

---

# Compiladores

## Aula 04: Análise Léxica: Conceitos e Implementação

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

Março de 2026

### 1 Objetivos

- Compreender formalmente o papel da Análise Léxica como escudo inicial do Front-End.
- Dominar a separação acadêmica entre os conceitos de *Token*, *Padrão* e *Lexema*.
- Revisar os fundamentos algébricos (Alfabeto, Cadeia, Linguagem) e Expressões Regulares (ER).
- Entender os desafios de engenharia mecânica na construção de um scanner: gerenciamento de I/O por Buffers Duplos e *Lookahead*.
- Aprender a construir analisadores léxicos manuais (Ad-hoc) lidando com transição de estados.
- Introduzir formalmente geradores de analisadores modernos (JFlex) e sua sintaxe profissional.

### 2 O Papel do Analisador Léxico (Scanner)

No vasto \*pipeline\* de um compilador, a \*\*Análise Léxica\*\* (frequentemente chamada de \*Scanner\*) tem um trabalho simultaneamente monótono e crucial: é a única fase que interage diretamente com o caos puro dos arquivos em disco do usuário.

O compilador vê o código fonte `int x = 5;` não como uma instrução lógica, mas como uma matriz unidimensional ininterrupta de bytes (códigos ASCII ou UTF-8): [105, 110, 116, 32, 120, 32, 61, 32, 53, 59]. A função do scanner é mastigar essa fileira estúpida de bytes, devorar tudo aquilo que for irrelevante e cuspir um vetor de dados lógicos que a máquina entende.

As principais responsabilidades incluem:

- 1 Agrupamento Lógico:** Isolar aglomerados de caracteres contíguos que formam uma unidade semântica indivisível (ex: os bytes 'i', 'f' juntos valem uma entidade matemática, o IF).

---

**2 Expurgo de Impurezas (Stripping):** O scanner varre e descarta silenciosamente dezenas de milhares de espaços em branco, tabulações (`\t`), quebras de linha (`\n`) e blocos extensos de documentação e comentários (`/* ... */`). O restante do compilador nunca saberá que comentários existiram.

**3 Rastreabilidade e Log:** Como o Scanner é quem "anda" pela matriz original, é dele a responsabilidade exclusiva de manter a contagem de *linha* e *coluna* atuais. Quando a Análise Semântica (fase 3) detectar um erro lógico e quiser gritar "Erro na linha 530", essa informação veio guardada e transportada pelo scanner desde o segundo inicial.

### 3 Os Três Pilares: Token, Lexema e Padrão

O jargão de compiladores exige rigor extremo ao usar essas três palavras, frequentemente confundidas por programadores iniciantes.

- **Lexema:** É o dado cru. A sequência `*real*` e palpável de caracteres escrita no arquivo fonte. Por exemplo, no código `x = 3.14`, os caracteres "x" e "3.14" são dois lexemas isolados distintos no mundo físico.
- **Padrão (Pattern):** É a fórmula abstrata (frequentemente uma Expressão Regular) matemática que define o que é válido. Exemplo: O padrão para definir um número inteiro no C++ diz: "Um ou mais dígitos entre 0 e 9, sem letras".
- **Token:** É o "objeto matemático" retornado pelo scanner ao parser. Um token é comumente representado como uma tupla estrita: `<Nome-da-Classe, Atributo-Opcional>`. O "Nome-da-Classe" avisa o Parser qual regra gramatical invocar. O "Atributo-Opcional" guarda o valor semântico exato.

**Estudo de Caso:** Analisando a declaração `taxa_extra := 10 + 20;`

Lexema Físico	Padrão Obedecido	Token Gerado ( <code>&lt;&lt;Classe, Atributo&gt;&gt;</code> )
<code>taxa_extra</code>	<code>[a-zA-Z][a-zA-Z0-9_]*</code>	<code>&lt;ID, ponteiro para Hash Table&gt;</code>
<code>:=</code>	<code>:=</code>	<code>&lt;ASSIGN, -&gt;</code>
<code>10</code>	<code>[0-9]+</code>	<code>&lt;NUM, valor inteiro 10&gt;</code>
<code>+</code>	<code>\+</code>	<code>&lt;PLUS, -&gt;</code>
<code>20</code>	<code>[0-9]+</code>	<code>&lt;NUM, valor inteiro 20&gt;</code>
<code>;</code>	<code>;</code>	<code>&lt;SEMI, -&gt;</code>

Table 1: A trilogia Lexema-Padrão-Token na prática.

Observe que, do ponto de vista do *Parser* (a próxima fase), os tokens 10 e 20 são matematicamente indistinguíveis; ambos são agrupados na mesma classe sintática `NUM`. Apenas a fase semântica, que olhará os atributos, saberá que um valia dez e o outro vinte.

---

## 4 Fundamentos Teóricos: Expressões Regulares

O compilador não inventa o que é uma variável; ele segue normas da Teoria da Computação. O mecanismo principal para modelar padrões na Fase Léxica são as **Expressões Regulares (ER)**.

