ENTENDENDO UM LEXER EM JAVA: UM GUIA COM BOAS PRÁTICAS

Se você está aprendendo Java e quer saber como um compilador ou interpretador processa código-fonte, este artigo é para você. Vamos explorar um **lexer**, uma peça fundamental de um **Pratt parser**, que transforma texto (como uma expressão matemática) em unidades chamadas **tokens**. Imagine o lexer como um bibliotecário que organiza um livro em capítulos, parágrafos e palavras, preparando o conteúdo para o parser interpretar.

O código usa recursos modernos do Java, como **records** e **interfaces funcionais**, e segue boas práticas como **SOLID** (responsabilidades únicas), **KISS** (simplicidade) e **DRY** (evitar repetição). A seguir está uma explicação passo a passo das classes implementadas, discutindo alguns conceitos que podem ser desafiadores, como enums, lambdas e records. Ao final, você entenderá como o lexer funciona e como aplicá-lo em projetos.

O que é um Lexer?

Um lexer (ou analisador léxico) converte uma string de entrada (como `x + 2`) em **tokens**, unidades significativas como variáveis (`x`), operadores (`+`) ou números (`2`). Pense nisso como separar ingredientes (cenoura, batata, cebola) antes de cozinhar. O lexer organiza o código-fonte em tokens para o parser processar.

O lexer suporta: - Números (ex.: `123`) - Variáveis (ex.: `x`, `soma`) - Operadores (ex.: `+`, `-`, `*`, `/`, etc.) - Parênteses (`(`, `)`) e colchetes (`[`, `]`) - Espaços em branco (ignorados) - Fim do arquivo (`EOF`) - Tokens inválidos (`UNKNOWN`, definido mas não usado)

Vamos analisar cada componente do código com detalhes.

Estrutura do Código

O lexer é composto por cinco partes principais:

1. **TokenType**: Enum que define os tipos de tokens.
2. **Token**: Representa um token com tipo e valor.
3. **Rule**: Define regras para identificar caracteres e criar tokens.
4. **LexerState**: Interface que gerencia o estado do lexer (posição, caractere atual, etc.).
5. **Lexer** e **LexerFactory**: Classes que implementam a tokenização e criam o lexer com regras predefinidas.

1. Enum TokenType

O enum `TokenType` lista os tipos de tokens que o lexer reconhece. É como uma lista de categorias para classificar cada pedaço do texto.

```
public enum TokenType {
    NUMBER,       // Números (ex.: "123").
    VARIABLE,     // Variáveis (ex.: "x", "abc").
    OPERATOR,     // Operadores (ex.: "+", "-", "*", "/", ".", "!", "=", "?", ":").
    LPAREN,       // Parêntese esquerdo "(".
    RPAREN,       // Parêntese direito ")".
    LBRACKET,     // Colchete esquerdo "[".
    RBRACKET,     // Colchete direito "]".
    EOF,          // Fim da entrada.
    UNKNOWN       // Token inválido (para erros).
}
```

- **O que faz?** Define categorias para tokens, como `NUMBER` para números ou `OPERATOR` para operadores.
- **Por que enum?** Enums são ideais para valores fixos, funcionando como opções predefinidas, como escolher “azul” ou “vermelho” em um menu.
- **Nota**: O tipo `UNKNOWN` está definido, mas não usado, pois a regra correspondente em `LexerFactory` está comentada. Caracteres desconhecidos causam uma exceção.
- **Dica**: Enums são como etiquetas que identificam o papel de cada parte do texto. Por exemplo, `"123"` recebe a etiqueta `TokenType.NUMBER`.
- **Boa prática (KISS)**: O enum é claro, com nomes descritivos e comentários explicativos.

2. Classe Token

A classe `Token` é um **record**, recurso do Java (a partir da versão 14) que cria classes imutáveis de forma concisa. Pense em um token a menor unidade significativa de uma entrada textual, que possui um **tipo** e um **conteúdo** associado. Eles não têm significado semântico ainda, essa é a função do parser que será explicado em outra oportunidade.

```
public record Token(TokenType type, String value) {
    @Override
    public String toString() {
        return type == TokenType.EOF ? "EOF" : String.format("%s(%s)", type, value);
    }
}
```

- **O que faz?** Armazena um token com tipo (`TokenType`, como `NUMBER`) e conteúdo (`String`, como `"123"`).
- **Por que record?** Records são imutáveis e geram automaticamente métodos como `equals`, `hashCode` e `toString`. São como fichas fixas.
- **Método `toString`**: Formata o token, como `NUMBER(123)` para números ou `"EOF"` para o fim.
- **Dica**: O operador ternário (`?:`) no `toString` é uma forma compacta de dizer: “Se o tipo é EOF, mostre ‘EOF’; senão, mostre tipo e valor”. Equivalente a:

```
        if (type == TokenType.EOF) {
            return "EOF";
        } else {
            return type + "(" + value + ")";
        }
```

- **Boa prática (DRY)**: O `record` elimina a necessidade de escrever getters ou `equals` manualmente.

