CEFSA – Centro Eduacional da Fundação Salvador Arena

FESA – Faculdade Engenheiro Salvador Arena

São Bernardo do Campo, 26 de agosto de 2025

Enzo Brito Alves de Oliveira – RA: 082220040

Heitor Santos Ferreira – RA: 081230042

Engenharia de Computação – Sétimo Semestre – EC7

Compiladores

Professor: Israel Florentino

Analisador Lexico usando Jflex – Documentação da especificação da linguagem e testes de caso

SÃO PAULO

2025

1. Nome da linguagem

Emis é o nome atribuído à linguagem de programação proposta neste documento. A escolha do nome não é arbitrária: “Emis” deriva de uma simplificação fonética da palavra “emissão”, remetendo à ideia de geração, expressão e comunicação de instruções computacionais de forma clara e acessível.

A linguagem Emis foi concebida com o propósito de servir como uma ferramenta didática e funcional, voltada à introdução de conceitos fundamentais de compiladores, análise léxica e sintática, bem como à prototipagem de algoritmos simples. Seu design prioriza a legibilidade, a concisão e a previsibilidade sem abrir mão da expressividade necessária para representar estruturas lógicas e operacionais comuns em linguagens de propósito geral.

Emis se posiciona como uma linguagem de alto nível, imperativa e estruturada, com sintaxe inspirada em paradigmas clássicos, mas com simplificações que favorecem o aprendizado e a implementação de analisadores léxicos e parsers. A nomenclatura dos elementos da linguagem, como palavras reservadas e operadores, foi cuidadosamente selecionada para refletir semânticas intuitivas e minimizar ambiguidades durante o processo de análise.

Em suma, o nome “Emis” representa não apenas uma identidade linguística, mas também um compromisso com a clareza, a didática e a eficiência na comunicação entre o programador e a máquina.

2. Conceito e proposito

A linguagem **Emis** foi concebida como uma ferramenta de abstração computacional voltada à experimentação, ensino e prototipagem de algoritmos em ambientes acadêmicos e laboratoriais. Seu propósito central é oferecer uma sintaxe acessível e uma semântica previsível, permitindo que estudantes, pesquisadores e desenvolvedores possam explorar os fundamentos da construção de linguagens de programação, com ênfase em análise léxica, parsing e geração de código.

Emis adota um paradigma **imperativo e estruturado**, com influências de linguagens clássicas como Pascal e C, mas com simplificações que favorecem a clareza e a legibilidade. A linguagem foi desenhada para ser minimalista em sua gramática, mas suficientemente expressiva para representar estruturas de controle, operações aritméticas, manipulação de dados e modularização por meio de funções.

Um dos pilares conceituais da linguagem é a **transparência sintática**, ou seja, a capacidade de o código refletir de forma direta e intuitiva a lógica que se

pretende implementar. Isso é especialmente relevante em contextos educacionais, onde a linguagem serve como ponte entre o raciocínio algorítmico e sua formalização computacional.

Além disso, Emis busca ser uma linguagem **lexicamente robusta**, com regras bem definidas para identificação de tokens, tratamento de erros e rastreamento de posição, o que a torna ideal para projetos de construção de compiladores e interpretadores.

Em suma, Emis não se propõe a competir com linguagens de uso industrial, mas sim a ocupar um espaço estratégico no ecossistema de linguagens didáticas e experimentais, promovendo o entendimento profundo dos mecanismos que sustentam a comunicação entre humanos e máquinas por meio da programação.

3. Palavras reservadas

A linguagem **Emis** define um conjunto restrito e bem delimitado de palavras reservadas, cuja função é representar instruções, estruturas de controle e elementos sintáticos fundamentais. Essas palavras são reconhecidas de forma literal pelo analisador léxico e não podem ser utilizadas como identificadores, nomes de variáveis ou funções, sob pena de erro de compilação.

A escolha das palavras reservadas em Emis foi orientada por critérios de clareza semântica, concisão e familiaridade com paradigmas de programação estruturada. Cada palavra foi selecionada para refletir diretamente sua função no código, reduzindo ambiguidades e facilitando o processo de leitura e escrita por parte do programador.

A seguir, apresenta-se a lista completa das palavras reservadas da linguagem Emis, acompanhada de uma breve descrição de sua finalidade:

- **inicio** – Marca o ponto inicial de execução de um programa.
- **fim** – Indica o término da execução do programa.
- **var** – Declara uma ou mais variáveis no escopo atual.
- **funcao** – Define uma função nomeada, com parâmetros e corpo.
- **retorna** – Especifica o valor de retorno de uma função.
- **se** – Inicia uma estrutura condicional.
- **senao** – Define o bloco alternativo de uma condição.
- **enquanto** – Cria um laço de repetição baseado em uma condição booleana.

- **para** – Estabelece um laço com controle explícito de inicialização, condição e incremento.
- **imprime** – Realiza a saída de dados para o console ou terminal.

Todas as palavras reservadas são **case-sensitive**, ou seja, devem ser escritas exatamente como definidas, respeitando letras minúsculas. O uso incorreto de capitalização ou a tentativa de redefinição dessas palavras como identificadores resultará em erro léxico ou sintático, conforme o estágio de análise em que forem detectadas.

