Trabalho 1 - Análise Léxica

Arthur Correia, Matheus Queiroz, Pedro Marcos Estrela Outubro, 2025

Introdução

Este projeto tem como objetivo o desenvolvimento de um analisador léxico para uma linguagem de programação imperativa criada pelo grupo, no contexto da disciplina MATA61 – Compiladores, ministrada pelo professor Adriano Maia na Universidade Federal da Bahia (UFBA). Neste documento, apresenta-se a documentação completa do processo de concepção da linguagem, bem como a descrição detalhada de suas funções e respectivas implementações em código.

A proposta do trabalho consiste em projetar uma linguagem própria e original, que suporte elementos fundamentais de linguagens de programação modernas, como declaração e inicialização de variáveis numéricas, vetores, atribuições e expressões aritméticas, estruturas condicionais simples e compostas, laços de repetição e definição de funções com tipagem estática.

A partir da definição formal dessa linguagem, implementamos um analisador léxico utilizando a ferramenta Flex (Fast Lexical Analyzer Generator), cuja função é identificar e classificar os tokens presentes no código-fonte, tais como palavras-chave, identificadores, operadores, números, delimitadores e símbolos especiais. O analisador também registra a posição dos identificadores na tabela de símbolos, de modo a permitir o reconhecimento e manipulação adequada durante as próximas etapas de compilação.

Inspiração e Ideia

O desenvolvimento desta linguagem de programação teve como objetivo criar uma forma de programação acessível e intuitiva para falantes de português, com uma sintaxe que permite ao usuário “dar comandos diretamente à máquina”. A proposta central foi projetar instruções que refletissem ações claras e imperativas, tornando o código mais próximo de uma comunicação natural entre o programador e o computador.

Dessa maneira, comandos como "receba" ou "imprima" substituem funções tradicionais, permitindo que o usuário “ordene” tarefas à máquina de maneira direta e compreensível. Essa abordagem visa tornar a programação mais didática, expressiva e envolvente, sem abrir mão da lógica estruturada necessária para o desenvolvimento.

O resultado é uma linguagem que combina simplicidade, clareza e poder de expressão, oferecendo uma experiência de programação que aproxima o usuário do funcionamento interno do computador de forma intuitiva, mantendo a consistência e a estrutura lógica esperadas em linguagens de programação.

Descrição do Código e Implementação

O código desenvolvido implementa a linguagem descrita anteriormente, realizando o reconhecimento léxico e a geração de tokens correspondentes às instruções do usuário. Ele foi organizado de forma modular, permitindo a leitura clara da lógica de cada componente, desde a inicialização das variáveis até o tratamento de funções e comandos da linguagem.

Inicialização de Variáveis

Na linguagem idealizada, a inicialização de variáveis segue uma lógica similar àquela encontrada em linguagens de programação convencionais. Sob essa perspectiva, os tipos de dados disponíveis são definidos da seguinte maneira:

Inteiro

Equivalente a **int**. É utilizado para representar valores numéricos inteiros.

```
// Entrada:
   inteiro a = 10;
// Saida:
   <inteiro> <id, 0> <=> <inteiro, 10> <;>
```

Listing 1: Inteiro

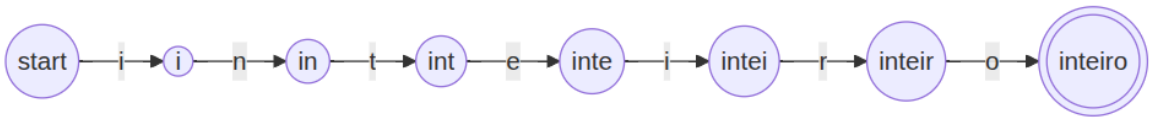

Figure 1: Diagrama de Transição: Inteiro

Quebrado

Equivalente a **float**. É utilizado para representar valores numéricos com casas decimais.

```
// Entrada:
   quebrado b = 20.5;
// Saida:
   <quebrado> <id, 0> <=> <quebrado, 20.5> <;>
```

