Campus Tancredo Neves

A3 - Teoria da Computação e Compiladores

Salvador - BA

2024

Membros da Equipe

Aluno	RA
Diego Pimenta dos Anjos	1272215402
Fernando de Caires Gonçalves	12724219799
Hélio José da Silva Júnior	12722129211
Lucca Cintra Poggio	1272219553
Reinan Carvalho Amaral	12722126641

Sumário

1. Introdução

1.1 Objetivo

Desenvolver um compilador capaz de traduzir programas escritos em uma linguagem definida pelo grupo para uma linguagem de programação de destino (como Java, C ou Python). O projeto busca aplicar conceitos da Teoria da Computação e de Compiladores, como a definição de gramáticas, análise léxica, análise sintática e semântica, bem como a geração de código equivalente. O compilador deve cumprir os requisitos de verificar tipos, escopos e precedência de operadores, além de suportar estruturas básicas de controle, repetição e entrada/saída, assegurando que o código gerado seja funcional e livre de erros ao ser compilado e executado na linguagem de destino.

1.2 Descrição

O projeto consiste no desenvolvimento de um compilador completo que traduz códigos escritos em uma linguagem criada pelo grupo para a linguagem Java. Para isso, implementamos as etapas fundamentais de construção de um compilador: um analisador léxico, que identifica os tokens a partir do código-fonte; um analisador sintático, responsável por validar a estrutura do programa de acordo com a gramática definida; e um analisador semântico, que verifica tipos, escopos e operações permitidas. Também desenvolvemos uma tabela de símbolos para armazenar informações sobre variáveis e funções declaradas, garantindo o correto rastreamento e validação durante a análise. Por fim, implementamos um gerador de código, que converte o código da linguagem definida pelo grupo em um programa equivalente escrito em Java, assegurando sua compilação e execução sem erros no ambiente de destino.

Para ter acesso ao projeto e compreender como ele funciona, acesse o link abaixo, porém ressaltamos que o README foi feito especificamente para v1 da aplicação, versão que faz uso da biblioteca Antlr4. Em caso de dúvidas em rodar a v2, entre em contato com a equipe.

https://github.com/Equipe-Rocket-Facs/compiler

2. Análise Léxica

2.1 O que é a Análise Léxica?

A análise léxica é a etapa inicial do processo de compilação, onde o código-fonte é examinado e dividido em unidades básicas chamadas *tokens*. Cada *token* representa uma categoria do programa, como palavras-chave (ex.: *if*, *while*), identificadores (ex.: nomes de variáveis), operadores (ex.: +, -), literais (ex.: números, strings) ou símbolos especiais (ex.: *{*, *;*). O principal objetivo dessa etapa é traduzir o fluxo de caracteres do código-fonte em uma forma mais estruturada que será utilizada nas etapas subsequentes, como a análise sintática.

2.2 Como a Análise Léxica Funciona?

O analisador léxico, também chamado de *lexer* ou *scanner*, percorre o código-fonte caracter por caracter, agrupando sequências que correspondem a padrões válidos definidos por expressões regulares ou regras de gramática. Durante esse processo, ele:

- **Identifica tokens válidos**: reconhece elementos como palavras-chave, números ou operadores.
- **Ignora espaços em branco, tabs e comentários**: elimina elementos que não são relevantes para a lógica do programa.
- **Gera erros**: se encontrar caracteres ou sequências que não correspondem a nenhum padrão esperado, ele sinaliza erros léxicos.

2.3 Implementação do Lexer

2.3.1 Definição dos Tokens

Os tokens foram definidos a partir de expressões regulares em um arquivo que o *lexer* fará uso, comparando a entrada com os tokens válidos de nossa gramática. Observe a imagem 1.