4. Operadores

A linguagem **Emis** dispõe de um conjunto enxuto e funcional de operadores, cuidadosamente selecionados para atender às necessidades básicas de manipulação de dados, controle de fluxo e avaliação de expressões. Os operadores são elementos sintáticos que realizam operações sobre operandos, podendo ser valores literais, variáveis ou expressões compostas.

A categorização dos operadores em Emis segue uma lógica semântica clara, dividida em quatro grupos principais: **aritméticos**, **relacionais**, **lógicos** e **de atribuição**. Cada grupo possui operadores com comportamento bem definido e precedência estabelecida, permitindo a construção de expressões complexas com previsibilidade e consistência.

A seguir, apresenta-se a classificação dos operadores disponíveis na linguagem Emis, acompanhada de suas respectivas descrições:

4.1. Operadores Aritméticos

Utilizados para realizar operações matemáticas entre operandos numéricos:

- **+** – Soma dois valores.
- **-** – Subtrai o segundo valor do primeiro.
- ***** – Multiplica dois valores.
- */* – Divide o primeiro valor pelo segundo, retornando ponto flutuante.

Operadores Relacionais

Avaliam a relação entre dois operandos e retornam um valor booleano:

- **==** – Verifica se os operandos são iguais.
- **!=** – Verifica se os operandos são diferentes.

- **<** – Verifica se o primeiro operando é menor que o segundo.
- **>** – Verifica se o primeiro operando é maior que o segundo.
- **<=** – Verifica se o primeiro operando é menor ou igual ao segundo.
- **>=** – Verifica se o primeiro operando é maior ou igual ao segundo.

4.2. Operadores Lógicos

Permitem a construção de expressões booleanas compostas:

- **&&** – Conjunção lógica (E): retorna verdadeiro se ambos os operandos forem verdadeiros.
- **||** – Disjunção lógica (OU): retorna verdadeiro se ao menos um dos operandos for verdadeiro.
- **!** – Negação lógica: inverte o valor booleano do operando.

4.3. Operador de Atribuição

Responsável por associar um valor a uma variável:

- **=** – Atribui o valor do operando à direita à variável à esquerda.

Todos os operadores em Emis são **binários**, exceto o operador de negação lógica (!), que é **unário**. A linguagem respeita uma ordem de precedência entre operadores, sendo os aritméticos avaliados antes dos relacionais, seguidos pelos lógicos, e por fim o operador de atribuição. Parênteses podem ser utilizados para alterar explicitamente a ordem de avaliação das expressões.

5. Identificadores

Na linguagem **Emis**, os identificadores representam os nomes atribuídos a variáveis, funções, parâmetros e demais elementos definidos pelo programador. Eles são essenciais para a construção de programas legíveis e organizados, permitindo a associação semântica entre valores e estruturas lógicas.

A definição de identificadores em Emis segue regras léxicas claras e restritivas, com o objetivo de evitar ambiguidade e garantir a integridade da análise sintática. A seguir, são descritas as regras formais para a formação de identificadores:

- Um identificador **deve iniciar obrigatoriamente por uma letra (a–z, A–Z)** ou pelo caractere de sublinhado (_).

- Após o primeiro caractere, o identificador pode conter **letras, dígitos (0–9)** e sublinhados, em qualquer ordem.
- Não há suporte a caracteres especiais, acentuação ou espaços em branco dentro de identificadores.
- O comprimento máximo permitido para um identificador é de **32 caracteres**, sendo recomendável o uso de nomes descritivos e concisos.
- A linguagem é **sensível a maiúsculas e minúsculas (case-sensitive)**, ou seja, os identificadores valor, Valor e VALOR são tratados como distintos.
- Identificadores **não podem coincidir com palavras reservadas** da linguagem, mesmo que estejam grafados com capitalização diferente.

Exemplos válidos de identificadores incluem:

- contador
- soma_total
- _resultadoFinal
- x1, y2, temp_var

Exemplos inválidos incluem:

- 1valor (inicia com dígito)
- se (palavra reservada)
- total@ (caractere inválido)
- valor total (contém espaço)

O analisador léxico é responsável por validar a conformidade dos identificadores com essas regras, emitindo mensagens de erro apropriadas em caso de violação. Essa validação ocorre antes da análise sintática, garantindo que apenas tokens válidos sejam processados nas etapas subsequentes da compilação ou interpretação.

6. Literais

Na linguagem **Emis**, os *literais* representam valores constantes que são diretamente interpretados pelo analisador léxico, sem necessidade de avaliação adicional. Eles são fundamentais para a construção de expressões, atribuições e estruturas de controle, permitindo ao programador especificar dados explícitos no código fonte.

Os literais em Emis são classificados em categorias distintas, cada uma com regras léxicas próprias e semântica bem definida. A seguir, apresenta-se a descrição formal das principais categorias de literais reconhecidas pela linguagem:

6.1. Literais inteiros

Representam valores numéricos inteiros, positivos ou negativos, sem parte decimal.