Listing 2: Decimal

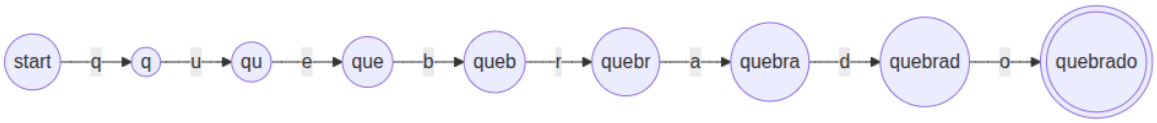

Figure 2: Diagrama de Transição: Quebrado

Texto

Equivalente a **string**. É utilizado para representar sequências de caracteres.

```
// Entrada:
   texto c = "ola mundo"
// Saida:
   <texto> <id, 0> <=> <texto, "ola mundo"> <;>
```

Listing 3: Texto

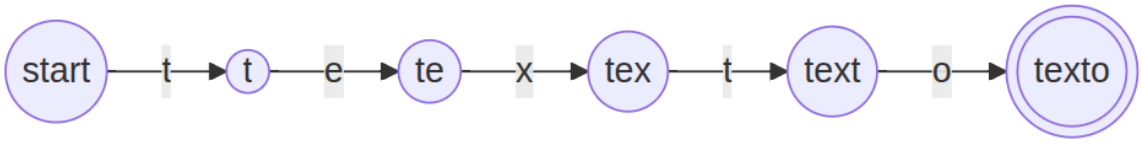

Figure 3: Diagrama de Transição: Texto

Caractere

Equivalente a **char**. É utilizado para representar valores de caractere único.

```
// Entrada:
   caractere d = 'd';
// Saida:
   <caractere> <id, 0> <=> <caractere, 'd'> <;>
```

Listing 4: Caractere

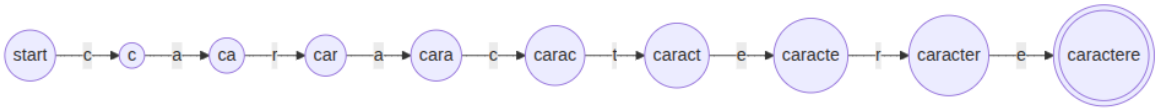

Figure 4: Diagrama de Transição: Caractere

Fato

Equivalente a **bool**. É utilizado para representar valores lógicos. Na linguagem, pode assumir os valores **real** e **fake** que representam, respectivamente, **true** e **false**.

```
// Entrada:
   fato e = real;
   fato f = fake;
// Saida:
   <fato> <id, 0> <=> <fato, real> <;> <fato> <id, 1> <=> <fato, fake> <;>
```

Listing 5: —

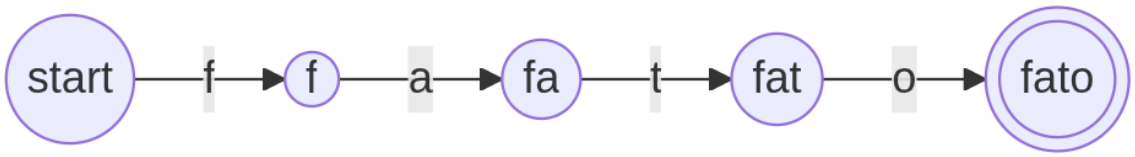

Figure 5: Diagrama de Transição: Fato

Serpente

Equivalente a **vetor**. É utilziado para armazenar sequências de elementos do mesmo tipo;

```
// Entrada:
   serpente lista_fatos (fato) = (fake, real, fake);

// Saida:
   <serpente> <id, 0> <(> <fato> <)> <=> <(> <fato, fake> <fato, real> <fato, fake>
    <)> <;>
```

Listing 6: —

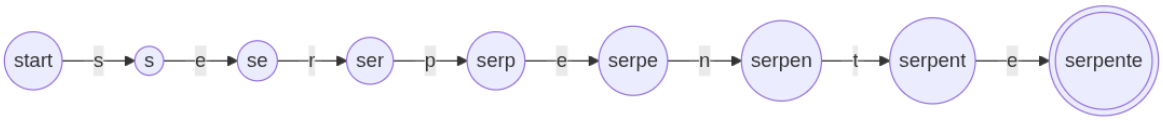

Figure 6: Diagrama de Transição: Serpente

Estruturas de Controle, Repetição e Entrada e Saída de Dados

A linguagem oferece estruturas de controle que permitem direcionar o fluxo de execução do programa, incluindo condicionais e laços de repetição. Além disso, também são disponibilizados comandos de entrada e saída de dados.