2.3.2 Reconhecimento dos Tokens

A seguir temos a devida implementação do nosso *lexer*, ele que fará reconhecimento dos tokens. E caso haja algum que não foi correspondido, ele avisará da existência de erros, contendo motivo, linha e coluna da ocorrência. Observe a imagem 2.

```
public enum TokenType {  235 usages  Diego-Pimenta *

    WS( pattern: "[ \t\r\n]+"),  1 usage
    COMMENT( pattern: "//.*"),  1 usage
    COMMENT_MULTILINE( pattern: "/\\*.*?\\*/"),  1 usage

    PROG( pattern: "\\bprograma\\b"),  1 usage
    END_PROG( pattern: "\\bfimprog\\b"),  5 usages
    INTEIRO( pattern: "\\binteiro\\b"),  12 usages
    DECIMAL( pattern: "\\bdecimal\\b"),  13 usages
    TEXTO( pattern: "\\btexto\\b"),  7 usages
    BOOL( pattern: "\\bbool\\b"),
    LEIA( pattern: "\\bleia\\b"),  10 usages
    ESCREVA( pattern: "\\bescreva\\b"),  9 usages
    IF( pattern: "\\bif\\b"),  9 usages
    ELIF( pattern: "\\belse\\s+if\\b"),  7 usages
    ELSE( pattern: "\\belse\\b"),  7 usages
    WHILE( pattern: "\\bwhile\\b"),  9 usages
    FOR( pattern: "\\bfor\\b"),  9 usages

    OU( pattern: "\\bOU\\b"),
    E( pattern: "\\bE\\b"),
    NAO( pattern: "\\bNAO\\b"),

    VERDADEIRO( pattern: "\\bVERDADEIRO\\b"),  5 usages
    FALSO( pattern: "\\bFALSO\\b"),  5 usages
    ID( pattern: "\\b[a-zA-Z_][a-zA-Z_0-9]*\\b"),
    NUM_DEC( pattern: "\\b[-+]?[0-9]+\\.[0-9]+\\b"),
    NUM_INT( pattern: "\\b[-+]?[0-9]+\\b"),
    STRING( pattern: "\"(\\\\.|[^\"])*\""),  4 usages

    PLUS( pattern: "\\+"),  3 usages
    MINUS( pattern: "-"),  1 usage
    MULT( pattern: "\\*"),  1 usage
    DIV( pattern: "/"),  1 usage
    LEQ( pattern: "<="),  1 usage
    GEQ( pattern: ">="),  1 usage
    LESS( pattern: "<"),  1 usage
    GREATER( pattern: ">"),  1 usage
    EQ( pattern: "=="),  3 usages
    ASSIGN( pattern: "="),  8 usages
    NEQ( pattern: "!="),  3 usages
    LPAREN( pattern: "\\("),  21 usages
    RPAREN( pattern: "\\)"),  21 usages
    LBRACE( pattern: "\\{"),  4 usages
    RBRACE( pattern: "\\}"),  8 usages
    COMMA( pattern: ","),  1 usage
    SEMICOLON( pattern: ";");  7 usages

    public final String pattern;  2 usages

    TokenType(String pattern) { this.pattern = pattern; }
}
```

Imagem 1

```
public class Lexer {  2 usages  Diego-Pimenta *
    public List<Token> tokenize() {  1 usage  Diego-Pimenta *
        int pos = 0;

        while (pos < input.length()) {
            boolean matched = false;

            for (TokenType tokenType : TokenType.values()) {
                Pattern pattern = Pattern.compile( regex: "^" + tokenType.pattern);
                // Aplica o regex para descobrir o token
                Matcher matcher = pattern.matcher(input.substring(pos));

                if (matcher.find()) {
                    String tokenValue = matcher.group();

                    // Ignora espacos em branco e comentarios
                    if (!isIgnored(tokenType)) {
                        // Adiciona variaveis que nao foram inseridas na tabela de simbolos
                        if (isVariable(tokenType) && !isPresent(tokenValue)) {
                            symbolTable.put(tokenValue, new Symbol());
                        }

                        tokens.add(new Token(tokenType, tokenValue, line, column));

                        // Confere se encontramos o fim do programa
                        if (isEOF(tokenType)) {
                            return tokens;
                        }
                    }

                    pos += tokenValue.length();
                    // Atualiza a linha/coluna dos proximos tokens
                    updatePosition(tokenValue);
                    matched = true;
                    break;
                }
            }

            // Checagem para tokens nao validos
            if (!matched) {
                throw new LexicalException("Unrecognized token", line, column);
            }
        }
        return tokens; // Retorna a lista de tokens para a proxima analise
    }
```

Imagem 2

3. Análise Sintática

3.1 O que é a Análise Sintática?

A análise sintática é a segunda etapa do processo de compilação, responsável por verificar se a sequência de *tokens* gerada pela análise léxica segue as regras gramaticais da linguagem de programação. Essa etapa constrói uma representação hierárquica, geralmente uma árvore sintática (ou árvore de derivação), que organiza os elementos do programa de acordo com a estrutura definida pela gramática da linguagem. O objetivo é identificar a organização lógica das instruções e detectar possíveis erros estruturais.

