RELATÓRIO TÉCNICO
COMPILADORES

AARON MARTINS LEÃO FERREIRA
GABRIEL ROGÉRIO APARECIDO PAES SILVA
JOÃO LUCAS PEREIRA DE ALMEIDA
VINÍCIUS DE OLIVEIRA FABIANO

LAVRAS
2025

COMPILADORES

RELATÓRIO TÉCNICO

Relatório técnico apresentado à disciplina de Compiladores, do professor Mauricio Ronny, para obtenção de nota relativa ao período vigente.

LAVRAS
2025

RELATÓRIO

1. INTRODUÇÃO

Este relatório apresenta uma análise detalhada do desenvolvimento de uma linguagem de programação projetada pelo grupo, abordando aspectos fundamentais de seu design e implementação. Serão explorados os componentes essenciais da linguagem, incluindo definições de regras sintáticas, estruturas semânticas e classificação de lexemas. Por meio de exemplos práticos, o relatório demonstrará a aplicação da linguagem em algoritmos específicos criados durante o projeto. Além disso, serão discutidos os processos de construção dos analisadores léxico, sintático e semântico, destacando suas etapas de implementação e os resultados obtidos em diferentes cenários de teste. O objetivo é proporcionar uma visão clara e abrangente das funcionalidades e da arquitetura da linguagem desenvolvida.

2. VISÃO GERAL DA LINGUAGEM

A linguagem desenvolvida pelo grupo, denominada Tupy, é uma linguagem de programação que combina a robustez da sintaxe de C++ com características adaptadas ao português. Ela visa facilitar o aprendizado e a utilização da programação para falantes de português, incorporando palavras-chave e comandos que refletem a linguagem natural. Suas principais características incluem uma estrutura orientada a objetos, suporte a tipos de dados complexos e uma sintaxe intuitiva que promove a legibilidade do código.

3. DEFINIÇÃO LÉXICA DA LINGUAGEM

Para a etapa de definição léxica da linguagem, optou-se por realizar uma tabela que contém os tokens, suas descrições e os lexemas correspondentes. Essa abordagem facilita a visualização e compreensão das diferentes classes de lexemas aceitas pela linguagem, bem como seus padrões de identificação. A tabela servirá como um guia claro para o analisador léxico, permitindo uma identificação eficiente dos elementos da linguagem.

Classe	Padrão	Lexema
ID	Nome da variável.	Caracteres sem espaço.
NUMINT	Números inteiros.	Números sem casas decimais.
NUMREAL	Números reais.	Números com casas decimais.
TIPO	Tipo forte da linguagem.	lin, int, real, boo
ESCRITA	Entrada de escrita pelo usuário.	esc
LEITURA	Impressão de variáveis.	lei
RETORNO	Retorno de função.	ret
NRETORNO	Sem retorno de função.	nret
FUNCAO	Declaração de função.	func
SE	Estrutura condicional “SE”.	se
ENTAO	Estrutura condicional “ENTAO”.	entao
SENAO	Estrutura condicional “SENÃO”.	senao
ENQUANTO	Estrutura de repetição “ENQUANTO”.	enquanto
OPLOG	Operadores lógicos.	e, ou
OPMAT	Operações matemáticas.	+, -, *, /, %
OPREL	Operações relacionais.	==, !=, >, <, >=, <=
OPINCDEC	Operações de incremento ou decremento.	++, --
OPATRIB	Operação de atribuição.	=
FLINHA	Final de linha.	;
AP	Abre parênteses.	(
FP	Fecha parênteses.	)
AC	Abre chaves.	{
FC	Fecha chaves.	}
WS	Caracteres ignorados.	, \r, \t, \n
ErrorChar	Caracteres com erro.	Caractere fora da linguagem.

4. EXEMPLOS DE USO DA LINGUAGEM

Para ilustrar o uso da linguagem, foram selecionados exemplos práticos com a implementação dos algoritmos de Fatorial e da Soma dos N Termos da Sequência de Fibonacci. Esses exemplos demonstram de maneira clara e concisa como a linguagem pode ser aplicada em situações reais de programação.

- **Fatorial**

```
func fatorial(n){
	se (n == 0) entao {
		ret 1;
	}
	senao {
		ret n * fatorial(n - 1);
	}
}
```

- **Soma dos N Termos da Sequência de Fibonacci**

```
func fibonacci(n){
	se (n <= 0) entao {
	   ret 0;
	}
	se (n == 1) entao {
	   ret 1;
	}
	senao {
	  ret fibonacci(n - 1) + fibonacci(n - 2);
	}
}

func somaFibonacci(n) {
	int soma = 0;
	int i = 0;
	enquanto (i < n) {
		soma = soma + fibonacci(i);
		i++;
	}
	ret soma;
}
```

5. IMPLEMENTAÇÃO DO ANALISADOR LÉXICO

No desenvolvimento do analisador léxico para a nova linguagem, foi optado pelo uso do ANTLR (Another Tool for Language Recognition), uma escolha motivada pela sua facilidade de uso e eficiência na geração de analisadores. Essa decisão foi fortemente recomendada pelo professor, graças as vantagens do ANTLR em relação a outras ferramentas, especialmente no que diz respeito à clareza na definição de gramáticas e à robustez na análise lexical. A imagem a seguir ilustra a estrutura e as