Se-Senao

Equivalente a **if-else**. O comando **se** é utilizado para avaliar uma condição e executa o bloco correspondente caso a condição seja verdadeira. O comando **senao** define o bloco alternativo a ser executado quando a condição do **se** não é satisfeita.

```
// Entrada:
   inteiro a = 10;
   inteiro b = 20;
   se (a > b) {
        b = b + 1;
   }
   senao {
        a = a + 1;
   }
// Saida:
   <inteiro> <id, 0> <=> <inteiro, 10> <;> <inteiro> <id, 1> <=> <inteiro, 20> <;> <
    se> <(> <id, 0> <<> <id, 1> <{> <id, 1> <=> <id, 1> <+> <inteiro, 1> <;> <}> <
    senao> <{> <id, 0> <=> <id, 0> <+> <inteiro, 1> <;> <}>
```

Listing 7: Se-Senao

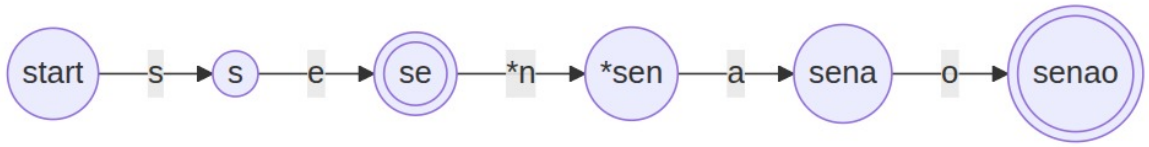

Figure 7: Diagrama de Transição: Se-Senao

Enquanto

Equivalente a **while**. O comando **enquanto** implementa laços de repetição, permitindo que um bloco de código seja executado repetidamente enquanto uma condição lógica for verdadeira.

```
// Entrada:
   inteiro c = 20;
   enquanto (c >= 15) {
        c = c - 1;
   }
// Saida:
   <inteiro> <id, 0> <=> <inteiro, 10> <;> <enquanto> <(> <id, 0> <>=> <inteiro, 15>
    <)> <{> <id, 0> <=> <id, 0> <-> <inteiro, 1> <;> <}>
```

Listing 8: Enquanto

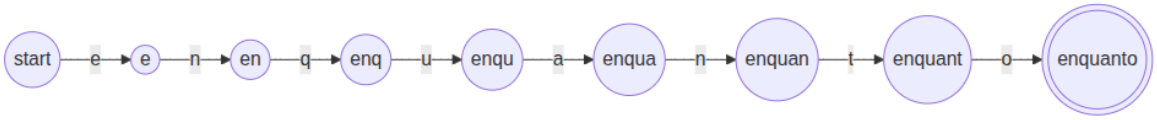

Figure 8: Diagrama de Transição: Enquanto

Ate

Equivalente a **for**. O comando **ate** permite definir um contador e executar um bloco de código repetidas vezes até que determinada condição seja satisfeita.

```
// Entrada:
   inteiro d = 5;
   ate (inteiro i > d) {
        i = i + 1;
   }
// Saida:
   <inteiro> <id, 0> <=> <inteiro, 5> <;> <ate> <(> <inteiro> <id, 1> <>> <id, 0> <)
    > <{> <id, 1> <=> <id, 1> <+> <inteiro, 1> <;> <}>
```

Listing 9: Ate

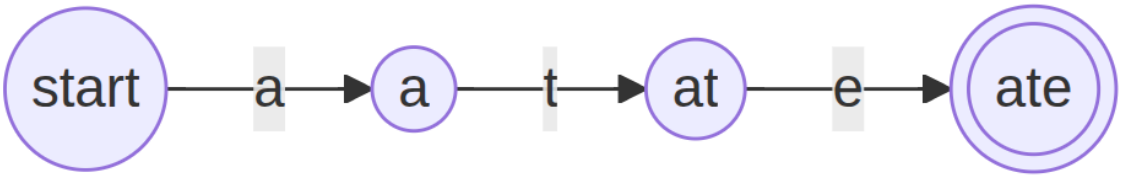

Figure 9: Diagrama de Transição: Ate

Receba

Equivalente a **input**. É utilizado para realizar a entrada de dados, permitindo que o programa capture informações fornecidas pelo usuário durante a execução.

```
// Entrada:
    inteiro e;
    receba(e);
// Saida:
   <inteiro> <id, 0> <;> <receba> <(> <id, 0> <;>
```