3.2 Como a Análise Sintática Funciona?

O analisador sintático, ou *parser*, recebe como entrada os *tokens* gerados pelo analisador léxico e tenta combiná-los usando uma gramática formal. Ele trabalha validando se a sequência de *tokens* corresponde a padrões gramaticais válidos, utilizando técnicas como análise *top-down* (ex.: *LL*) ou *bottom-up* (ex.: *LR*). Durante o processo, ele:

- **Reconhece estruturas válidas**: como declarações de variáveis, comandos condicionais (*if-else*), laços de repetição (*while*, *for*) e expressões matemáticas.
- **Constrói a árvore sintática**: organiza os *tokens* em uma hierarquia que reflete a estrutura lógica do programa.
- **Sinaliza erros**: detecta problemas estruturais, como uma falta de parênteses ou uso incorreto de palavras-chave.

3.3 Implementação do Parser

Em razão do tamanho da implementação da lógica do *parser*, traremos apenas exemplos de como funciona a verificação dos tokens na declaração de variáveis (Imagem 1) e em expressões matemáticas (Imagem 2). Vale ressaltar que nosso sintático permite das mais diversas expressões booleanas, relacionais e matemáticas, contendo as devidas regras na sua formação, como impedir que haja uma expressão relacional entre um tipo booleano e um número.

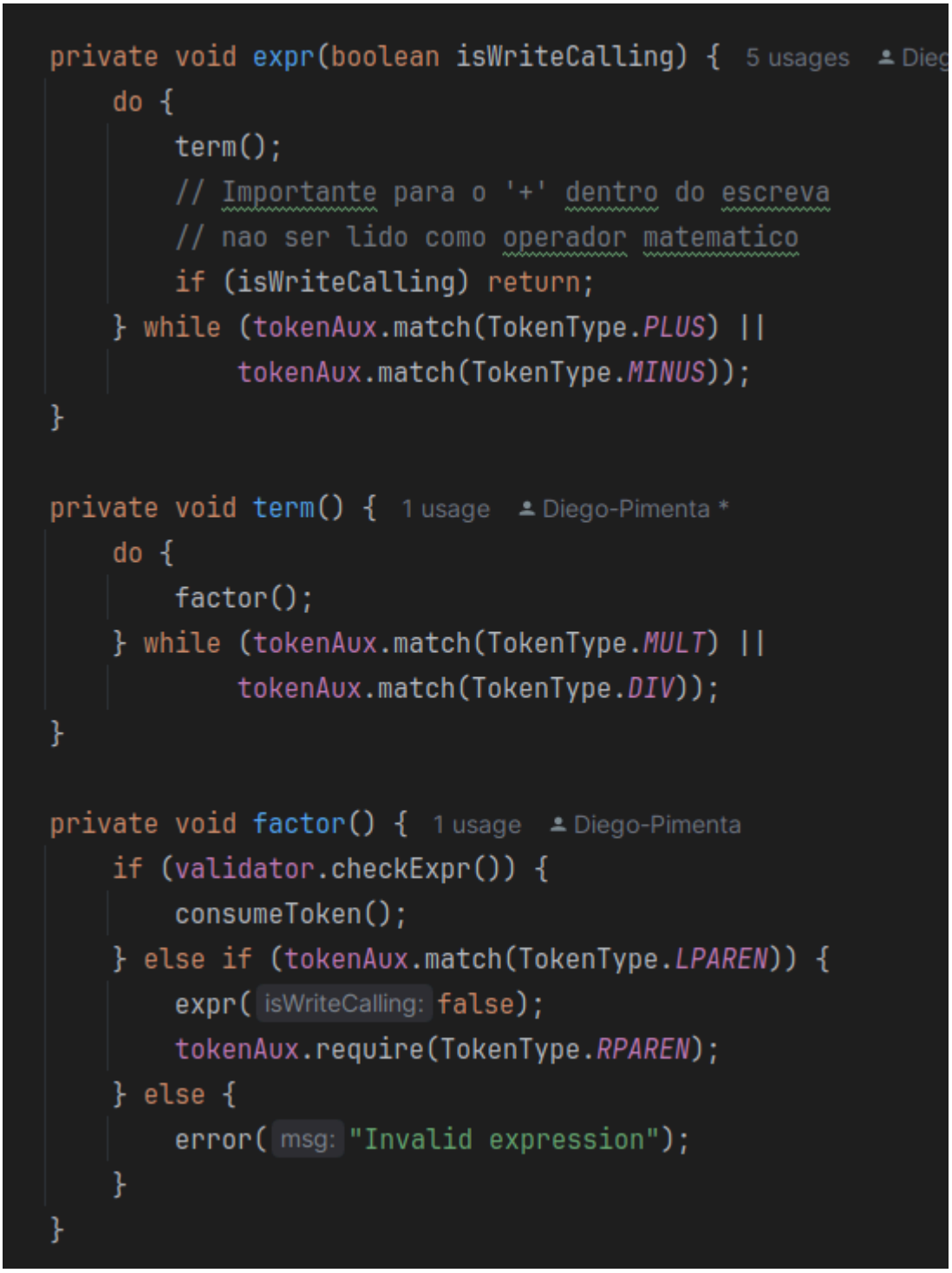
```java
private void declarations() {  1 usage  Diego-Pimenta
    while (validator.checkType()) {
        declaration();
    }
}

private void declaration() {  1 usage  Diego-Pimenta
    TokenType type = type();
    declarationList(type);
}

private void declarationList(TokenType type) {  1 usage  Dieg
    // Altera a tabela de simbolos para guardar o
    // tipo e quantidade de ocorrencia das variaveis
    do {
        String idName = tokenAux.peek().getValue();
        tokenAux.require(TokenType.ID);
        Symbol symbol = symbolTable.get(idName);
        symbol.setType(type);
        symbol.incrementCount();
        symbolTable.put(idName, symbol);
    } while (tokenAux.match(TokenType.COMMA));
}

private TokenType type() {  1 usage  Diego-Pimenta
    TokenType type = tokenAux.peek().getType();
    consumeToken();
    return type;
}
```