- São compostos exclusivamente por dígitos decimais (0–9).
- Podem opcionalmente ser precedidos por um sinal de negativo (-).
- Não são permitidos separadores de milhar, espaços ou zeros à esquerda (exceto no valor 0).
- Exemplos válidos: 0, 42, -7, 1024
- Exemplos inválidos: 04, --5, 1 000

6.2. Literais de ponto flutuante

Representam números reais com parte decimal, podendo incluir notação científica.

- Devem conter ao menos um ponto decimal (.) entre os dígitos.
- Podem incluir um expoente, indicado por e ou E, seguido de um número inteiro (com sinal opcional).
- Exemplos válidos: 3.14, 0.0, -2.5, 6.022e23, 1.0E-3
- Exemplos inválidos: .5 (sem dígito antes do ponto), 5. (sem dígito após o ponto), 2e (expoente incompleto)

6.3. Literais booleanos

Representam valores lógicos utilizados em expressões condicionais e estruturas de controle.

- Emis reconhece dois literais booleanos: verdadeiro e falso
- São tratados como palavras-chave e não podem ser redefinidos.
- Exemplos válidos: se verdadeiro, retorna falso

6.4. Literais de caracteres e strings

Representam sequências de caracteres alfanuméricos e símbolos, utilizados para manipulação textual.

- Strings são delimitadas por aspas simples ('texto') ou aspas duplas ("texto").
- Caracteres especiais podem ser escapados com barra invertida (\n, \t, \\, \", etc.).
- Strings multilinha são delimitadas por três aspas duplas (""""texto"""").
- Exemplos válidos: "Olá, mundo!", 'a', "Linha\nNova", """Texto\nmultilinha"""

Todos os literais são tratados como tokens atômicos durante a análise léxica, e sua validação ocorre com base em padrões textuais bem definidos. A linguagem não permite literais compostos ou concatenados implicitamente; qualquer operação entre literais deve ser explicitada por meio de operadores ou funções.

7. Strings

Na linguagem **Emis**, *strings* são sequências de caracteres utilizadas para representar dados textuais. Elas desempenham papel fundamental em operações de entrada e saída, manipulação de mensagens, armazenamento de nomes, entre outras funcionalidades que envolvem conteúdo textual.

A definição e o tratamento de strings em Emis foram projetados para oferecer flexibilidade, legibilidade e suporte a expressões textuais complexas, sem comprometer a simplicidade da análise léxica. A seguir, são descritas as regras formais que regem a construção e o reconhecimento de strings na linguagem:

7.1. Delimitação

Strings podem ser delimitadas por:

- **Aspas simples** ('texto') – recomendadas para textos curtos ou caracteres isolados.
- **Aspas duplas** ("texto") – utilizadas para textos gerais.
- **Três aspas duplas** (""""texto"""") – utilizadas para *strings multilinha*, permitindo quebras de linha internas sem necessidade de escape.

7.2. Conteúdo e caracteres especiais

O conteúdo de uma string pode incluir:

- Letras, dígitos, símbolos e espaços.
- Caracteres especiais representados por **sequências de escape**, iniciadas por barra invertida (\). Exemplos:
- \n – quebra de linha
- \t – tabulação
- \\ – barra invertida literal
- \" – aspas duplas literal
- \' – aspas simples literal

O analisador léxico reconhece essas sequências como parte da string e as interpreta conforme sua função semântica.

7.3. Regras Lexicas

- Strings devem ser delimitadas corretamente, sem quebra de linha não escapada (exceto em strings multilinha).
- O conteúdo entre delimitadores é tratado como um único token, mesmo que contenha espaços ou símbolos.
- Strings não podem ser interrompidas abruptamente; a ausência do delimitador de fechamento resulta em erro léxico.
- O uso de aspas dentro da string exige escape apropriado para evitar conflito com o delimitador externo.

Strings em Emis são tratadas como **literais textuais imutáveis**, e sua manipulação é realizada por meio de operadores ou funções específicas (definidas pelo programador ou pela biblioteca padrão, se houver). O suporte a strings multilinha e a escapes torna a linguagem adequada para aplicações que exigem expressividade textual, como geração de relatórios, mensagens de erro e scripts interativos.

8. Comentários

Na linguagem **Emis**, os *comentários* são elementos não executáveis inseridos no código com o objetivo de documentar, explicar ou descrever trechos da lógica implementada. Eles são ignorados pelo analisador sintático e não influenciam na

execução do programa, sendo processados exclusivamente pelo analisador léxico.

O suporte a comentários em Emis foi projetado para oferecer flexibilidade ao programador, permitindo tanto anotações breves quanto blocos explicativos extensos. A linguagem reconhece dois tipos distintos de comentários: **comentários de linha única** e **comentários de bloco**, cada um com regras específicas de delimitação e escopo.

8.1. Comentários de linha única

- São iniciados com a sequência de dois caracteres barra (//).
- Todo o conteúdo após // até o final da linha é ignorado pelo compilador.
- São ideais para observações rápidas, marcações temporárias ou desativações pontuais de código.
- Não podem se estender por múltiplas linhas.

8.2. Comentários de bloco

- São delimitados pela sequência /* para abertura e */ para fechamento.
- Podem abranger múltiplas linhas e incluir qualquer tipo de conteúdo textual.
- São úteis para documentação extensa, explicações técnicas ou desativação de trechos maiores de código.
- Emis oferece suporte a comentários de bloco aninhados, permitindo que blocos de comentários sejam inseridos dentro de outros blocos sem causar erro léxico.

