Compiladores

Aula 04: Análise Léxica: Conceitos, FLEX e Pascal

Prof. Aléssio Miranda Júnior
alessio@cefetmg.br

CEFET-MG - Campus Timoteo
Dep. Engenharia de Computação

Março de 2026

Sumário

Objetivos da Aula

- Compreender a posição da análise léxica na estrutura do compilador.
- Definir os conceitos fundamentais: tokens, padrões e lexemas.
- Conhecer as **Expressões Regulares** como formalismo para tokens.
- Aprender a usar o **JFlex** (gerador léxico) com **Java**.
- Construir passo a passo um **analisador léxico para Pascal simplificado**.
- Integrar o lexer gerado com código Java.

1. Estrutura de um Compilador

O processo de compilação é dividido em duas grandes fases:

Fase de Análise (Front-End)

1. Programa Fonte
2. **Analisador Léxico**
3. Analisador Sintático
4. Analisador Semântico

Fase de Síntese (Back-End)

5. Gerador de Código Intermediário
6. Otimizador de Código
7. Gerador de Código Objeto
8. Programa Objeto

Posição do Analisador Léxico

É a **primeira fase** do compilador, fazendo a interface direta com o código-fonte.

O que é Análise Léxica?

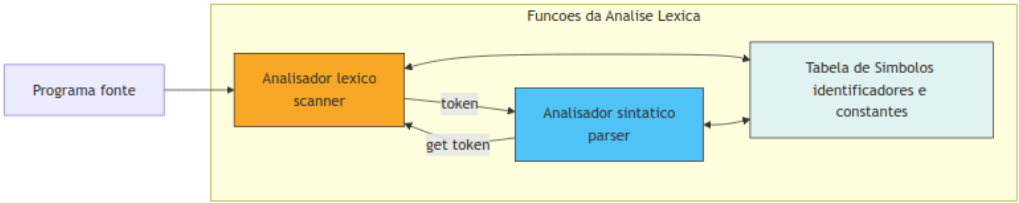
Definição Ampla

Análise léxica é o processo de analisar a entrada de linhas de caracteres (tal como o código-fonte de um programa de computador) e produzir uma sequência de símbolos chamados **"símbolos léxicos"** (lexical tokens), ou somente **"símbolos"** (tokens), que podem ser manipulados mais facilmente por um parser (leitor de saída).

Verificação de Alfabeto

A Análise léxica é a forma de verificar determinado alfabeto. Quando analisamos uma palavra, podemos definir se existe ou não algum caractere que não faz parte do nosso alfabeto (ou um alfabeto inventado). **É a primeira etapa de um compilador**, vindo logo antes da análise sintática.

Fluxo de Dados: Léxico e Sintático



Nota: Este é o fluxo básico do léxico.

2. Definição de Análise Léxica

A análise léxica é a **primeira etapa** de um compilador. Suas principais funções são:

- **Leitura da entrada**: Lê o fluxo de caracteres do código-fonte.
- **Agrupamento**: Produz uma sequência de **tokens** (símbolos léxicos).
- **Filtragem**: Remove espaços em branco, quebras de linha e comentários.
- **Verificação**: Valida se os caracteres pertencem ao alfabeto da linguagem.
- **Relatório de erros**: Informa caracteres inválidos (erros léxicos).

Atenção

O analisador léxico **não verifica estrutura gramatical** — isso é papel do analisador sintático.

3. Tokens, Padrões e Lexemas

É essencial distinguir estes três conceitos fundamentais:

Token

Par $\langle$*nome, atributo*$\rangle$. O **nome** é um símbolo abstrato; o **atributo** carrega informações extras.
Ex.: $\langle$`id`, "taxa"$\rangle$, $\langle$`num`, 100$\rangle$, $\langle$`if`$\rangle$

Lexema

A **sequência real de caracteres** no código-fonte que casa com o padrão de um token.
Ex.: `taxa`, `100`, `if`

Padrão (Pattern)

A **regra** (expressão regular) que descreve o conjunto de lexemas de um token.
Ex.: `[a-zA-Z][a-zA-Z0-9]*` para identificadores.