Listing 10: Receba

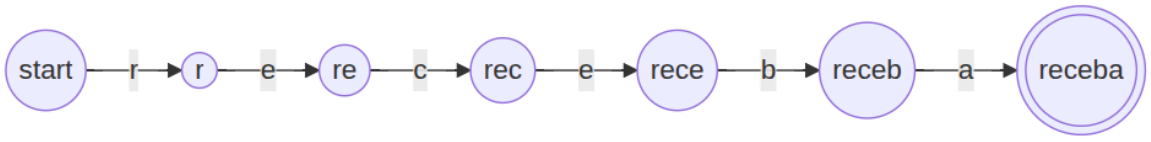

Figure 10: Diagrama de Transição: Receba

Imprima

Equivalente a **print**. É responsável pela saída de dados, permitindo exibir informações na tela durante a execução do programa.

```
// Entrada:
    inteiro f = 10;
    imprima(f);
// Saida:
   <inteiro> <id, 0> <=> <inteiro, 10> <;> <imprima> <(> <id, 0> <;>
```

Listing 11: Imprima

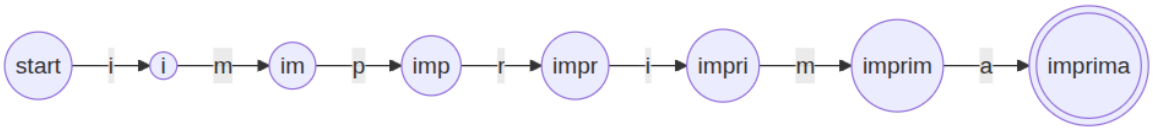

Figure 11: Diagrama de Transição: Imprima

Funções e Controle de Fluxo

Funções

Na linguagem, as funções podem ser dos respectivos tipos: inteiro, quebrado, nada.

Devolva

Equivalente a **return**. É um comando de controle que interrompe a execução de uma função e retorna um valor (ou nenhum) ao ponto onde ela foi chamada.

```
// Entrada:
    inteiro exemplo(inteiro a) {
        devolva a;
    }
// Saida:
   <inteiro> <id, 0> <(> <inteiro> <id, 1> <)> <{> <devolva> <id, 1> <;> <}>
```

Listing 12: Devolva

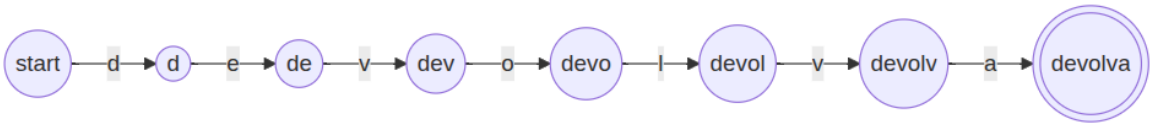

Figure 12: Diagrama de Transição: Devolva

Funções do tipo Inteiro

São as funções que devolvem um valor do tipo inteiro.

```
// Entrada:
    inteiro exemplo(inteiro a) {
        devolva a;
    }
// Saida:
   <inteiro> <id, 0> <(> <inteiro> <id, 1> <)> <{> <devolva> <id, 1> <;> <}>
```

Listing 13: Inteiro

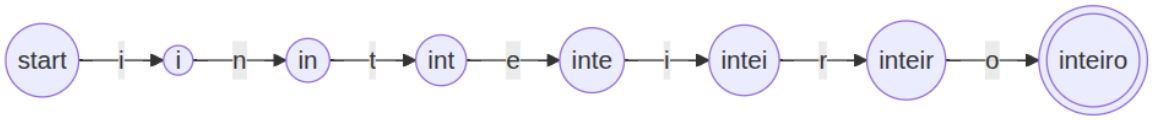

Figure 13: Diagrama de Transição: Inteiro

Funções do tipo Quebrado

São as funções que devolvem um valor do tipo quebrado.

```
// Entrada:
    quebrado exemplo(quebrado b) {
        devolva b;
    }
// Saida:
   <quebrado> <id, 0> <(> <quebrado> <id, 1> <)> <{> <devolva> <id, 1> <;> <}>
```