3. Interface LexerState

A interface `LexerState` define métodos para gerenciar a entrada. É como um conjunto de instruções para um bibliotecário: "Pegue a próxima página", "Pule páginas em branco", etc.

```
public interface LexerState {
    char getCurrentChar();
    void advance();
    boolean hasNextChar();
    void skipWhitespace();
}
```

- **Métodos**:
  - `getCurrentChar()`: Retorna o caractere atual (ex.: `'x'`).
  - `advance()`: Avança para o próximo caractere.
  - `hasNextChar()`: Verifica se há mais caracteres.
  - `skipWhitespace()`: Pula espaços, tabulações e quebras de linha.
- **Dica**: Interfaces são contratos que definem o que uma classe deve fazer. O `Lexer` implementa esses métodos para controlar a leitura.
- **Boa prática (SOLID)**: A interface segue a **segregação de interfaces**, incluindo apenas métodos essenciais.

4. Classe Rule

A classe `Rule` é um `record` que define como identificar caracteres e criar tokens. É como uma receita: "Se o ingrediente é uma cenoura, corte-a assim".

```
import java.util.function.Predicate;
import java.util.function.Function;

public record Rule(Predicate<Character> condition, Function<LexerState, Token> action) {
    public boolean appliesTo(char c) {
        return condition.test(c);
    }

    public Token apply(LexerState state) {
        return action.apply(state);
    }
}
```

- **Componentes**:
  - `condition`: Um `Predicate<Character>` que verifica se um caractere corresponde à regra (ex.: "É um dígito?").
  - `action`: Uma `Function<LexerState, Token>` que cria um token (ou `null` para regras como pular espaços).
- **O que é `Predicate` e `Function`?** São **interfaces funcionais** do Java 8:
  - `Predicate<Character>` decide "sim" ou "não" (ex.: `Character::isDigit` para dígitos).
  - `Function<LexerState, Token>` transforma o estado em um token (ex.: cria `Token(TokenType.NUMBER, "1")`).
- **Métodos**:
  - `appliesTo`: Verifica se a regra se aplica.
  - `apply`: Executa a ação, retornando um token.
- **Dica**: O `::` em `Character::isDigit` é uma abreviação para usar o método `isDigit`. É como delegar uma tarefa sem escrever o código completo.
- **Boa prática (KISS)**: A classe é simples, delegando a lógica para `condition` e `action`.

Exemplo: Regra para números: - Condição: `Character::isDigit` (verdadeiro para `'1'`, falso para `'x'`). - Ação: Lê dígitos (ex.: `'123'`) e cria `NUMBER(123)`.

5. Classe LexerFactory

A classe `LexerFactory` cria instâncias do `Lexer` com regras predefinidas. É como uma fábrica que monta uma máquina com instruções específicas.

```
import parser.prat_parser.model.Token;
import parser.prat_parser.model.TokenType;
import java.util.List;

public class LexerFactory {
    public static Lexer createDefaultLexer(String input) {
        var rules = List.of(
            new Rule(Character::isWhitespace, state -> {
                state.skipWhitespace();
                return null;
            }),
            new Rule(Character::isDigit, state -> {
                var number = new StringBuilder();
                while (state.hasNextChar() && Character.isDigit(state.getCurrentChar())) {
```

```
                    number.append(state.getCurrentChar());
                    state.advance();
                }
                return new Token(TokenType.NUMBER, number.toString());
            }),
            new Rule(Character::isLetter, state -> {
                var variable = new StringBuilder();
                while (state.hasNextChar() && Character.isLetter(state.getCurrentChar())) {
                    variable.append(state.getCurrentChar());
                    state.advance();
                }
                return new Token(TokenType.VARIABLE, variable.toString());
            }),
            new Rule(c -> "+-*/.!:=?:".contains(String.valueOf(c)), state -> {
                var op = String.valueOf(state.getCurrentChar());
                state.advance();
                return new Token(TokenType.OPERATOR, op);
            }),
            new Rule(c -> c == '(', state -> {
                state.advance();
                return new Token(TokenType.LPAREN, "(");
            }),
            new Rule(c -> c == ')', state -> {
                state.advance();
                return new Token(TokenType.RPAREN, ")");
            }),
            new Rule(c -> c == '[', state -> {
                state.advance();
                return new Token(TokenType.LBRACKET, "[");
            }),
            new Rule(c -> c == ']', state -> {
                state.advance();
                return new Token(TokenType.RBRACKET, "]");
            })
        );
        return new Lexer(input, rules);
    }
}
```