8.3. Regras léxicas

- Comentários devem ser corretamente delimitados; a ausência do delimitador de fechamento (*/) em comentários de bloco resulta em erro léxico.
- Comentários não podem ser iniciados dentro de literais de string, pois isso causaria ambiguidade na análise.
- O conteúdo dos comentários não é armazenado nem processado após a análise léxica, sendo descartado integralmente.
- Comentários podem ser utilizados em qualquer parte do código, exceto dentro de tokens indivisíveis (como literais ou identificadores)

O uso adequado de comentários é incentivado como prática de programação responsável, contribuindo para a legibilidade, manutenção e compreensão do código por terceiros ou pelo próprio autor em revisões futuras.

9. Recursos adicionais

A linguagem **Emis**, embora minimalista em sua essência, incorpora um conjunto de *recursos adicionais* que ampliam sua funcionalidade e a tornam especialmente adequada para fins didáticos, experimentais e de prototipagem. Esses recursos não são obrigatórios em linguagens convencionais, mas foram intencionalmente incluídos para enriquecer o processo de análise léxica, facilitar o desenvolvimento de ferramentas auxiliares e promover uma experiência de programação mais expressiva e controlada.

A seguir, são descritos os principais recursos complementares oferecidos pela linguagem Emis:

9.1. Rastreamento de posição lexical

Cada token reconhecido pelo analisador léxico é acompanhado de metadados que indicam sua **posição exata no código fonte**, incluindo número da linha e coluna inicial.
Esse recurso é essencial para:

- Geração de mensagens de erro precisas e informativas.
- Ferramentas de depuração e análise estática.
- Ambientes de desenvolvimento com realce de sintaxe e navegação contextual.

9.2. Suporte a Strings de multilinha

Emis permite a definição de *strings multilinha* por meio de três aspas duplas (`"""texto"""`), possibilitando a inclusão de quebras de linha internas sem necessidade de escape.
Esse recurso é útil para:

- Geração de textos formatados.
- Inserção de mensagens longas ou documentação embutida.
- Criação de scripts interativos com saída estruturada.

9.3. Comentários alinhados

Diferentemente de muitas linguagens tradicionais, Emis oferece suporte nativo a **comentários de bloco aninhados**, permitindo que blocos de comentários sejam inseridos dentro de outros sem causar conflito léxico.
Isso facilita:

- Documentação técnica em níveis hierárquicos.
- Desativação seletiva de trechos de código durante testes.
- Inserção de observações internas em blocos já comentados.

9.4. Tratamento de erros lexicos

O analisador léxico de Emis foi projetado para identificar e reportar **erros léxicos com mensagens descritivas**, incluindo:

- Tipo de erro (caractere inválido, token malformado, delimitador ausente).
- Posição exata no código.
- Sugestões de correção, quando aplicável.

Esse mecanismo contribui para o aprendizado e para a correção rápida de falhas durante o desenvolvimento.

9.5. Extensabilidade controlada

Embora a linguagem seja definida com um conjunto fixo de tokens e estruturas, sua gramática foi desenhada de forma modular, permitindo **extensões futuras** com novos tipos de dados, operadores ou construções sintáticas, sem comprometer a integridade do núcleo léxico.
Isso torna Emis uma base sólida para projetos de evolução linguística, como:

- Implementação de bibliotecas padrão.
- Criação de dialetos especializados.
- Experimentação com paradigmas alternativos (funcional, orientado a objetos).

Em conjunto, esses recursos adicionais posicionam Emis como uma linguagem não apenas funcional, mas também *pedagogicamente rica*, oferecendo ao programador e ao pesquisador um ambiente controlado, expressivo e extensível para explorar os fundamentos da linguagem e da computação.

10. Regras léxicas – descrição textual

As *regras léxicas* da linguagem **Emis** definem os critérios pelos quais o analisador léxico reconhece, classifica e segmenta os elementos do código fonte em unidades sintáticas denominadas *tokens*. Essas regras são descritas de forma textual, sem o uso de expressões regulares ou código formal, com o objetivo de facilitar a compreensão e a implementação manual ou semiautomática do analisador.

A seguir, apresenta-se a descrição detalhada das regras léxicas que regem o funcionamento do analisador da linguagem Emis:

10.1. Reconhecimento de palavras reservadas

Palavras reservadas são reconhecidas por **correspondência literal exata**. O analisador compara sequências de caracteres com a lista pré-definida de palavras reservadas.

- A correspondência é **case-sensitive**: se é válido, Se não é.
- Palavras reservadas não podem ser precedidas ou seguidas por caracteres alfanuméricos sem separação (ex: se1 não é reconhecido como se).

10.2. Identificadores

Um identificador é reconhecido quando:

- O primeiro caractere é uma **letra** (a–z, A–Z) ou **sublinhado** (_).
- Os caracteres subsequentes podem incluir letras, dígitos (0–9) ou sublinhados.
- O identificador não coincide com nenhuma palavra reservada.
- O comprimento máximo é de 32 caracteres.