Listing 14: Quebrado

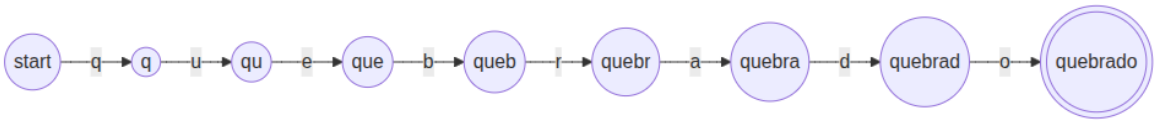

Figure 14: Diagrama de Transição: Quebrado

Funções do tipo Nada

São as funções que não devolvem nada.

```
// Entrada:
    nada exemplo() {
    }
// Saida:
   <nada> <id, 0> <(> <)> <{> <}>
```

Listing 15: Nada

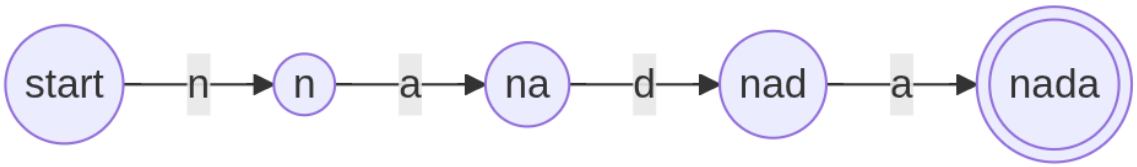

Figure 15: Diagrama de Transição: Nada

Operadores e Separadores

Operadores Aritméticos

Na linguagem, utilizamos os seguintes operadores aritméticos:

- **+**: Representa **adição**;
- **-**: Representa **subtração**;
- **/**: Representa **divisão**;
- **%**: Representa **módulo**;
- ***:** Representa **multiplicação**;
- **=**: Representa **atribuição**.

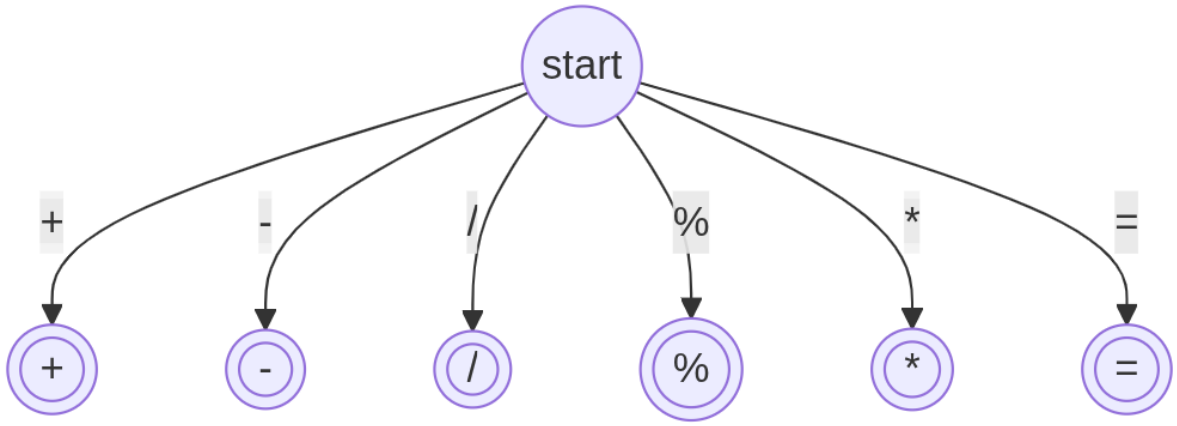

Figure 16: Diagrama de Transição: Operadores Aritméticos

Operadores de Comparação

Na linguagem, utilizamos os seguintes operadores de comparação:

- **>**: Representa **melhor**;
- **>=**: Representa **melhor ou igual**;
- **==**: Representa **igualdade**;
- **<**: Representa **pior**;
- **<=**: Representa **pior ou igual**;
- **!=**: Representa **diferença**.