regras definidas para a linguagem, facilitando a compreensão dos componentes léxicos. O arquivo utilizado para essa definição é um arquivo com extensão .g4, específico para a sintaxe do ANTLR.

```
grammar Tupy;

TIPO: 'lin' | 'int' | 'real' | 'boo';
ESCRITA: 'esc';
LEITURA: 'lei';
RETORNO: 'ret';
NRETORNO: 'nret';
FUNCAO: 'func';
SE: 'se';
ENTAO: 'entao';
SENAO: 'senao';
ENQUANTO: 'enquanto';
OPLOG: 'e' | 'ou';
OPMAT: '+' | '-' | '*' | '/' | '%';
OPREL: '==' | '!=' | '>' | '<' | '>=' | '<=';
OPINCDEC: '++' | '--';
OPATRIB: '=';
FLINHA: ';';
AP: '(';
FP: ')';
AC: '{';
FC: '}';
ID: LETRA(DIGITO|LETRA)*;
NUMINT: DIGITO+;
NUMREAL: DIGITO+ ',' +DIGITO+;
fragment LETRA: [a-zA-Z];
fragment DIGITO: [0-9];
WS: [ \r\t\n]* ->skip;
ErrorChar: . ;
```

O dicionário da linguagem foi implementado e testado em um ambiente Java, o que possibilitou a geração de tokens e a identificação de seus respectivos lexemas de forma simples. Durante os testes, o analisador léxico demonstrou sua capacidade de processar entradas variadas, imprimindo com precisão os tokens reconhecidos e suas representações lexicais. Essa abordagem não apenas garantiu a correta

funcionalidade do dicionário, mas também facilitou a depuração e a validação da gramática da linguagem. O seguinte código .java foi utilizado para a realização dos testes:

```java
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.CharStreams;
import org.antlr.v4.runtime.Token;
import org.antlr.v4.runtime.TokenStream;

import java.io.IOException;

public class Main {

    public static void main(String[] args) {
        String filename = "C:\\Users\\Downloads\\Analisador-Lexico-main\\codigos\\Fatorial (Com erros).txt";

        try {
            CharStream input = CharStreams.fromFileName(filename);
            TupyLexer lexer = new TupyLexer(input);
            Token token;

            while ((token = lexer.nextToken()).getType() != Token.EOF) {
                String tokenType = lexer.getVocabulary().getSymbolicName(token.getType());

                if (tokenType.equalsIgnoreCase("ErrorChar") || tokenType.isEmpty()) {
                    System.out.println("Erro: Token inválido na linha "
                    + token.getLine() + " com lexema: \""
                    + token.getText() + "\"");

                } else {
                    System.out.println("Token: <Classe: " + tokenType + " ,Lexema: " + token.getText() + ">");
                }
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

6. CASOS DE TESTE - LÉXICO

Na etapa de testes, o analisador léxico foi avaliado com os códigos em formato .txt referentes ao cálculo do Fatorial e à Soma dos N Termos da Sequência de Fibonacci, os quais estão apresentados no documento. Esses testes foram fundamentais para validar a eficácia do dicionário e do analisador léxico desenvolvidos. As imagens a seguir mostram os resultados, com sucesso, obtidos ao executar o código em Java, trabalhando como o analisador léxico:

- **Fatorial**

```
Token: <Classe: FUNCAO ,Lexema: func>
Token: <Classe: ID ,Lexema: fatorial>
Token: <Classe: AP ,Lexema: (>
Token: <Classe: ID ,Lexema: n>
Token: <Classe: FP ,Lexema: )>
Token: <Classe: AC ,Lexema: {>
Token: <Classe: SE ,Lexema: se>
Token: <Classe: AP ,Lexema: (>
Token: <Classe: ID ,Lexema: n>
Token: <Classe: OPREL ,Lexema: ==>
Token: <Classe: NUMINT ,Lexema: 0>
Token: <Classe: FP ,Lexema: )>
Token: <Classe: ENTAO ,Lexema: entao>
Token: <Classe: AC ,Lexema: {>
Token: <Classe: RETORNO ,Lexema: ret>
Token: <Classe: NUMINT ,Lexema: 1>
Token: <Classe: FLINHA ,Lexema: ;>
Token: <Classe: FC ,Lexema: }>
Token: <Classe: SENAO ,Lexema: senao>
Token: <Classe: AC ,Lexema: {>
Token: <Classe: RETORNO ,Lexema: ret>
Token: <Classe: ID ,Lexema: n>
Token: <Classe: OPMAT ,Lexema: *>
Token: <Classe: ID ,Lexema: fatorial>
Token: <Classe: AP ,Lexema: (>
Token: <Classe: ID ,Lexema: n>
Token: <Classe: OPMAT ,Lexema: ->
Token: <Classe: NUMINT ,Lexema: 1>
Token: <Classe: FP ,Lexema: )>
Token: <Classe: FLINHA ,Lexema: ;>
Token: <Classe: FC ,Lexema: }>
Token: <Classe: FC ,Lexema: }>
Token: <Classe: EOF ,Lexema: <EOF>>
```

- **Soma dos N Termos da Sequência de Fibonacci**