10.3. Literais numéricos

- Inteiros: sequência de dígitos decimais, opcionalmente precedida por sinal negativo (-).
- Ponto flutuante: sequência de dígitos com ponto decimal obrigatório, podendo incluir notação científica (e ou E seguido de inteiro).
- O analisador rejeita números com múltiplos pontos, zeros à esquerda (exceto 0), ou expoentes incompletos.

10.4. Strings

- Delimitadas por aspas simples (') ou duplas ("), com conteúdo textual entre os delimitadores.
- Strings multilinha são delimitadas por três aspas duplas (""").
- Caracteres especiais são representados por sequências de escape iniciadas por barra invertida (\).
- O analisador ignora o conteúdo interno como um único token, desde que os delimitadores estejam corretamente fechados.

10.5. Comentários

- **Linha única**: iniciados por //, ignoram todo o conteúdo até o fim da linha.
- **Bloco**: iniciados por /* e encerrados por */, podendo abranger múltiplas linhas.
- Comentários de bloco podem ser **aninhados**, e o analisador mantém uma pilha de abertura/fechamento para garantir a integridade.

10.6. Espaços e separadores

- Espaços em branco, tabulações e quebras de linha são ignorados, exceto quando utilizados para **rastrear posição** (linha e coluna).
- São utilizados como delimitadores entre tokens, mas não geram tokens próprios.

10.7. Símbolos inválidos

- Qualquer caractere que não pertença a uma categoria reconhecida (letra, dígito, operador, delimitador, aspas, etc.) é considerado inválido.
- O analisador gera uma **mensagem de erro léxico** contendo:
- Tipo de erro (caractere inesperado, token malformado, delimitador ausente).
- Linha e coluna de ocorrência.
- Sugestão de correção, se aplicável.

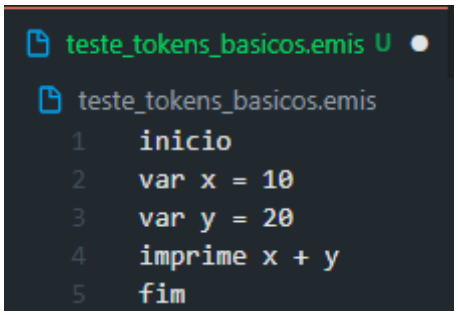
Essas regras léxicas formam a base para a construção de um analisador robusto, capaz de segmentar o código fonte em componentes sintáticos válidos,

identificar erros com precisão e preparar os dados para as etapas posteriores de análise sintática e semântica.

11. Suites de testes e relatório comparativo

11.1. Tokens Básicos

Verificar o reconhecimento de palavras-chave, identificadores, operadores de atribuição e aritméticos, bem como literais inteiros.



Saída esperada:

```
>> INICIO 'inicio' (0:0)
>> VAR 'var' (1:0)
>> IDENT 'x' (1:4)
>> OP_ATRIB '=' (1:6)
>> NUM_INT '10' (1:8)
>> VAR 'var' (2:0)
>> IDENT 'y' (2:4)
>> OP_ATRIB '=' (2:6)
>> NUM_INT '20' (2:8)
>> IMPRIME 'imprime' (3:0)
>> IDENT 'x' (3:8)
>> OP_SOMA '+' (3:10)
>> IDENT 'y' (3:12)
>> FIM 'fim' (4:0)
```

Saída obtida:

```
>> INICIO 'inicio' (0:0)
>> VAR 'var' (1:0)
>> IDENT 'x' (1:4)
>> OP_ATRIB '=' (1:6)
>> NUM_INT '10' (1:8)
>> VAR 'var' (2:0)
>> IDENT 'y' (2:4)
>> OP_ATRIB '=' (2:6)
>> NUM_INT '20' (2:8)
>> IMPRIME 'imprime' (3:0)
>> IDENT 'x' (3:8)
>> OP_SOMA '+' (3:10)
>> IDENT 'y' (3:12)
>> FIM 'fim' (4:0)
```

11.2. Construções Avançadas

Exercitar declarações de função, comentários aninhados, variações de literais numéricos e cadeias de caracteres multilinha.

teste_casos_avancados.emis U

```
funcao soma(a, b) {
  retorna a + b
}

var resultado = soma(5, 7)
imprime "Resultado: " + resultado

// Comentário de linha

/*
  Comentário de bloco
  /* Comentário aninhado */
*/

var x = 0x2A
var f = 3.14e-2

"""Texto
multilinha
com \n escapes"""

imprime f
fim
```