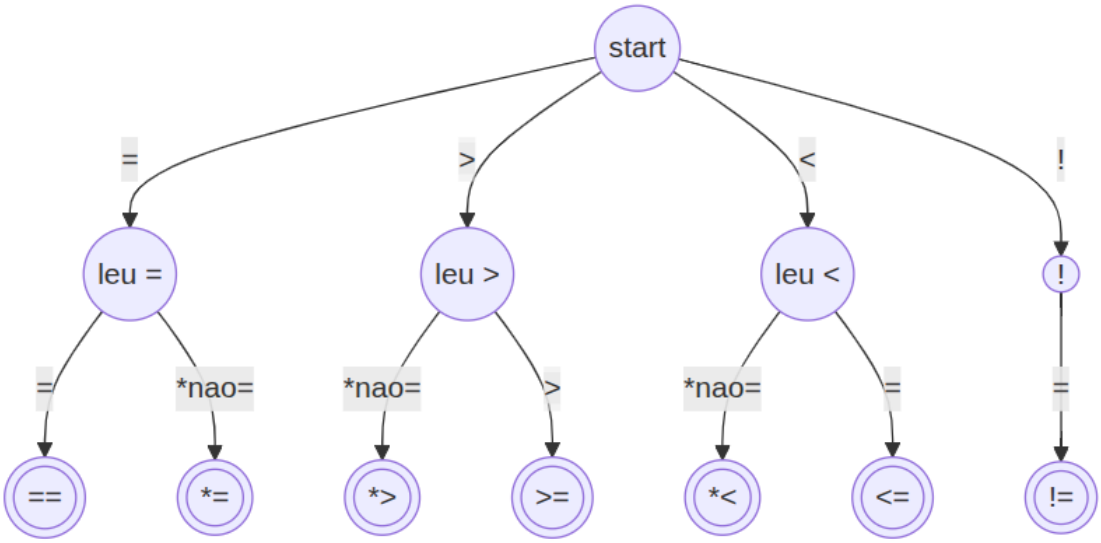

Figure 17: Diagrama de Transição: Operadores de Comparação

Operadores Lógicos

Na linguagem, utilizamos os seguintes operadores lógicos:

- **e**: Equivalente a **&&**;

```
// Exemplo
fato a = real;
fato b = real;
se (a e b) {
    devolva 1;
}

```

Listing 16: E

- **ou** Equivalente a **||**;

```
// Exemplo
fato a = fake;
fato b = real;
se (a ou b) {
    devolva 1;
}
```

Listing 17: Ou

- **nao** Equivalente a **!**.

```
// Exemplo
fato a = fake;
se (nao a) {
    devolva 1;
}

```

Listing 18: Nao

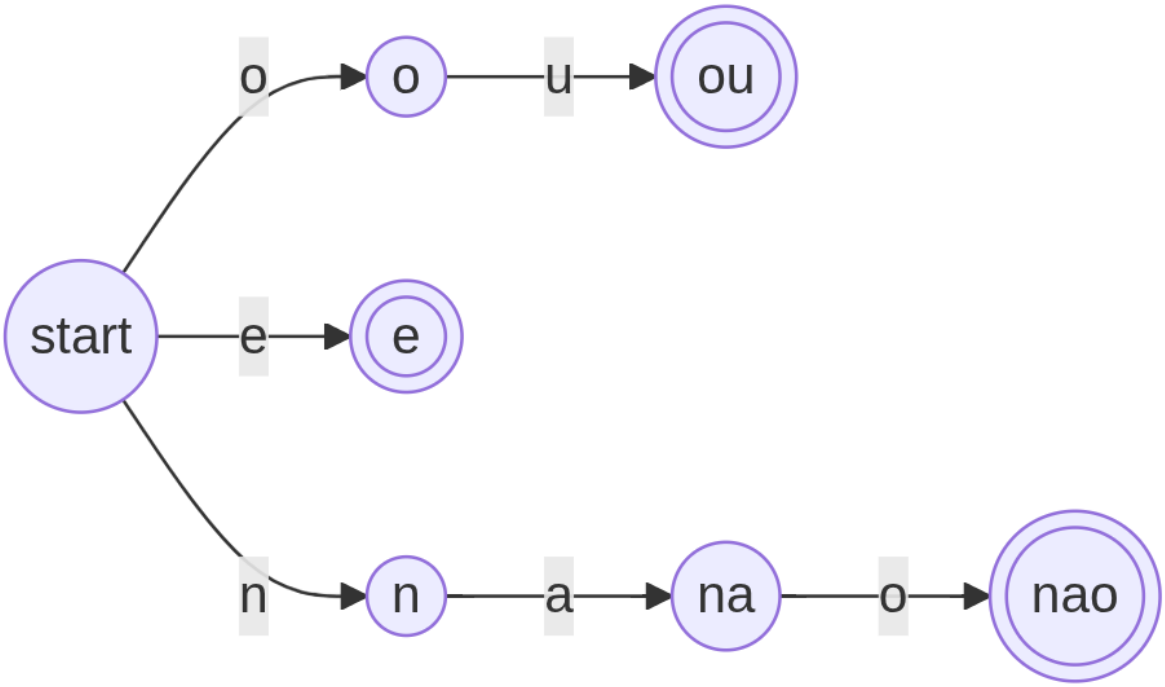

Figure 18: Diagrama de Transição: Operadores Lógicos

Separadores

Na linguagem, são utilizados os seguintes separadores:

- **(**: Representa parenteses direito ou abertura de parenteses;
- **)**: Representa parenteses esquerdo ou fechamento de parenteses;
- **{**: Representa chaves direita ou abertura de chaves;
- **}**: Representa chaves esquerda ou fechamento de chaves;
- **;**: Representa ponto e vírgula;
- **,**: Representa vírgula.

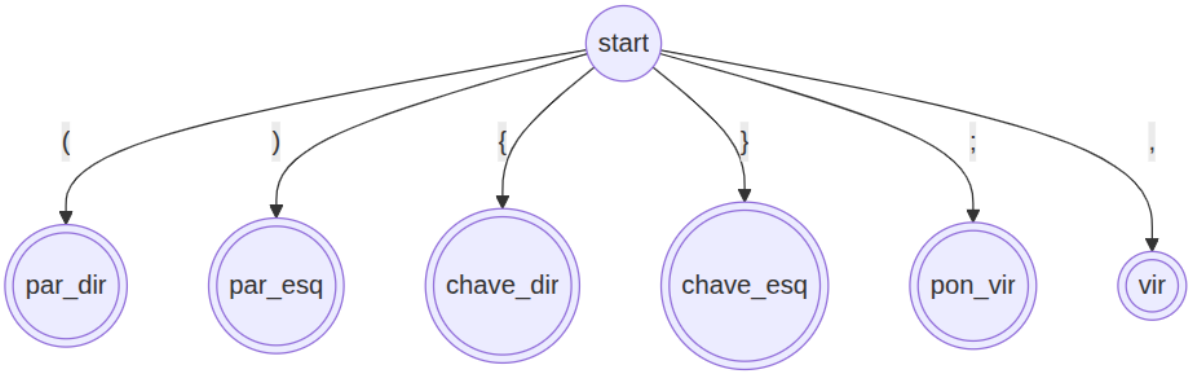

Figure 19: Diagrama de Transição: Separadores

Reconhecimento de Literais

Inteiro

Representa a expressão regular **[0-9]+**.

Quebrado

Representa a expressão regular **[0-9]+.[0-9]+**.

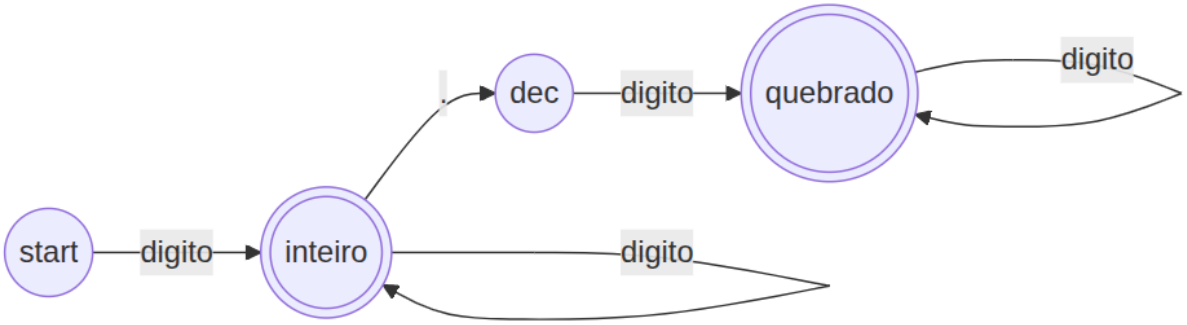

Figure 20: Diagrama de Transição: Inteiros e Quebrados

Caractere

Representa a expressão regular **'[letra]'**.