- **O que faz?** Cria um `Lexer` com regras para espaços, números, variáveis, operadores, parênteses e colchetes.
- **Regras**:
  - **Espaços**: Pula espaços e retorna `null`.
  - **Números**: Lê dígitos (ex.: `'123'`) e cria `NUMBER`.
  - **Variáveis**: Lê letras (ex.: `'x'`, `'soma'`) e cria `VARIABLE`.
  - **Operadores**: Reconhece `+`, `-`, `*`, etc., e cria `OPERATOR`.
  - **Parênteses e colchetes**: Reconhece `(`, `)`, `[`, `]` e cria tokens correspondentes.
- **Dica**:
  - `List.of` cria uma lista imutável, garantindo segurança.
  - Lambdas (ex.: `c -> "+-*/.!:=?:".contains(String.valueOf(c))`) são funções curtas que verificam condições, como se um caractere é um operador.
  - `StringBuilder` é eficiente para construir strings, evitando concatenações lentas com `+`.
- **Nota**: A regra comentada para `UNKNOWN` indica que caracteres não reconhecidos causam exceções, não tokens `UNKNOWN`.
- **Boa prática (SOLID)**: A fábrica encapsula a criação do `Lexer`, seguindo a **responsabilidade única**.

Exemplo: Para `"x + 2"`: - `'x'`: Regra de letras → `VARIABLE(x)` - `' '`: Regra de espaços → Ignorado - `'+'`: Regra de operadores → `OPERATOR(+)` - `'2'`: Regra de dígitos → `NUMBER(2)`

6. Classe Lexer

A classe `Lexer` implementa a tokenização, lendo a entrada e gerando tokens. É como o bibliotecário que lê o livro e anota os capítulos.

```
import parser.prat_parser.model.Token;
import parser.prat_parser.model.TokenType;
import java.util.ArrayList;
import java.util.List;

public class Lexer implements LexerState {
    private final String input;
    private final List<Rule> rules;
    private int position;
    private char currentChar;

    public Lexer(String input, List<Rule> rules) {
        this.input = input;
        this.rules = new ArrayList<>(rules); // Cópia defensiva
        this.position = 0;
        this.currentChar = input.isEmpty() ? '\0' : input.charAt(0);
    }

    public List<Token> tokenize() {
        var tokens = new ArrayList<Token>();
```

```
        while (currentChar != '\0') {
            boolean matched = false;
            for (var rule : rules) {
                if (rule.appliesTo(currentChar)) {
                    var token = rule.apply(this);
                    if (token != null) {
                        tokens.add(token);
                    }
                    matched = true;
                    break;
                }
            }
            if (!matched) {
                throw new IllegalStateException("No rule matched for character: " + currentChar);
            }
        }
        tokens.add(new Token(TokenType.EOF, ""));
        return tokens;
    }

    @Override
    public char getCurrentChar() {
        return currentChar;
    }

    @Override
    public void advance() {
        position++;
        currentChar = position < input.length() ? input.charAt(position) : '\0';
    }

    @Override
    public boolean hasNextChar() {
        return currentChar != '\0';
    }

    @Override
    public void skipWhitespace() {
        while (hasNextChar() && Character.isWhitespace(getCurrentChar())) {
            advance();
        }
    }

    public String getInput() {
        return input;
    }

    public int getPosition() {
        return position;
    }

    public void printTokens() {
        List<Token> tokens = tokenize();
        tokens.forEach(System.out::println);
    }
}
```

- **Atributos**:
  - `input`: String de entrada (ex.: `"x + 2"`).
  - `rules`: Lista de regras.
  - `position`: Índice do caractere atual.
  - `currentChar`: Caractere atual (ou `'\0'` no fim).
- **Construtor**:
  - Inicializa atributos e faz **cópia defensiva** da lista de regras, protegendo contra alterações externas.
  - Define `currentChar` como o primeiro caractere ou `'\0'` se a entrada for vazia.
- **Método `tokenize`**:
  - Lê até o fim (`currentChar != '\0'`).
  - Aplica regras ao caractere atual, adiciona tokens (se não `null`) e marca correspondência.
  - Lança exceção se nenhuma regra corresponder.
  - Adiciona `EOF` ao final.
- **Métodos de `LexerState`**:
  - `getCurrentChar`: Retorna o caractere atual.
  - `advance`: Avança, usando `'\0'` para o fim.
  - `hasNextChar`: Verifica se há mais caracteres.
  - `skipWhitespace`: Pula espaços.
- **Métodos extras**:
  - `getInput`, `getPosition`: Úteis para depuração.
  - `printTokens`: Exibe tokens (provavelmente para testes).