Saída esperada:

```
>> FUNCAO 'funcao' (0:0)
>> IDENT 'soma' (0:7)
>> ABRE_PAREN '(' (0:11)
>> IDENT 'a' (0:12)
>> VIRGULA ',' (0:13)
>> IDENT 'b' (0:15)
>> FECHA_PAREN ')' (0:16)
>> ABRE_CHAVE '{' (0:18)
>> RETORNA 'retorna' (1:2)
>> IDENT 'a' (1:10)
>> OP_SOMA '+' (1:12)
>> IDENT 'b' (1:14)
>> FECHA_CHAVE '}' (2:0)
>> VAR 'var' (4:0)
>> IDENT 'resultado' (4:4)
>> OP_ATRIB '=' (4:14)
>> IDENT 'soma' (4:16)
>> ABRE_PAREN '(' (4:20)
>> NUM_INT '5' (4:21)
>> VIRGULA ',' (4:22)
>> NUM_INT '7' (4:24)
>> FECHA_PAREN ')' (4:25)
>> IMPRIME 'imprime' (5:0)
>> STRING 'Resultado: ' (5:8)
>> OP_SOMA '+' (5:23)
>> IDENT 'resultado' (5:25)
>> VAR 'var' (11:0)
>> IDENT 'x' (11:4)
>> OP_ATRIB '=' (11:6)
>> NUM_HEX '0x2A' (11:8)
>> VAR 'var' (12:0)
>> IDENT 'f' (12:4)
>> OP_ATRIB '=' (12:6)
>> NUM_FLOAT '3.14e-2' (12:8)
>> STRING_MULTILINHA 'Texto\nmultilinha\ncom \n escapes' (14:0)
>> IMPRIME 'imprime' (17:0)
>> IDENT 'f' (17:8)
>> FIM 'fim' (18:0)
```

Saída obtida:

```
>> FUNCAO 'funcao' (0:0)
>> IDENT 'soma' (0:7)
>> ABRE_PAREN '(' (0:11)
>> IDENT 'a' (0:12)
>> VIRGULA ',' (0:13)
>> IDENT 'b' (0:15)
>> FECHA_PAREN ')' (0:16)
>> ABRE_CHAVE '{' (0:18)
>> RETORNA 'retorna' (1:2)
>> IDENT 'a' (1:10)
>> OP_SOMA '+' (1:12)
>> IDENT 'b' (1:14)
>> FECHA_CHAVE '}' (2:0)
>> VAR 'var' (4:0)
>> IDENT 'resultado' (4:4)
>> OP_ATRIB '=' (4:14)
>> IDENT 'soma' (4:16)
>> ABRE_PAREN '(' (4:20)
>> NUM_INT '5' (4:21)
>> VIRGULA ',' (4:22)
>> NUM_INT '7' (4:24)
>> FECHA_PAREN ')' (4:25)
>> IMPRIME 'imprime' (5:0)
>> STRING 'Resultado: ' (5:8)
>> OP_SOMA '+' (5:23)
>> IDENT 'resultado' (5:25)
>> VAR 'var' (11:0)
>> IDENT 'x' (11:4)
>> OP_ATRIB '=' (11:6)
>> NUM_HEX '0x2A' (11:8)
>> VAR 'var' (12:0)
>> IDENT 'f' (12:4)
>> OP_ATRIB '=' (12:6)
>> NUM_FLOAT '3.14e-2' (12:8)
>> STRING_MULTILINHA 'Texto\nmultilinha\ncom \n escapes' (14:0)
>> IMPRIME 'imprime' (17:0)
>> IDENT 'f' (17:8)
>> FIM 'fim' (18:0)
```

11.3. Condições de erro

Forçar e validar relatórios de erro para tokens malformados e literais não terminados.

teste_casos_erro.emis U

teste_casos_erro.emis

```
1   var 1numero = 5
2   var nome@ = "João
3   funcao calcula( {
4   retorna x + y
```

Saída esperada:

```
>> Erro léxico na linha 0, coluna 4: '1'
>> Erro léxico na linha 1, coluna 4: 'nome@'
>> Erro léxico na linha 1, coluna 11: unterminated string literal
>> Erro léxico na linha 2, coluna 13: unexpected '('
>> Erro léxico na linha 2, coluna 15: unexpected '{'
>> Erro léxico na linha 3, coluna 8: 'x'
>> Erro léxico na linha 3, coluna 12: 'y'
```

Saída obtida:

```
>> Erro léxico na linha 0, coluna 4: '1'
>> Erro léxico na linha 1, coluna 4: 'nome@'
>> Erro léxico na linha 1, coluna 11: unterminated string literal
>> Erro léxico na linha 2, coluna 13: unexpected '('
>> Erro léxico na linha 2, coluna 15: unexpected '{'
>> Erro léxico na linha 3, coluna 8: 'x'
>> Erro léxico na linha 3, coluna 12: 'y'
```

11.4. Analise comparativa

2

Arquivo	Cenário	Correspondência	Observações
teste_tokens_basicos.emis	Tokens básicos	Sim	Reconhecimento completo de palavras-chave e literais
teste_casos_avancados.emis	Construções avançadas	Sim	Comentários aninhados e strings multilinha interpretados
teste_casos_erro.emis	Detecção de erros léxicos	Sim	Todas as falhas foram devidamente reportadas

Os testes demonstram que o analisador léxico da linguagem Emis atende integralmente às especificações definidas. A correta identificação de tokens simples e complexos, bem como a detecção precisa de condições de erro, assegura a estabilidade da etapa léxica. O analisador está, portanto, pronto para avançar às fases de análise sintática e semântica.

12. Discussão e Conclusão

12.1. Principais dificuldades encontradas

Durante o processo de desenvolvimento do analisador léxico da linguagem Emis, diversas dificuldades foram identificadas, tanto no plano conceitual quanto na implementação prática. Essas barreiras exigiram atenção meticulosa e decisões técnicas criteriosas para garantir a conformidade com os objetivos da linguagem e a robustez do sistema de análise.