Mini-Pascal: Exemplo 1

Estrutura Básica

```
1 program Exemplo1;
2 var
3   x, y: integer;
4 begin
5   x := 10;
6   y := 20;
7   if x < y then
8     writeln('Menor');
9 end.
```

Mini-Pascal: Exemplo 2

Repetição

```
program Fatorial;
var
  n, i, res: integer;
begin
  n := 5; i := 1; res := 1;
  while i <= n do
  begin
    res := res * i;
    i := i + 1;
  end;
  writeln(res);
end.
```

Exemplo de Classificação de Tokens

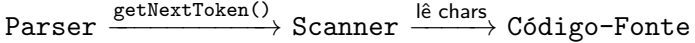
Entrada: `if (a < 100) x := 5;`

Lexema	Token	Categoria
`if`	`IF`	Palavra-chave
`(`	`LPAREN`	Símbolo
`a`	`ID,"a"`	Identificador
`<`	`LT`	Operador
`100`	`NUM,100`	Número
`)`	`RPAREN`	Símbolo

Lexema	Token	Categoria
`x`	`ID,"x"`	Identificador
`:=`	`ASSIGN`	Operador
`5`	`NUM,5`	Número
`;`	`SEMI`	Símbolo

O Papel do Scanner na Compilação

O **scanner** (analisador léxico) e o **parser** (analisador sintático) interagem:



- O scanner age **sob demanda** do parser.
- Mantém a **linha atual** para mensagens de erro.
- Consulta e popula a **Tabela de Símbolos**.

Vantagem de separar léxico de sintático

Simplifica o design, permite reutilização e melhora as mensagens de erro.

4. Expressões Regulares (ER)

Utilizadas para descrever os padrões de tokens de forma compacta:

Operações Básicas:

- **Concatenação**: ab (a seguido de b)
- **Alternativa**: $a \mid b$ (a ou b)
- **Kleene**: a^* (zero ou mais a)
- **Positiva**: a^+ (um ou mais a)
- **Opcional**: $a?$ (zero ou um a)
- **Classe**: $[a\text{-}z]$ (faixa de chars)
- **Ponto**: `.` (qualquer char)

Exemplos:

- `[a-zA-Z_]` — letra ou _
- `[0-9]+` — inteiro positivo
- `[0-9]+\.[0-9]+` — real
- `[a-zA-Z_][a-zA-Z0-9_]*` — ID
- `"[^"]*"` — string literal
- `//[^\n]*` — comentário linha

ERs para Tokens de Pascal

Token	Expressão Regular	Descrição
ID	`[a-zA-Z][a-zA-Z0-9_]*`	Identificador
INTEGER	`[0-9]+`	Inteiro
REAL	`[0-9]+\.[0-9]+`	Real
STRING	`'[^']*'`	String (aspas simples)
ASSIGN	`:=`	Atribuição
RANGE	`..`	Intervalo (`1..10`)
COMMENT	`{[^}]*}`	Comentário `{...}`
WS	`[ \t\n\r]+`	Espaços (ignorar)

ER → AF: Exemplo 1 (Identificadores)

Expressão Regular: `[a-zA-Z][a-zA-Z0-9]*`

- **S0** → **S1**: Transição ao ler uma letra.
- **S1**: Estado final (aceitação). Continua em S1 para qualquer letra ou dígito.

ER $\rightarrow$ AF: Exemplo 2 (Números Inteiros)

Expressão Regular: `[0-9]+`

- Garante que pelo menos um dígito seja lido ($+$).
- O estado S1 consome todos os dígitos subsequentes.

ER → AF: Exemplo 3 (Atribuição)

Expressão Regular: :=

- Exemplo de lexema fixo (sem repetição).
- O scanner deve ver o caractere : e logo após obrigatoriamente o =.