```
Token: <Classe: FUNCAO ,Lexema: func>
Token: <Classe: ID ,Lexema: fibonacci>
Token: <Classe: AP ,Lexema: (>
Token: <Classe: ID ,Lexema: n>
Token: <Classe: FP ,Lexema: )>
Token: <Classe: AC ,Lexema: {>
Token: <Classe: SE ,Lexema: se>
Token: <Classe: AP ,Lexema: (>
Token: <Classe: ID ,Lexema: n>
Token: <Classe: OPREL ,Lexema: <=>
Token: <Classe: NUMINT ,Lexema: 0>
Token: <Classe: FP ,Lexema: )>
Token: <Classe: ENTAO ,Lexema: entao>
Token: <Classe: AC ,Lexema: {>
Token: <Classe: RETORNO ,Lexema: ret>
Token: <Classe: NUMINT ,Lexema: 0>
Token: <Classe: FLINHA ,Lexema: ;>
Token: <Classe: FC ,Lexema: }>
Token: <Classe: SE ,Lexema: se>
Token: <Classe: AP ,Lexema: (>
Token: <Classe: ID ,Lexema: n>
Token: <Classe: OPREL ,Lexema: ==>
Token: <Classe: NUMINT ,Lexema: 1>
Token: <Classe: FP ,Lexema: )>
Token: <Classe: ENTAO ,Lexema: entao>
Token: <Classe: AC ,Lexema: {>
Token: <Classe: RETORNO ,Lexema: ret>
Token: <Classe: NUMINT ,Lexema: 1>
Token: <Classe: FLINHA ,Lexema: ;>
Token: <Classe: FC ,Lexema: }>
Token: <Classe: SENAO ,Lexema: senao>
Token: <Classe: AC ,Lexema: {>
Token: <Classe: RETORNO ,Lexema: ret>
Token: <Classe: ID ,Lexema: fibonacci>
Token: <Classe: AP ,Lexema: (>
Token: <Classe: ID ,Lexema: n>
Token: <Classe: OPMAT ,Lexema: ->
Token: <Classe: NUMINT ,Lexema: 1>
```

```
Token: <Classe: FP ,Lexema: )>
Token: <Classe: OPMAT ,Lexema: +>
Token: <Classe: ID ,Lexema: fibonacci>
Token: <Classe: AP ,Lexema: (>
Token: <Classe: ID ,Lexema: n>
Token: <Classe: OPMAT ,Lexema: ->
Token: <Classe: NUMINT ,Lexema: 2>
Token: <Classe: FP ,Lexema: )>
Token: <Classe: FLINHA ,Lexema: ;>
Token: <Classe: FC ,Lexema: }>
Token: <Classe: FC ,Lexema: }>
Token: <Classe: FUNCAO ,Lexema: func>
Token: <Classe: ID ,Lexema: somaFibonacci>
Token: <Classe: AP ,Lexema: (>
Token: <Classe: ID ,Lexema: n>
Token: <Classe: FP ,Lexema: )>
Token: <Classe: AC ,Lexema: {>
Token: <Classe: TIPO ,Lexema: int>
Token: <Classe: ID ,Lexema: soma>
Token: <Classe: OPATRIB ,Lexema: =>
Token: <Classe: NUMINT ,Lexema: 0>
Token: <Classe: FLINHA ,Lexema: ;>
Token: <Classe: TIPO ,Lexema: int>
Token: <Classe: ID ,Lexema: i>
Token: <Classe: OPATRIB ,Lexema: =>
Token: <Classe: NUMINT ,Lexema: 0>
Token: <Classe: FLINHA ,Lexema: ;>
Token: <Classe: ENQUANTO ,Lexema: enquanto>
Token: <Classe: AP ,Lexema: (>
Token: <Classe: ID ,Lexema: i>
Token: <Classe: OPREL ,Lexema: <>
Token: <Classe: ID ,Lexema: n>
Token: <Classe: FP ,Lexema: )>
Token: <Classe: AC ,Lexema: {>
Token: <Classe: ID ,Lexema: soma>
Token: <Classe: OPATRIB ,Lexema: =>
Token: <Classe: ID ,Lexema: soma>
Token: <Classe: OPMAT ,Lexema: +>
Token: <Classe: ID ,Lexema: fibonacci>
```

```
Token: <Classe: AP ,Lexema: (>
Token: <Classe: ID ,Lexema: i>
Token: <Classe: FP ,Lexema: )>
Token: <Classe: FLINHA ,Lexema: ;>
Token: <Classe: ID ,Lexema: i>
Token: <Classe: OPINCDEC ,Lexema: ++>
Token: <Classe: FLINHA ,Lexema: ;>
Token: <Classe: FC ,Lexema: }>
Token: <Classe: RETORNO ,Lexema: ret>
Token: <Classe: ID ,Lexema: soma>
Token: <Classe: FLINHA ,Lexema: ;>
Token: <Classe: FC ,Lexema: }>
Token: <Classe: EOF ,Lexema: <EOF>>
```

Do mesmo modo, foi realizado um caso de erro para testar a capacidade do analisador de reconhecer um caractere não presente no dicionário da linguagem. Para isso, foi inserido o caractere conhecido como trema (¨) no código do Fatorial, conforme mostrado na imagem a seguir. Esse teste visou avaliar como o analisador lida com entradas inválidas e sua habilidade de identificar e reportar erros lexicais de forma eficaz.

```
1  func fatorial(n){
2      se (n == 0) entao {
3          ret 1;
4      }
5      senao {
6          ret n * fatorial(n - 1);
7      }
8  }¨¨
```

Assim, o trema (¨) não foi reconhecido pela linguagem e resultou na apresentação do token ErrorChar, indicando que esse caractere não está contido no dicionário da linguagem.

```
Token: <Classe: FLINHA ,Lexema: ;>
Token: <Classe: FC ,Lexema: }>
Token: <Classe: ErrorChar ,Lexema: ¨>
Token: <Classe: ErrorChar ,Lexema: ¨>
Token: <Classe: FC ,Lexema: }>
```

Ao realizar a análise da função de Fibonacci, com lexemas incompatíveis, neste caso o “@”, o mesmo token apereceu, indicando que qualquer caractere inválido e fora do dicionário será tratado como um erro pelo analisador. Com isso o analisador léxico é capaz de encaminhar o código para um futuro analisador sintático, evidenciando que todos os lexemas apresentados estão de acordo com o dicionário da linguagem, caso não apresente tokens de erro e estejam de acordo com as normas impostas pelo analisador.