Imagem 1



```
private void expr(boolean isWriteCalling) {  5 usages  Dieg
    do {
        term();
        // Importante para o '+' dentro do escreva
        // nao ser lido como operador matematico
        if (isWriteCalling) return;
    } while (tokenAux.match(TokenType.PLUS) ||
             tokenAux.match(TokenType.MINUS));
}

private void term() {  1 usage  Diego-Pimenta *
    do {
        factor();
    } while (tokenAux.match(TokenType.MULT) ||
             tokenAux.match(TokenType.DIV));
}

private void factor() {  1 usage  Diego-Pimenta
    if (validator.checkExpr()) {
        consumeToken();
    } else if (tokenAux.match(TokenType.LPAREN)) {
        expr( isWriteCalling: false);
        tokenAux.require(TokenType.RPAREN);
    } else {
        error( msg: "Invalid expression");
    }
}
```

Imagem 2

4. Equivalência entre Linguagens

1. Caso de teste nº 1 - Tabuada de multiplicação do número 9

Código na linguagem criada:

```
programa

inteiro x, y

y = 10

escreva("Tabuada do 9")

for (x = 0; x <= y; x = x + 1) {
   escreva("Resultado de 9 x " + x + " = " + x * 9)
}

fimprog
```

Código na linguagem Java:

```
public class Program {
   public static void main(String[] args) {
       int x, y;
       y = 10;
       System.out.println("Tabuada do 9");
       for (x = 0; x <= y; x = x + 1) {
           System.out.println("Resultado de 9 x " + x + " = " + x
* 9);
       }
   }
}
```

2. Caso de teste nº 7 - Calculadora de média com notas decimais

Código na linguagem criada:

```
programa

decimal nota1, nota2, nota3, media
texto situacao

escreva("Digite a primeira nota")
leia(nota1)
escreva("Digite a segunda nota")
leia(nota2)
escreva("Digite a terceira nota")
leia(nota3)

media = (nota1 + nota2 + nota3) / 3

if (media >= 7.0) {
   situacao = "Aprovado"
} else if (media >= 5.0) {
   situacao = "Recuperação"
} else {
   situacao = "Reprovado"
}

escreva("Média: " + media)
escreva("Situação: " + situacao)

fimprog
```

Código na linguagem Java:

```java
import java.util.Scanner;

public class Program {
   public static void main(String[] args) {
       String situacao;
       double nota1, nota2, nota3, media;
       Scanner scanner = new Scanner(System.in);
       System.out.println("Digite a primeira nota");
       nota1 = scanner.nextDouble();
       System.out.println("Digite a segunda nota");
       nota2 = scanner.nextDouble();
       System.out.println("Digite a terceira nota");
       nota3 = scanner.nextDouble();
       media = (nota1 + nota2 + nota3) / 3;
       if (media >= 7.0) {
           situacao = "Aprovado";
       } else if (media >= 5.0) {
           situacao = "Recuperação";
       } else {
           situacao = "Reprovado";
       }
       System.out.println("Média: " + media);
       System.out.println("Situação: " + situacao);
   }
}
```