### 4.1 Preliminares Algébricas

A matemática das ERs repousa sobre três definições primordiais:

- 1 Alfabeto ( $\Sigma$ ):** Um conjunto finito e não-vazio de símbolos. No C++, o alfabeto são todos os caracteres ASCII válidos.
- 2 Cadeia (String):** Uma sequência finita de símbolos tirados de um alfabeto. O comprimento da cadeia  $s$  é denotado  $|s|$ . O símbolo vazio (tamanho zero) é denotado por  $\epsilon$  (Epsilon).
- 3 Linguagem ( $L$ ):** Qualquer conjunto enumerável de cadeias sobre  $\Sigma$ . O C++ inteiro pode ser modelado matematicamente como uma "Linguagem" de cadeias incrivelmente longas.

### 4.2 As Operações de Expressões Regulares

Uma Expressão Regular  $r$  denota uma linguagem  $L(r)$ . As ERs são formadas usando apenas três operações brutas universais:

- **União ( $r | s$ ):** A linguagem gerada ou pertence a  $r$  ou pertence a  $s$ . (Operador OR).
- **Concatenação ( $r \cdot s$ ):** Uma cadeia de  $r$  seguida exatamente por uma cadeia de  $s$ .
- **Fecho de Kleene ( $r^*$ ):** A repetição arbitrária. A concatenação de  $r$  consigo mesmo, zero ou mais vezes. (Inclui o vazio  $\epsilon$ ).

### 4.3 Extensões Sintáticas Profissionais

Como escrever  $A|B|C|D|E\dots$  para todo o alfabeto consumiria páginas e páginas, a indústria de software (ferramentas como GNU Flex, JFlex e regex do Python) adotou abreviações (extensões práticas):

- **Fecho Positivo ( $r^+$ ):** Uma ou mais vezes. Matematicamente equivale a  $r(r^*)$ . Evita que o padrão vazio seja aceito.
- **Opcional ( $r^?$ ):** O padrão pode aparecer uma vez ou nenhuma. Equivale a  $r | \epsilon$ .
- **Classes de Caracteres ( $[a-z]$ ):** Representa a escolha | massiva de toda a cadeia do caractere 'a' ao 'z'. O acento circunflexo ( $[\hat{a}]$ ) nega a classe (casando com qualquer coisa \*exceto\* 'a').

A aplicação destas extensões simplifica barbaaramente a vida do projetista do compilador. Um Padrão para "Número em Notação Científica" do Pascal ( $1.5E-10$ ) é matematicamente assustador, mas numa expressão comercial é reduzido para:  $[0-9]^+ (\backslash. [0-9]^+)^?$  (E  $[+-]^?$   $[0-9]^+)^?$ .

---

## 5 Os Desafios da Implementação Manual (Ad-hoc)

Na construção manual do scanner, esbarramos imediatamente na física do hardware e do disco rígido.

### 5.1 A Fúria do Lookahead e o Algoritmo "Maior Casamento"

Quando o scanner começa a ler, ele consome caracteres um por um de uma esteira invisível. Se ele consome a letra 'i' e depois a letra 'f', ele deve emitir o Token IF? A resposta mecânica é: **ainda não**. Como o scanner sabe que o desenvolvedor não estava digitando uma variável chamada `if_condicao`?

O algoritmo regente dos Scanners é o **Maior Casamento (\*Maximal Munch\*)**: O scanner continua lendo avidamente caractere após caractere do arquivo *até que o padrão se quebre e torne-se inválido*. Apenas quando o padrão é corrompido, ele emite o último estado válido.

Esse fenômeno obriga o scanner a ler um caractere a mais no disco rígido para "ter certeza" de que o lexema acabou. Por exemplo, ao ler `>`, ele precisa ler o próximo caractere. Se for `=`, o Token é `GEQ ( $\geq$ )`. Se for `A`, ele deve emitir o token `GT ( $>$ )` e **devolver (pushback)** a letra `A` para a fita de leitura, pois aquele caractere não pertence ao `GT`.

Essa visão antecipada de 1 caractere na fita sem consumi-la definitivamente é chamada de **Lookahead**.

### 5.2 Dual-Buffering: Resolvendo a Lentidão de Disco

Ler do Disco Rígido (HD ou SSD) 1 caractere de cada vez usando chamadas brutas ao Sistema Operacional (`read(1)`) destruiria a performance de um compilador. Para resolver a leitura ineficiente e os recuos de lookahead constantes, os compiladores utilizam uma estrutura de Array duplo (Dual-Buffering).

Eles criam na memória RAM dois *arrays* de cerca de 4096 bytes (4KB). O compilador pede blocos imensos de 4KB ao disco e foca-se na RAM. O Scanner usa dois ponteiros mecânicos: um marca o **Início do Lexema**, e outro o **Olho de Leitura (Lookahead)** (que avança caractere por caractere). Para evitar checar milhares de vezes se o array acabou (o que custaria pesados ciclos de instrução `if` na CPU), insere-se um caractere terminador falso no fim de cada metade do array (um caractere "Sentinela", como EOF). Isso dobra a velocidade de leitura global do compilador.