```
func fibonacci(n){
    se (n <= 0) entao {
        ret 0;
    }
    se (n == 1) entao {
        ret 1;
    }
    senao {
        ret fibonacci(n - 1) + fibonacci(n - 2);
    }
}

func som@Fibon@cci(n) {
    int som@ = 0;
    int i = 0;
    enqu@nto (i < n) {
        som@ = som@ + fibon@cci(i);
        i++;
    }
    ret som@;
}
```

```
Token: <Classe: FP ,Lexema: )>
Token: <Classe: AC ,Lexema: {>
Token: <Classe: ID ,Lexema: som>
Token: <Classe: ErrorChar ,Lexema: @>
Token: <Classe: OPATRIB ,Lexema: =>
Token: <Classe: ID ,Lexema: som>
Token: <Classe: ErrorChar ,Lexema: @>
Token: <Classe: OPMAT ,Lexema: +>
Token: <Classe: ID ,Lexema: fibon>
Token: <Classe: ErrorChar ,Lexema: @>
Token: <Classe: ID ,Lexema: cci>
Token: <Classe: AP ,Lexema: (>
Token: <Classe: ID ,Lexema: i>
Token: <Classe: FP ,Lexema: )>
Token: <Classe: FLINHA ,Lexema: ;>
Token: <Classe: ID ,Lexema: i>
Token: <Classe: OPINCDEC ,Lexema: ++>
Token: <Classe: FLINHA ,Lexema: ;>
Token: <Classe: FC ,Lexema: }>
Token: <Classe: RETORNO ,Lexema: ret>
Token: <Classe: ID ,Lexema: som>
Token: <Classe: ErrorChar ,Lexema: @>
Token: <Classe: FLINHA ,Lexema: ;>
Token: <Classe: FC ,Lexema: }>
Token: <Classe: EOF ,Lexema: <EOF>>
```

7. DEFINIÇÃO SINTÁTICA DA LINGUAGEM

Foi adicionada ao arquivo tupy.g4 uma gramática livre de contexto para definir a estrutura sintática da linguagem Tupy. Essa gramática abrange regras para declarações, atribuições, operações, chamadas de função, comandos de escrita e leitura, controle de fluxo e definições de funções. Com isso, a base foi criada para o desenvolvimento do analisador sintático, complementando o analisador léxico e permitindo a validação estrutural dos códigos.

```
programa: instrucao+ EOF;

instrucao: declaracao                  // Declarações de variáveis
         | declaracaoComAtribuicao     // Declarações de variáveis e
atribuição de valores
         | atribuicao                  // Atribuição de valores
         | incrementoOuDecremento      // Operações de incremento ou
decremento
         | chamadaFuncao               // Chamada de função
         | escrita                     // Comando de escrita
         | leitura                     // Comando de leitura
         | retorno                     // Comando de retorno
         | funcao                      // Definição de função
         | condicional                 // Estrutura condicional
         | repeticao                   // Estrutura de repetição
         ;

declaracao: TIPO ID FLINHA;

declaracaoComAtribuicao: TIPO ID OPATRIB expressao FLINHA;

atribuicao: ID OPATRIB expressao FLINHA;

incrementoOuDecremento: ID OPINCDEC FLINHA;

chamadaFuncao: ID AP argumentos? FP;

escrita: ESCRITA expressao expressao? FLINHA;

leitura: LEITURA ID FLINHA;

retorno: RETORNO expressao FLINHA;

funcao: FUNCAO TIPO? ID AP parametrosDef? FP AC bloco FC;

condicional: SE AP condicao FP ENTAO AC bloco FC (SENAO AC bloco FC)?;

repeticao: ENQUANTO AP condicao FP AC bloco FC;

expressao: termo                       // Um termo básico
         | expressao OPMAT expressao   // Operações matemáticas
         | AP expressao FP             // Operações considerando precedência
         ;
```

```
condicao: expressao                    // Um termo básico
        | condicao OPREL condicao      // Operações relacionais
        | condicao OPLOG condicao      // Operações lógicas
        | AP condicao FP               // Operações considerando precedência
        ;

termo: ID
     | NUMINT
     | NUMREAL
     | chamadaFuncao
     | AP expressao FP
     | Str
     ;

parametros: expressao (',' expressao)*;

parametrosDef: ID (',' ID)*;

argumentos: expressao (',' expressao)*;

bloco: (instrucao)*;
```

8. IMPLEMENTAÇÃO DO ANALISADOR SINTÁTICO

O seguinte arquivo Main.java foi modificado para possibilitar a visualização da saída no terminal em formato de árvore sintática, gerada com base no código de entrada escrito na linguagem Tupy. Esse código de entrada é fornecido por meio de um arquivo cujo caminho é especificado na variável filename. O objetivo principal dessa modificação é integrar as etapas de análise léxica e sintática para validar a estrutura do programa e exibir sua árvore sintática correspondente.

Inicialmente, o programa lê o conteúdo do arquivo de entrada e o transforma em um fluxo de caracteres utilizando o método CharStreams.fromFileName(). Em seguida, esse fluxo é processado pelo analisador léxico, que identifica os elementos estruturais do código e os converte em tokens. Esses tokens são então encapsulados em um objeto CommonTokenStream, que serve como entrada para o analisador sintático. Com o analisador sintático configurado, o método parser.programa() é invocado. Esse método indica que o nó inicial da gramática é o símbolo programa, marcando o ponto de partida para a construção da árvore sintática. A estrutura gerada reflete os componentes e a hierarquia do programa analisado. Por fim, a árvore é exibida no terminal por meio do método toStringTree() da classe ParseTree, permitindo uma visualização detalhada da análise.