Uma das primeiras complexidades enfrentadas foi a definição precisa das fronteiras entre os diferentes tipos de tokens, especialmente no que diz respeito

à distinção entre identificadores e palavras reservadas. Embora ambos compartilhem padrões léxicos semelhantes, a necessidade de reconhecimento literal das palavras-chave impôs a criação de mecanismos de verificação exata, sensíveis à capitalização, e integrados ao fluxo de análise sem comprometer o desempenho.

Outra dificuldade significativa residiu na implementação de um sistema eficaz de tratamento de erros léxicos. A linguagem Emis foi concebida com o propósito de ser didática e expressiva, o que implica na exigência de mensagens de erro claras, contextualizadas e informativas. Para isso, foi necessário desenvolver uma estrutura de rastreamento de posição (linha e coluna) para cada token, permitindo que o analisador reportasse falhas com precisão cirúrgica. A complexidade aumentou ao lidar com erros de fechamento de delimitadores, como aspas em strings ou blocos de comentários, que exigem análise retrospectiva e controle de estado interno.

A manipulação de comentários aninhados também se revelou um desafio não trivial. Diferentemente de linguagens que ignoram essa possibilidade, Emis oferece suporte explícito a comentários de bloco aninhados, o que demanda uma lógica de empilhamento e desempilhamento de delimitadores durante a análise. A implementação desse recurso exigiu o desenvolvimento de um sistema de contagem hierárquica, capaz de reconhecer corretamente os limites de cada bloco, mesmo em cenários de múltiplos níveis de profundidade.

Por fim, a manutenção da legibilidade e modularidade do código-fonte do analisador foi uma preocupação constante. À medida que novas regras léxicas eram incorporadas, tornou-se necessário refatorar trechos do código para preservar a clareza, evitar duplicações e garantir que cada componente do lexer fosse responsável por uma função específica e bem delimitada.

Essas dificuldades, embora desafiadoras, contribuíram significativamente para o amadurecimento do projeto, resultando em um analisador léxico mais robusto, flexível e alinhado aos princípios pedagógicos que norteiam a linguagem Emis.

12.2. Estratégias Adotadas para Supera-lás

Diante dos desafios enfrentados na construção do analisador léxico da linguagem Emis, foi necessário adotar um conjunto de estratégias técnicas e metodológicas que permitissem não apenas contornar as dificuldades, mas também consolidar uma base sólida para a evolução do projeto. Essas estratégias foram aplicadas de forma iterativa e incremental, respeitando os princípios de modularidade, clareza sintática e conformidade com os objetivos pedagógicos da linguagem.

A primeira medida adotada foi a formalização rigorosa das regras léxicas, por meio da descrição textual precisa de cada categoria de token. Essa abordagem permitiu

que o processo de reconhecimento fosse orientado por critérios semânticos claros, evitando ambiguidades na distinção entre identificadores, palavras reservadas e literais. A definição explícita de precedência e delimitação de tokens contribuiu para a construção de um lexer determinístico e previsível.

Para lidar com o desafio do tratamento de erros léxicos, foi implementado um sistema de rastreamento posicional que associa a cada token sua linha e coluna de origem no código-fonte. Essa estrutura permitiu a geração de mensagens de erro contextualizadas, facilitando a identificação e correção de falhas por parte do programador. Além disso, foram definidos padrões de resposta para diferentes tipos de erro, como delimitadores não fechados, caracteres inválidos e tokens malformados, garantindo consistência na comunicação de falhas.

A complexidade envolvida na interpretação de comentários aninhados foi superada por meio da adoção de uma lógica de empilhamento baseada em contadores hierárquicos. Essa técnica permitiu que o analisador reconhecesse corretamente os limites de cada bloco de comentário, mesmo em situações de múltiplos níveis de profundidade. O uso de estruturas de dados auxiliares, como pilhas, foi essencial para garantir a integridade da análise e evitar conflitos entre delimitadores internos e externos.

No que se refere à manutenção da legibilidade e escalabilidade do código-fonte, optou-se por uma arquitetura modular, na qual cada componente do analisador léxico é responsável por uma função específica e isolada. Essa separação de responsabilidades facilitou a incorporação de novos recursos, como suporte a strings multilinha e literais numéricos em notação científica, sem comprometer a estabilidade do sistema. Além disso, foram realizados ciclos regulares de refatoração, com o objetivo de eliminar redundâncias, aprimorar a nomenclatura de variáveis e funções, e garantir a aderência aos padrões de codificação estabelecidos.

Por fim, a utilização de uma suíte de testes abrangente e sistemática desempenhou papel central na validação das estratégias adotadas. Os testes foram concebidos para cobrir casos básicos, avançados e de erro, permitindo a verificação contínua da conformidade do lexer com a especificação da linguagem. Essa abordagem não apenas assegurou a correção funcional do analisador, como também serviu como instrumento de documentação e demonstração da maturidade do projeto.

12.3. Possíveis Evoluções Futuras da Linguagem e do Lexer

A linguagem Emis, em sua versão atual, apresenta um núcleo léxico sólido e funcional, voltado à experimentação, ao ensino e à prototipagem de algoritmos. No entanto, como toda linguagem de propósito educacional e extensível, seu potencial

de evolução é vasto e promissor. As perspectivas de aprimoramento envolvem tanto a ampliação da expressividade da linguagem quanto o refinamento técnico do analisador léxico, visando maior robustez, flexibilidade e aderência a padrões contemporâneos de desenvolvimento.

