-8), em um único componente.

O Analisador Léxico recebe como entrada o fluxo de caracteres bruto do código-fonte e tem como saída um fluxo estruturado de **Tokens**. Durante essa leitura, o Scanner também realiza tarefas auxiliares essenciais: ignora espaços em branco e tabulações, remove comentários, rastreia linhas e colunas para emissão de mensagens de erro e realiza as primeiras inserções na Tabela de Símbolos.

2.1 Tokens, Padrões e Lexemas

Para entender o funcionamento da análise léxica, é crucial não confundir três conceitos fundamentais:

Token: É um símbolo terminal na gramática da linguagem. Representa uma classe lógica (ex: `ID`, `NUM`, `IF`, `PLUS`). Frequentemente, carrega um *atributo* com seu valor semântico.

Padrão (Pattern): É a regra formal que descreve como a sequência de caracteres de um token pode ser formada. É especificado matematicamente via Expressões Regulares.

Lexema: É a sequência exata, concreta e real de caracteres extraída do código-fonte que casou com um padrão estrutural.

Por exemplo, na expressão em C `taxa = base * 1.15;`, a palavra `taxa` é o lexema que corresponde ao padrão de identificadores, gerando o token `ID`.

A Tabela **??** ilustra essa distinção para uma atribuição típica:

Table 1: Exemplos de Tokens, Lexemas e Padrões.

Lexema	Token	Padrão (Informal)
`if`	`IF`	`if`
`=`	`ASSIGN`	`=`
`count`	`ID`	Letra seguida de letras ou dígitos
`3.14`	`NUM`	Sequência de dígitos com um ponto decimal
`"erro"`	`STRING`	Caracteres entre aspas duplas

3 O Gargalo de Leitura e Bufferização

Na engenharia de compiladores, o Analisador Léxico é a única fase que lê o código-fonte caractere por caractere. Fazer uma chamada de sistema operacional (`read()`) para o disco rígido a cada caractere tornaria a compilação inviável devido ao alto custo de I/O. A análise léxica não otimizada pode consumir até 50% do tempo total de compilação.

Para resolver isso, utilizamos a **Bufferização em Memória**. Lemos grandes blocos de dados do disco (ex: 4KB) de uma só vez para um *array* na memória primária.

Para evitar que um lexema seja "cortado" no meio ao final de um buffer, emprega-se o esquema de **Pares de Buffers** (*Double Buffering*). Mantemos dois ponteiros principais: `inicio_lexema` e `avanco` (*forward*). Quando o ponteiro de avanço chega ao fim do primeiro buffer, carregamos a continuação do código no segundo buffer, garantindo uma leitura contínua.

Para otimizar ainda mais o laço de leitura, introduzimos o conceito de **Sentinelas**. Colocamos um caractere especial de Fim de Arquivo (`EOF`) no final de cada buffer. Isso permite que o algoritmo verifique o fim do buffer apenas quando encontra o `EOF`, reduzindo os testes condicionais dentro do laço de leitura pela metade.

O algoritmo de leitura com sentinelas funciona da seguinte forma:

```
Avança ponteiro 'avanco'
Se (*avanco == EOF) {
    Se (avanco no final do buffer 1) {
        Recarrega buffer 2
        avanco = início do buffer 2
    } Senão se (avanco no final do buffer 2) {
        Recarrega buffer 1
        avanco = início do buffer 1
    } Senão {
        Terminar scanning (Fim de Arquivo Real)
    }
}
```

4 Expressões Regulares (ER)

Como o compilador entende a regra de formação de um identificador? Utilizamos a ferramenta matemática padrão para especificação de padrões: as Expressões Regulares.

Seja Σ um alfabeto. As expressões regulares sobre Σ e as linguagens que elas denotam são definidas de forma indutiva:

- **Base:** A palavra vazia ϵ é uma ER denotando o conjunto $\{\epsilon\}$. Para cada símbolo $a \in \Sigma$, a é uma ER denotando o conjunto $\{a\}$.
- **Indução (Operadores):** Se r e s são ERs:
  - **União ($r|s$):** Denota a união das duas linguagens.
  - **Concatenação (rs):** Denota uma string de r seguida por uma string de s.
  - **Fecho de Kleene (r^*):** Denota zero ou mais concatenações de r.

Para facilitar o design prático, utilizamos extensões (açúcar sintático) como o Fecho Positivo (r^+ para uma ou mais ocorrências), Opcionalidade ($r?$ para zero ou uma) e Classes de Caracteres ($[a-z]$).

Exemplo clássico: um número de ponto flutuante genérico pode ser expresso como: inteiro -> digito+ fracao -> . digito+ expoente -> (E|e) (+|-)? inteiro numero -> inteiro fracao? expoente?