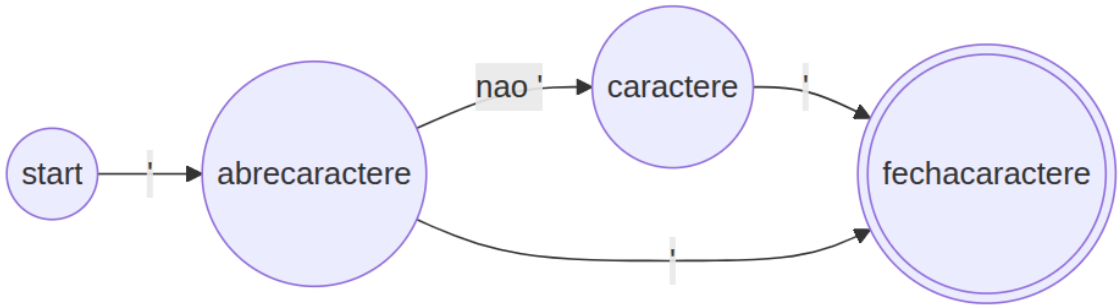

Figure 21: Diagrama de Transição: Caractere

Texto

Representa a expressão regular **"[letra]*"**.

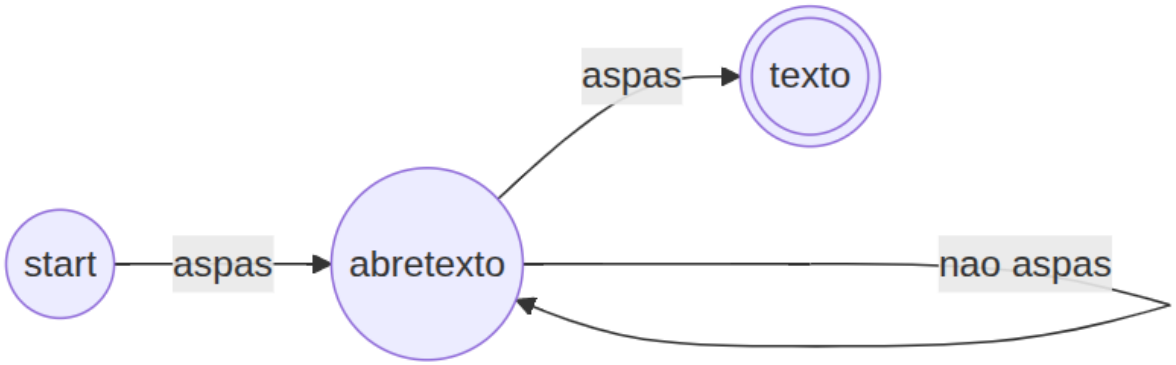

Figure 22: Diagrama de Transição: Texto

Fato

O reconhecimento dos literais do tipo **fato** são da seguinte maneira:

- **Real**:

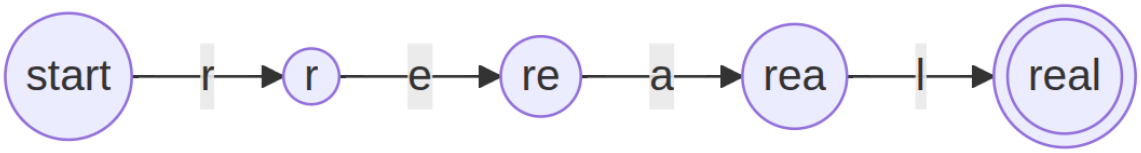

Figure 23: Diagrama de Transição: Real

- **Fake**:

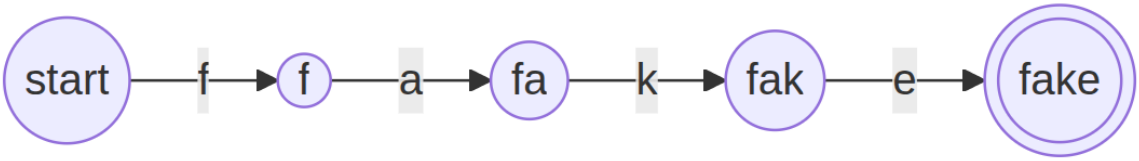

Figure 24: Diagrama de Transição: Fake

Considerações Finais

O desenvolvimento do analisador léxico e a criação de uma nova linguagem a proporcionou uma compreensão aprofundada sobre a estrutura e o funcionamento interno de uma linguagem de programação.

Portanto, a criação das regras para reconhecimento de literais, operadores, variáveis, estruturas de controle e comandos de entrada e saída permitiu consolidar conceitos fundamentais de compiladores.

GitHub

Link para acesso ao repositório do projeto: https://github.com/arthurscorreia/lexical-analysis.