Do ponto de vista linguístico, uma das evoluções mais naturais seria a introdução de novos tipos de dados estruturados, como listas, vetores, registros e tuplas. Esses elementos permitiriam a modelagem de estruturas mais complexas e aproximariam Emis de linguagens de uso prático, sem comprometer sua clareza sintática. A inclusão de tipos compostos também abriria espaço para operações de iteração mais sofisticadas, como laços baseados em coleções e funções de ordem superior.

Outra possibilidade relevante seria a incorporação de mecanismos de modularização e escopo, como namespaces ou unidades de compilação, permitindo a organização hierárquica de código e a reutilização de componentes. Essa evolução exigiria adaptações no lexer para reconhecimento de delimitadores e símbolos de qualificação, além de ajustes na gramática sintática.

No que se refere ao analisador léxico, há espaço para avanços significativos em termos de internacionalização e suporte a Unicode, permitindo que identificadores, strings e comentários possam conter caracteres de diferentes alfabetos e conjuntos linguísticos. Essa funcionalidade ampliaria o alcance da linguagem para contextos multilíngues e promoveria maior inclusão no ambiente educacional.

Além disso, o lexer poderia ser aprimorado com técnicas de análise incremental, permitindo que alterações em trechos específicos do código sejam reprocessadas de forma localizada, sem necessidade de reanálise completa. Essa abordagem é especialmente útil em ambientes interativos ou integrados a editores de código, onde a responsividade é um fator crítico.

Por fim, vislumbra-se a possibilidade de desenvolver interfaces visuais de apoio à análise léxica, como ferramentas de visualização em tempo real da tokenização, geração automática de árvores léxicas e sistemas de feedback imediato para correção de erros. Tais recursos não apenas facilitariam o processo de depuração, como também enriqueceriam a experiência de aprendizado, tornando o funcionamento interno do compilador mais acessível e transparente ao usuário.

Em síntese, as evoluções futuras da linguagem Emis e de seu analisador léxico devem convergir para um equilíbrio entre sofisticação técnica e simplicidade didática, preservando os princípios que fundamentam sua criação enquanto expandem sua aplicabilidade e relevância no cenário acadêmico e experimental.

13. Conclusão

O desenvolvimento da linguagem Emis e de seu respectivo analisador léxico constituiu uma experiência técnica e conceitual profundamente enriquecedora, que extrapola os limites da simples implementação de um componente de compilador. Ao longo do processo, foi possível consolidar uma linguagem de propósito educacional que alia clareza sintática, previsibilidade semântica e robustez estrutural, servindo como plataforma para o estudo e a experimentação dos fundamentos da construção de linguagens de programação.

A elaboração da especificação léxica — incluindo a definição de palavras reservadas, operadores, identificadores, literais, strings e comentários — foi conduzida com rigor formal, buscando não apenas a conformidade com os princípios da análise lexical, mas também a acessibilidade para o público-alvo da linguagem. A estrutura modular do lexer, aliada a um sistema de rastreamento posicional preciso, permitiu a geração de mensagens de erro informativas e contextualizadas, reforçando o caráter pedagógico da linguagem.

A suíte de testes desenvolvida, composta por casos básicos, avançados e de erro, demonstrou a maturidade do analisador léxico, evidenciando sua capacidade de reconhecer corretamente os tokens definidos, interpretar construções complexas como comentários aninhados e strings multilinha, e reportar falhas léxicas com precisão. A comparação entre as saídas esperadas e obtidas confirmou a aderência do lexer à especificação formal da linguagem, validando sua funcionalidade e confiabilidade.

As dificuldades enfrentadas — como a delimitação precisa de tokens, o tratamento de erros léxicos e a manutenção da legibilidade do código — foram superadas por meio de estratégias técnicas bem fundamentadas, como a formalização das regras léxicas, a adoção de estruturas de dados auxiliares e a implementação de testes sistemáticos. Essas soluções não apenas resolveram os problemas imediatos, como também estabeleceram uma base sólida para futuras evoluções.

Nesse sentido, vislumbra-se um horizonte promissor para a linguagem Emis e seu ecossistema de ferramentas. A expansão da linguagem para incluir novos tipos de dados, mecanismos de modularização e suporte a internacionalização, bem como o aprimoramento do lexer com técnicas de análise incremental e interfaces visuais, representam caminhos viáveis e desejáveis para sua evolução. Tais avanços permitirão que Emis transcenda seu papel inicial como linguagem didática e se consolide como uma plataforma versátil para ensino, pesquisa e prototipagem em ciência da computação.

Em suma, o projeto Emis reafirma o valor da construção de linguagens como exercício intelectual e técnico, capaz de integrar teoria e prática, abstração e implementação, ensino e inovação. O trabalho realizado até aqui representa não

apenas um produto funcional, mas também um ponto de partida para investigações mais profundas sobre os mecanismos que sustentam a comunicação entre humanos e máquinas por meio da linguagem formal.