```java
import ...

public class Main {

    public static void main(String[] args) {
        String filename = "caminho do arquivo a ser analisado";

        try {
            CharStream input = CharStreams.fromFileName(filename);
            TupyLexer lexer = new TupyLexer(input);
            CommonTokenStream tokens = new CommonTokenStream(lexer);
            TupyParser parser = new TupyParser(tokens);

            ParseTree ast = parser.programa();

            System.out.println(ast.toStringTree());


        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

9. CASOS DE TESTE – SINTÁTICO

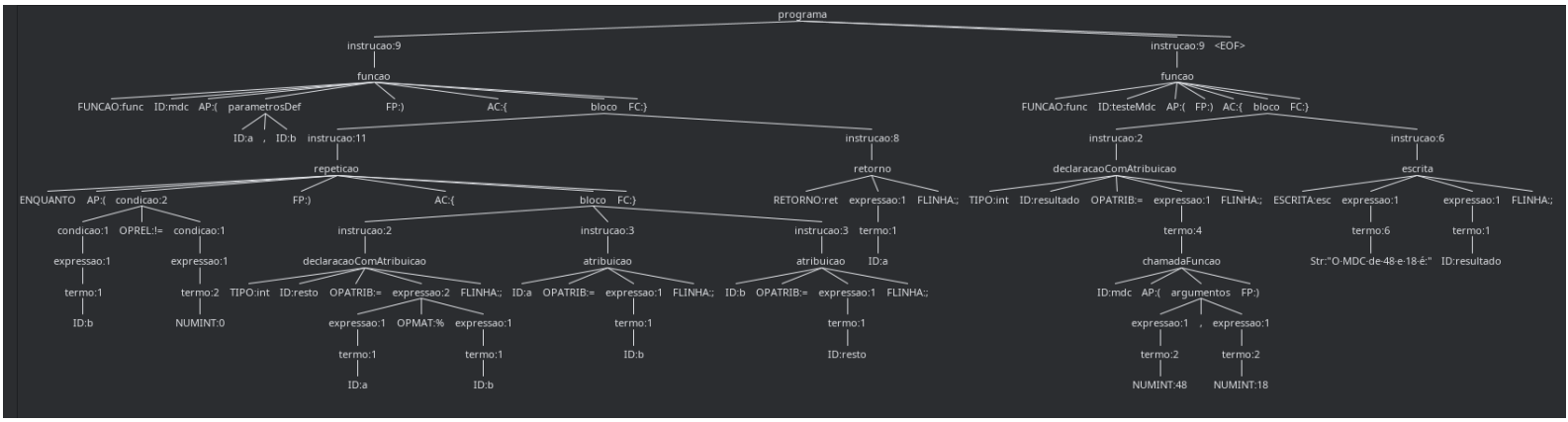
Na etapa de testes, o analisador sintático foi avaliado utilizando um código em formato .txt que implementa o cálculo do Máximo Divisor Comum (MDC) e um teste para a função mdc. Esses códigos, apresentados no documento, foram essenciais para validar a eficácia do analisador sintático desenvolvido. A implementação avaliada inclui a função mdc, que calcula o MDC utilizando um laço enquanto, e a função testeMdc, que verifica a correção do cálculo ao comparar os resultados para os valores 48 e 18.

Os resultados obtidos durante a execução demonstraram que o analisador sintático identificou corretamente as estruturas condicionais e de repetição, bem como as atribuições de valores realizadas dentro do escopo das funções. A análise confirmou que o código segue a sintaxe especificada e que o analisador responde conforme esperado para essas estruturas, apresentando a árvore de derivação correta.

```
func mdc(a, b) {
    enquanto (b != 0) {
        int resto = a % b;
        a = b;
        b = resto;
    }
    ret a;
}

func testeMdc() {
    int resultado = mdc(48, 18);
    esc "O MDC de 48 e 18 é:" resultado;
}
```

```
() ([40] ([55 40] func mdc ( ([107 55 40] a , b) ) { ([112 55 40] ([208 112 55 40] ([57 208 112 55 40] enquanto ( ([132 57 208 112 55 40] ([28 132 57 208 11

rocess finished with exit code 0
```