Outros exemplos comuns de padrões léxicos:

- **Identificador:** `[a-zA-Z_][a-zA-Z0-9_]*`
- **String Literal:** `"` $[^"]*$"($Simplificado: aspasseguidasdequalquercoisaquenãosejaaspa$).
- **Comentário de Linha (C++):** `//` $[^]*$

5 De ER para Autômatos

Embora as Expressões Regulares sejam ótimas para a especificação humana, os computadores precisam de uma estrutura de estados algorítmica. A solução é traduzir a ER para **Autômatos Finitos**. http://googleusercontent.com/image$_{c}$*ontent*/288

O processo de tradução ocorre num pipeline muito bem definido:

1 **ER para AFND (Algoritmo de Thompson):** Transforma a expressão em um Autômato Finito Não-Determinístico (AFND). Ele processa a árvore sintática da ER montando blocos pré-definidos para união, concatenação e fecho. O AFND gerado possui transições vazias (ϵ) e múltiplos caminhos, sendo fácil de gerar, mas custoso para simular na CPU.

2 **AFND para AFD (Construção de Subconjuntos):** Remove o não-determinismo agrupando estados. O ϵ-closure calcula todos os estados alcançáveis sem consumir entrada. O resultado é um Autômato Finito Determinístico (AFD), onde cada transição é única para um dado caractere.

Matematicamente, um AFD é definido como uma 5-upla $M = (Q, \Sigma, \delta, q_0, F)$, onde:

- Q é um conjunto finito de estados.
- Σ é o alfabeto de entrada.
- $\delta : Q \times \Sigma \to Q$ é a função de transição.
- $q_0 \in Q$ é o estado inicial.
- $F \subseteq Q$ é o conjunto de estados finais (aceitação).

O Trade-off de Desempenho: Embora o AFND seja modular e fácil de construir via Thompson, sua simulação exige manter um conjunto de estados ativos, o que é lento. O AFD, por outro lado, é extremamente rápido (uma única consulta à tabela por caractere), mas sua construção pode sofrer com a *explosão de estados* (no pior caso, 2^n estados onde n é o número de estados do AFND).

3 **Minimização de Estados:** Reduz o número de estados do AFD para poupar memória na implementação final (Algoritmo de Hopcroft).

6 Arquitetura de Scanners e Resolução de Conflitos

Uma vez com o AFD construído, como ele é convertido em código executável? Existem três estilos principais de implementação:

1 **Ad-hoc (Manual):** Uso de grandes blocos de `while` e `switch/case`. É fácil de criar para linguagens simples, mas difícil de manter.

2 **Dirigido por Tabela (*Table-Driven*):** O AFD vira uma matriz bidimensional (linhas como estados, colunas como o alfabeto). O laço central usa a matriz genérica para transitar. É a técnica utilizada por ferramentas geradoras modernas como JFlex.

3 **Codificação Direta (*Direct-Coded*):** Traduz estados para instruções e rótulos (`goto`), produzindo o código mais veloz possível para a arquitetura de máquina atual, contudo, ilegível para humanos.

6.1 A Regra do Maximal Munch

Quando ocorre uma ambiguidade léxica (por exemplo, a entrada `ifStatement` sendo lida, que começa com `if`), os analisadores léxicos utilizam a regra do **Maior Casamento** (*Maximal Munch*). O scanner consome a maior quantidade possível de caracteres que formem um lexema válido.

Considere os tokens `IF` (padrão `if`), `ID` (padrão `[a-z]+`), `EQUALS` (padrão `==`) e `ASSIGN` (padrão `=`).

- Para a entrada `if`, o scanner poderia casar com `IF` ou `ID`. Como ambos têm o mesmo tamanho, a prioridade de declaração decide (geralmente palavras reservadas vêm antes).
- Para a entrada `if_var`, o scanner não para em `if`, pois continuar lendo resulta em um lexema maior para `ID`.
- Para a entrada `==`, o scanner não retorna dois tokens `ASSIGN`, mas sim um único `EQUALS`, pois o padrão associado ao `EQUALS` casa com dois caracteres, vencendo o padrão de um caractere.

Se houver empate exato de tamanho, a ordem de declaração na especificação dita o vencedor.

6.2 Recuperação de Erros (Panic Mode)

Quando um caractere que não faz parte do alfabeto ou de nenhuma estrutura é encontrado (ex: um caractere `@` fora de uma string em C), ocorre um erro léxico. A abordagem mais comum em compiladores de produção é a recuperação em *Panic Mode* (Modo Pânico), onde o scanner deleta sucessivos caracteres invasores da entrada até encontrar o início de um lexema válido, avisa o usuário sobre o caractere inválido, e continua a compilação, o que permite o relato de múltiplos erros numa única execução.

7 Referências

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. (O Livro do Dragão - Capítulos 3).
- Cooper, K., & Torczon, L. (2011). *Engineering a Compiler*. 2ª Edição. (Capítulo 2 - Scanners).
- Louden, K. C. (2004). *Compiladores: Princípios e Práticas*. (Capítulo 2).