### 5.3 Recuperação de Erro e Loops Infinitos

Se um programador em C colocar um caractere exótico '@' no meio do código, e não houver Expressão Regular configurada para casar o '@', o Scanner manual não tem para onde pular. Se ele retornar erro sem avançar os ponteiros da fita, na próxima requisição do parser, ele lerá o @ de novo, gerando um **Loop Infinito** fatal que travará o compilador da máquina do programador. A regra suprema de recuperação léxica é: ao se deparar com lixo, emita uma mensagem severa de **Erro Léxico (Invalid Char)** e, mais importante, force mecanicamente o consumo (*avanço do ponteiro*) desse caractere maldito para fora da fita, retomando a leitura do caractere limpo subsequente.

---

## 6 Automação Profissional: O Gerador JFlex

A dor de gerenciar buffers duplos gigantescos, arrays sentinelas, estados de variáveis e ponteiros de *\*Lookahead\** à mão levou a ciência a criar os geradores automáticos de Scanner. A grande maravilha do **\*\*Lex\*\*** (C) e **\*\*JFlex\*\*** (Java) é que eles automatizam brutalmente a engenharia de *Front-End*.

Você não programa *\*ifs\** ou *buffers*. Você cria um arquivo declarativo (`.jflex`) ditando apenas e puramente os **Padrões (Expressões Regulares)**. O JFlex embute um compilador interno baseado na teoria algorítmica de construções de subconjuntos (Construção de Thompson), converte suas regex declaradas matematicamente para NFA, reduz para DFA, gera e te entrega uma classe `Lexer.java` contendo loops otimizados e tabelas de estado hiper-rápidas para compor o seu projeto final.

### 6.1 Sintaxe de uma Especificação JFlex Avançada

O arquivo fonte do JFlex segue três fatias seccionadas pelo operador `%%`:

```
1  /* ----- */
2  /* SECAO 1: Codigo de Usuario          */
3  /* ----- */
4  package compilador.scanner;
5  import compilador.tokens.Token;
6
7  %%
8
9  /* ----- */
10 /* SECAO 2: Macros e Opcoes           */
11 /* ----- */
12 %public
13 %class MiniPascalScanner
14 %type Token
15 %unicode
16 %line
17 %column
18
19 /* Definindo padroes matematicos reutilizaveis (Macros) */
20 Letra = [a-zA-Z]
21 Digito = [0-9]
22 AlfaNumerico = {Letra} | {Digito}
23 Identificador = {Letra} {AlfaNumerico}*
24 NumeroInteiro = {Digito}+
25 EspacoBranco = [ \t\r\n]+
26
27 /* Estado especial para absorver grandes comentarios em bloco */
28 %state COMENTARIO
29
30 %%
31
32 /* ----- */
```

```

33  /* SECAO 3: Regras Lexicas (Acoes)  */
34  /* ----- */
35
36  <YYINITIAL> {
37
38      /* Ignora completamente todo o fluxo que for puramente lixo
39         */
40      {EspacoBranco} { /* Silencio (faz nada e segue lendo) */ }
41
42      /* Palavras-Chave de Alta Prioridade (acima dos
43         identificadores) */
44      "if"           { return new Token(TokenType.IF, yyline); }
45      "then"         { return new Token(TokenType.THEN, yyline); }
46      "while"        { return new Token(TokenType.WHILE, yyline); }
47
48      /* Operadores e Pontuacoes */
49      ":@"          { return new Token(TokenType.ASSIGN, yyline); }
50      ">="          { return new Token(TokenType.GEQ, yyline); }
51      ";"            { return new Token(TokenType.SEMI, yyline); }
52
53      /* Expressoes Curinga Genericas */
54      {NumeroInteiro} { return new Token(TokenType.NUM, yytext());
55         }
56      {Identificador} { return new Token(TokenType.ID, yytext()); }
57
58      /* Gatilho para transicao de estado (Comentarios longos) */
59      "/*"           { yybegin(COMENTARIO); }
60
61      /* Casamento final - Qualquer coisa nao capturada pelas ERs
62         acima (ERRO) */
63      [^]            { throw new RuntimeException("Erro lexico: "
64         + yytext()); }
65  }
66
67  <COMENTARIO> {
68      "*/"           { yybegin(YYINITIAL); } /* Volta pra leitura
69         normal */
70      [^]            { /* Ignora tudo que for lido dentro do
71         comentario */ }
72  }

```

Notamos nesta especificação de nível industrial do JFlex que a precedência é garantida de cima para baixo. A regra literal "if" está acima do curinga geral {Identificador}. Logo, se a palavra "if" for lida, ela não cai na malha geral de nome de variável; ela se consolida como TokenType.IF. O recurso formidável de "Estados Ocultos" (%state COMENTARIO) retira do Scanner o fardo de criar expressões regulares inviáveis de infinitos wildcards e o transforma num miniautômato elegante e performático que despreza a imensa sujeira dos /\* comentários longos \*/ perfeitamente.

---

## 7 Referências

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Pearson. (Capítulo 3)
- Cooper, K., & Torczon, L. (2011). *Engineering a Compiler*. Morgan Kaufmann. (Capítulo 2)
- Klein, Gerwin. *The JFlex Manual*. Disponível em: <https://jflex.de>.