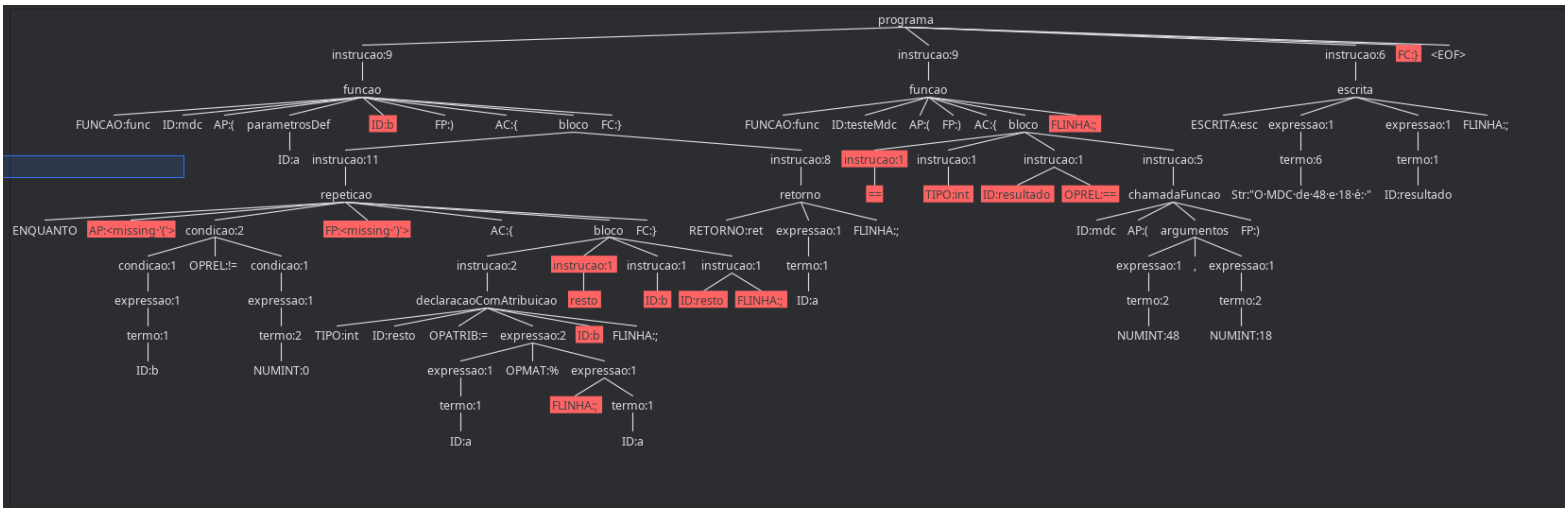
```
[139 87 52 208 112 55 40] "O MDC de 48 e 18 é:")) ([88 52 208 112 55 40] ([139 88 52 208 112 55 40] resultado)) ;))) })) <EOF>)
```



De forma semelhante, foi realizado um teste com um código de mesma lógica, mas que continha um erro sintático intencional. Esse teste teve como objetivo avaliar a capacidade do analisador sintático de identificar e reportar inconsistências na estrutura do código.

O código apresentado possui um erro proposital na sua sintaxe, desafiando o analisador a reconhecer desvios das regras especificadas. Durante a execução, o analisador não apenas identificou o erro, mas também gerou uma árvore de derivação, onde os erros foram destacados em vermelho. Esses marcadores visuais correspondem exatamente às partes do código que violam as regras de sintaxe, facilitando a localização e correção das falhas.

```
line 1:11 extraneous input 'b' expecting ')'
line 2:13 missing '(' at 'b'
line 2:20 missing ')' at '{'
line 3:23 extraneous input ';' expecting {'(', ID, NUMINT, NUMREAL, Str}
line 4:10 extraneous input 'b' expecting ';'
line 5:10 no viable alternative at input 'bresto'
line 11:18 no viable alternative at input 'intresultado=='
line 11:32 mismatched input ';' expecting {',', OPMAT, ')'}
line 13:0 extraneous input '}' expecting {<EOF>, TIPO, 'esc', 'lei', 'ret', 'func', 'se', 'enquanto', ID}
([] ([40] ([55 40] func mdc ( ([107 55 40] a) b ) { ([112 55 40] ([208 112 55 40] ([57 208 112 55 40] enquanto <missing '('> ([132 57 208 112 55 40] ([28 13
```

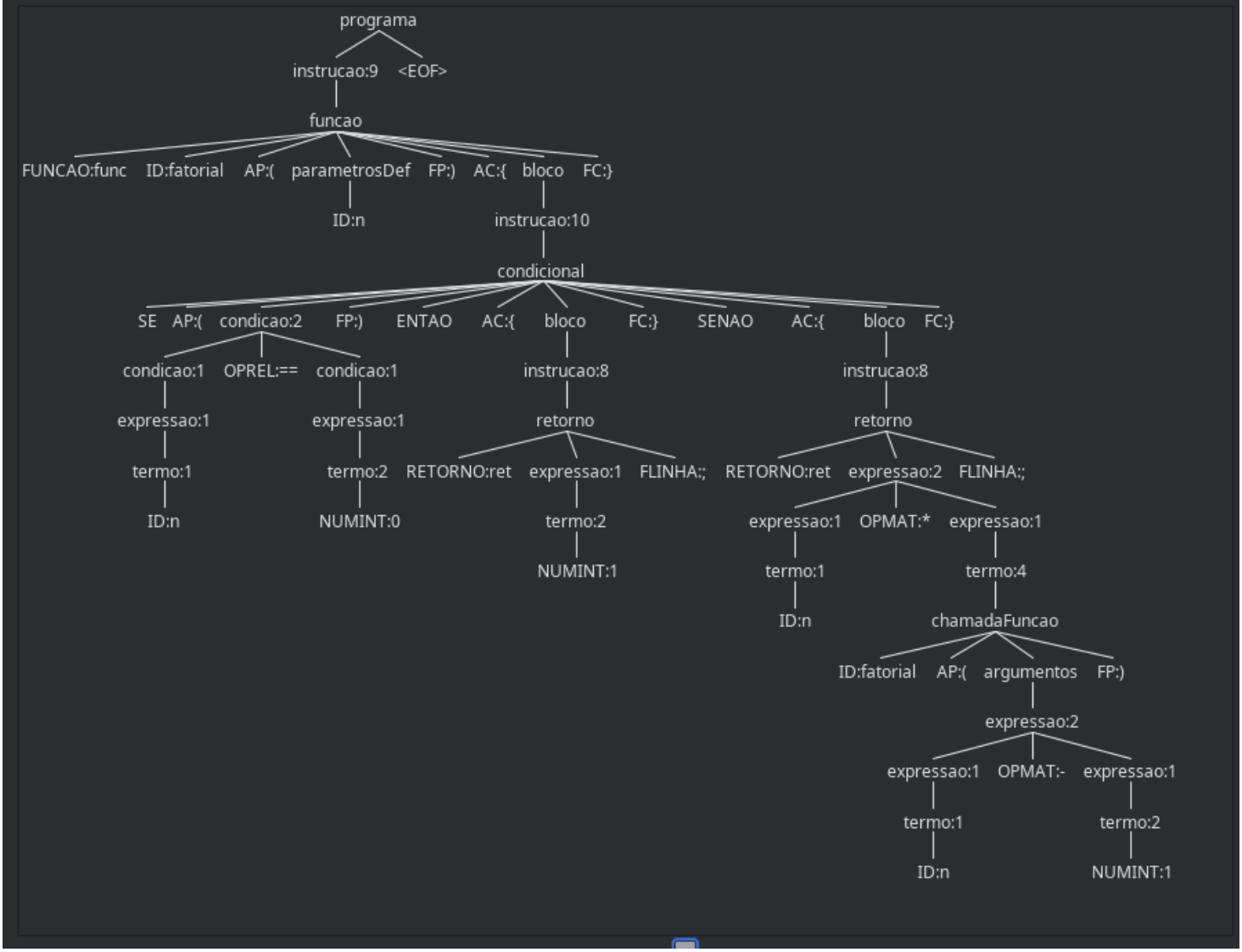



```
func mdc(a b) { // Erro:Falta a vírgula entre os parâmetros.
    enquanto b != 0 { // Erro: Parênteses ausentes na expressão condicional.
        int resto = a %; // Erro: Expressão incompleta.
        a b; // Erro: Operador de atribuição ausente.
        b resto; // Erro: Operador de atribuição ausente novamente.
    }
    ret a; // Correto.
}

func testeMdc() {
    int resultado == mdc(48, 18); // Erro: Uso de operador relacional em vez de atribuição.
    esc "O MDC de 48 e 18 é: " resultado; // Erro: Falta a vírgula para separar os argumentos.
}
```