- **Dica**:
  - O `'\0'` é um marcador de fim, indicando o término da entrada.
  - A cópia defensiva (`new ArrayList<>(rules)`) protege contra mudanças externas, como copiar uma lista de tarefas.
  - `forEach(System.out::println)` imprime tokens de forma moderna, equivalente a um laço `for`.
- **Boa prática (SOLID)**: O `Lexer` tem uma única responsabilidade (tokenizar), e `LexerState` garante modularidade.

Exemplo: Para `"x + 2"`: 1. `'x'`: Regra de letras → `VARIABLE(x)` 2. `' '`: Regra de espaços → Ignorado 3. `'+'`: Regra de operadores → `OPERATOR(+)` 4. `'2'`: Regra de dígitos → `NUMBER(2)` 5. Fim → `EOF`

Saída:

```
VARIABLE(x)
OPERATOR(+)
NUMBER(2)
EOF
```

Exemplo de uso do Lexer

Agora que você entendeu como o `Lexer` funciona internamente, talvez esteja se perguntando:

Como eu uso esse Lexer na prática?

A resposta é simples. Você pode criar um lexer usando a `LexerFactory`, que já vem com todas as regras configuradas. Veja um exemplo funcional abaixo:

```java
public class Main {
    public static void main(String[] args) {
        // Expressão que será analisada
        String expressao = "x + 2";

        // Cria o lexer com as regras padrão
        Lexer lexer = LexerFactory.createDefaultLexer(expressao);

        // Tokeniza e imprime cada token
        lexer.printTokens();
    }
}
```

Neste exemplo, a string `"x + 2"` será processada e os tokens resultantes impressos no console:

```
VARIABLE(x)
OPERATOR(+)
NUMBER(2)
EOF
```

Explicando o método tokenize passo a passo

O método `tokenize` percorre os caracteres da entrada e aplica as regras uma a uma, conforme o trecho abaixo:

```java
public List<Token> tokenize() {
    var tokens = new ArrayList<Token>();

    // Enquanto o caractere atual não for o marcador de fim
    while (currentChar != '\0') {
        boolean matched = false;

        // Testa cada regra da lista
        for (var rule : rules) {
            // Verifica se a regra se aplica ao caractere atual
            if (rule.appliesTo(currentChar)) {
                // Executa a regra, passando o estado atual do Lexer (this)
                var token = rule.apply(this);

                // Adiciona o token, se a regra não retornou null
                if (token != null) {
                    tokens.add(token);
                }

                matched = true;
                break; // uma regra já tratou o caractere
            }
        }

        // Nenhuma regra conseguiu tratar o caractere atual
        if (!matched) {
            throw new IllegalStateException("No rule matched for character: " + currentChar);
        }
    }

    // Adiciona token de fim de arquivo
    tokens.add(new Token(TokenType.EOF, ""));
```

```
        return tokens;
    }
```

Por que usamos this em rule.apply(this)?

A classe `Lexer` implementa a interface `LexerState`, que define os métodos:

```
char getCurrentChar();
void advance();
boolean hasNextChar();
void skipWhitespace();
```

Como `Lexer` implementa `LexerState`, ele pode ser passado diretamente para métodos que esperam essa interface. É exatamente o caso de `rule.apply(LexerState state)`. Quando usamos `rule.apply(this)`, estamos injetando **a própria instância do `Lexer`** como dependência, **através da interface**, e não da implementação concreta e usando seus métodos implementados como o `advance()`.

Essa é uma forma prática e idiomática de injeção de dependência:

- A `Rule` conhece apenas a interface `LexerState`, e não a implementação `Lexer`.
- Isso favorece a manutenção, testabilidade e flexibilidade do código.
- Caso quiséssemos testar as `Rules` isoladamente, bastaria criar uma classe de teste que implementa `LexerState`.

Esse loop é o núcleo do processo de análise léxica: cada caractere é processado por uma regra que sabe interpretá-lo. A separação entre `appliesTo()` (verifica) e `apply()` (executa) é uma aplicação direta do **Princípio da Responsabilidade Única (SRP)**, pois a verificação e a ação são claramente distintas.

DIAGRAMA DE CLASSES LEXER