5. Implementando um Scanner Manualmente

Como transformamos um DFA em código Java? Usamos um laço de leitura e estruturas de decisão!

```java
public Token nextToken() {
    // 1. Pular espacos em branco e comentarios
    pularEspacos();

    char c = readChar();

    // 2. Identificar o estado inicial
    if (Character.isLetter(c)) {
        return lerIdentificadorOuPalavraChave(c);
    } else if (Character.isDigit(c)) {
        return lerNumero(c);
    } else if (c == ':') {
        // Lookahead: espiar o proximo char
        if (peek() == '=') {
            readChar(); // consome o '='
            return new Token(TokenType.ASSIGN, ":=");
        }
        return new Token(TokenType.COLON, ":");
    }
    // ...
}
```

Scanner Manual: O laço principal

A implementação manual exige um controle rigoroso do ponteiro de leitura:

- **readChar()**: consome o caractere atual e avança.
- **peek()**: espia o próximo caractere **sem consumi-lo** (essencial para operadores como `:=`, `<=`).
- **Atributos**: `posicao`, `linha`, `coluna`.

Dica para o TP

Para consolidar a lógica dos autômatos, implementaremos o laço de repetição e as transições de estado "na unha". Isso desenvolve uma base algorítmica fortíssima!

6. Tabela de Símbolos

A Tabela de Símbolos (T.S.) armazena informações sobre os identificadores encontrados no código.

O que guardar?

- Nome (lexema)
- Categoria (vairável, constante, função)
- Tipo (`integer`, `boolean`)
- Escopo e Endereço

Implementação:

- Geralmente uma **Tabela Hash**.
- Busca rápida $O(1)$.
- Palavras reservadas podem ser inseridas previamente.

Tabela de Símbolos: Exemplo Prático

Para o trecho: `var saldo: integer;`

Lexema	Token	Tipo	Escopo
`integer`	`T_INTEGER`	—	global (keyword)
`saldo`	`ID`	`integer`	local

```java
// Estrutura basica em Java
Map<String, Symbol> tabela = new HashMap<>();
tabela.put("saldo", new Symbol("saldo", TokenType.ID, "integer"));
```

7. Para onde vão os Tokens? A AST

O Analisador Sintático (Parser) solicita os tokens ao Lexer e constrói uma **Árvore Sintática Abstrata (AST)**.

- A AST é a representação em memória do seu programa.
- Descarta detalhes sintáticos irrelevantes e foca na **estrutura lógica**.
- Base para análise semântica e geração de código.

Código: `x := 10 + 5` $\rightarrow$ **Atribuição**(ID(x), **Soma**(Num(10), Num(5)))

Padrão de Projeto: Composite

Na O.O. (Java), implementamos a AST usando o padrão **Composite**.

- **AstNode**: Classe base abstrata.
- **Expression**: Nós que representam valores (soma, números).
- **Command**: Nós que representam ações (if, while, atribuição).

Conexão com o Trabalho Prático

Entender como os tokens formam essa estrutura é vital para a Etapa 1 do seu roteiro!

8. Geradores de Analisadores Léxicos

Na prática profissional, utilizamos ferramentas chamadas **Geradores Léxicos** (como Lex, Flex ou **JFlex**).

- **Como funciona?** Você define as regras via **Expressões Regulares**.
- A ferramenta gera automaticamente o código Java que implementa o DFA.
- **Vantagem:** Produtividade e robustez.

Estrutura JFlex

Dividido em 3 seções permanentes: `/* Codigo Java */ %% /* Opcoes e Macros */ %% /* Regras */`

Exemplo Pascal no JFlex (Resumido)

```
/* Opcoes */
%class PascalLexer
%type Token
%%
/* Regras (Apenas Exemplos) */
"if"        { return new Token(TokenType.IF); }
"program"   { return new Token(TokenType.PROGRAM); }
{ID}        { return new Token(TokenType.ID, yytext()); }
{INTEGER}   { return new Token(TokenType.INT, yytext()); }
":="        { return new Token(TokenType.ASSIGN); }
```

Referências

- **JFlex**: `https://jflex.de` — Documentação oficial do gerador de lexers para Java.
- Wirth, N. *Algorithms + Data Structures = Programs* — referência original do Pascal.
- *The JFlex Manual* ve. 1.9.1 — guia completo de directivas e regras.

Obrigado!

Email: alessio@cefetmg.br

"A análise léxica é o alicerce — sem ela, o compilador não enxerga nada."