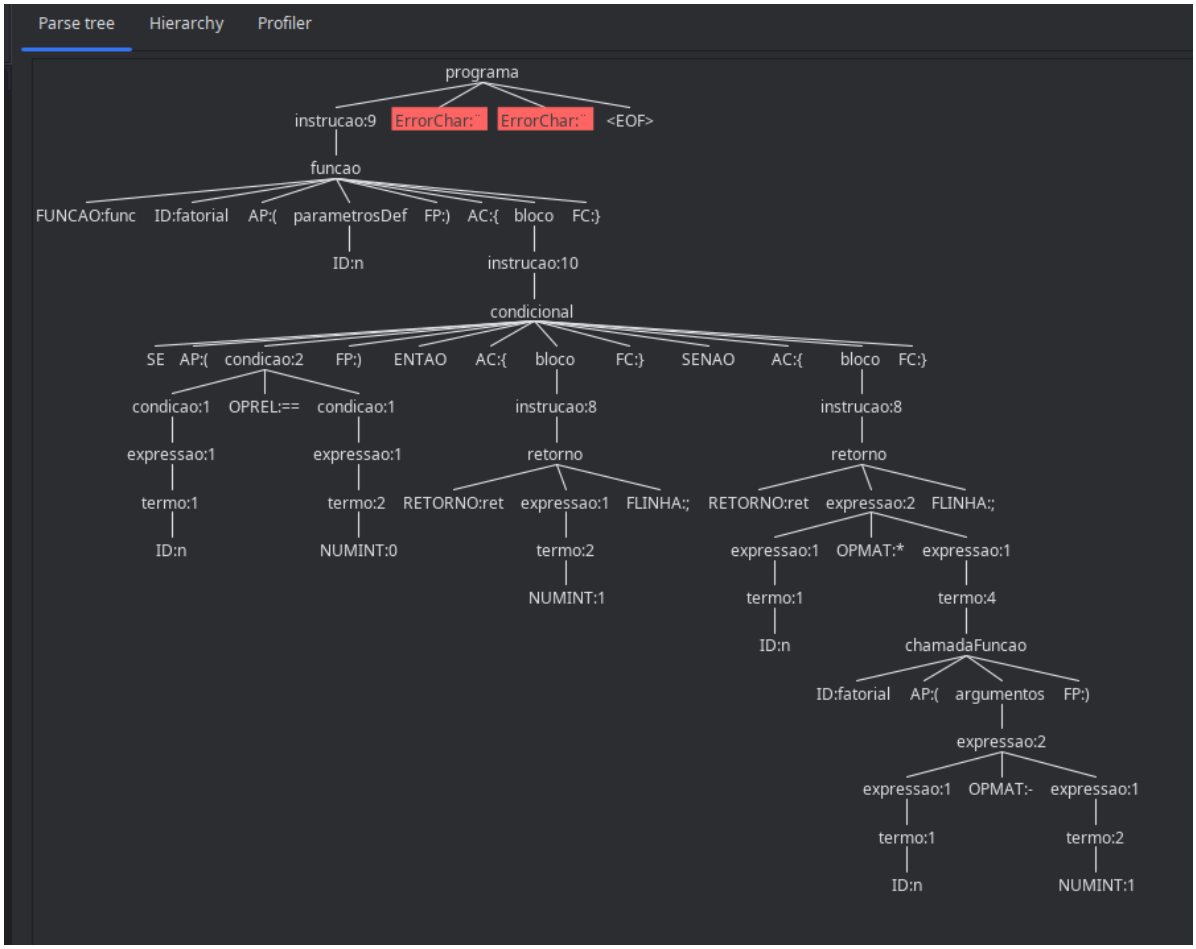
Foram realizados também testes com os códigos da etapa do analisador léxico, incluindo o código Fatorial, tanto em sua versão correta quanto em uma versão com erro.

```
/usr/lib/jvm/java-23-jdk/bin/java -javaagent:/usr/share/idea/lib/idea_rt.jar=43567:/usr/share/idea/bin -
([] ([40] ([55 40] func fatorial ( ([107 55 40] n) ) { ([112 55 40] ([208 112 55 40] ([56 208 112 55 40]

Process finished with exit code 0
```

Parse tree | Hierarchy | Profiler




```
/usr/lib/jvm/java-23-jdk/bin/java -javaagent:/usr/share/idea/lib/idea_rt.jar=44629:/usr/share/idea/bin -Dfile.
line 8:1 extraneous input '`' expecting {<EOF>, TIPO, 'esc', 'lei', 'ret', 'func', 'se', 'enquanto', ID}
([] ([40] ([55 40] func fatorial ( ([107 55 40] n) ) { ([112 55 40] ([208 112 55 40] ([56 208 112 55 40] se (

Process finished with exit code 0
```



10. DEFINIÇÃO SEMÂNTICA DA LINGUAGEM

Foi adicionado nesta etapa o código MyListener.java, que implementa uma parte fundamental da análise semântica de uma linguagem de programação. Este código tem o objetivo de realizar verificações semânticas no código fonte, validando o uso de variáveis, tipos e escopos dentro do programa. Durante o processo de análise, o ouvinte verifica se as variáveis são utilizadas de forma correta, se os tipos são compatíveis nas operações e se as regras de escopo e declaração estão sendo seguidas corretamente.

Uma das principais verificações realizadas pelo código é a checagem de tipo, que garante que as operações e atribuições sejam realizadas com variáveis de tipos compatíveis. Quando uma variável é usada em uma operação ou atribuída a um valor, o ouvinte valida se o tipo da variável corresponde ao tipo esperado para aquela operação. Além disso, o código realiza a checagem de variáveis não declaradas, onde o ouvinte verifica se todas as variáveis que estão sendo utilizadas

no programa foram previamente declaradas em algum escopo válido. Caso uma variável seja usada sem declaração, um erro é gerado. A checagem de declarações duplicadas de variáveis também é feita, onde o ouvinte verifica se uma variável foi declarada mais de uma vez dentro do mesmo escopo, o que resultaria em um erro semântico. Finalmente, o código cuida da checagem de escopo de variáveis, assegurando que as variáveis sejam acessadas apenas dentro do escopo no qual foram declaradas, respeitando as regras de visibilidade e evitando conflitos de nomes.

Essas verificações são cruciais para garantir que o código esteja sem erros semânticos e siga as regras da linguagem de programação, ajudando na construção de programas corretos e robustos. O código implementado, portanto, se concentra nessas questões semânticas, como a checagem de tipo, a validação de variáveis declaradas corretamente, o controle de escopos e a detecção de declarações duplicadas de variáveis, conforme o código abaixo mostra.

```java
import java.util.HashMap;
import java.util.Map;
import java.util.Stack;
import org.antlr.v4.runtime.Token;

public class MyListener extends TupyBaseListener {  2 usages

    // Pilha de escopos: cada escopo é uma tabela de símbolos (mapa de variável -> tipo)
    private Stack<Map<String, String>> escopos = new Stack<>();  8 usages
    private String tipoElemento = null;  11 usages

    // Construtor: cria o escopo global
    public MyListener() { escopos.push(new HashMap<>()); }

    // Ao entrar em um bloco, cria um novo escopo
    @Override  1 usage
    public void enterBloco(TupyParser.BlocoContext ctx) { escopos.push(new HashMap<>()); }

    // Ao sair de um bloco, descarta o escopo atual
    @Override  1 usage
    public void exitBloco(TupyParser.BlocoContext ctx) { escopos.pop(); }

    @Override  1 usage
    public void exitInstrucao(TupyParser.InstrucaoContext ctx) {
        System.out.println("Instrução: " + ctx.getText());

        // Caso seja uma declaração com atribuição: TIPO ID OPATRIB expressao FLINHA;
        if (ctx.declaracaoComAtribuicao() != null) {
            String id = ctx.declaracaoComAtribuicao().ID().getText();
            String tipo = ctx.declaracaoComAtribuicao().TIPO().getText();
            if (!verificarVarDuplicacao(id, ctx.start)) {
                if (!verificarTipoIncompativelExpressao(id, tipo, ctx.start)) {
                    getEscopoAtual().put(id, tipo);
                }
            }
        }
```

```java
    // Caso seja uma declaração sem atribuição: TIPO ID FLINHA;
    else if (ctx.declaracao() != null) {
        String id = ctx.declaracao().ID().getText();
        String tipo = ctx.declaracao().TIPO().getText();
        if (!verificarVarDuplicacao(id, ctx.start)) {
            getEscopoAtual().put(id, tipo);
        }
    }
    // Caso seja uma atribuição: ID OPATRIB expressao FLINHA;
    else if (ctx.atribuicao() != null) {
        String id = ctx.atribuicao().ID().getText();
        if (!verificarVarNaoDeclarada(id, ctx.start)) {
            verificarTipoIncompativelExpressao(id, tipo: null, ctx.start);
        }
    }
    // Caso seja um incremento/decremento: ID OPINCDEC FLINHA;
    else if (ctx.incrementoOuDecremento() != null) {
        String id = ctx.incrementoOuDecremento().ID().getText();
        if (!verificarVarNaoDeclarada(id, ctx.start)) {
            verificarTipoIncompativelOpIncDec(id, ctx.start);
        }
    }
    // Caso seja uma leitura: LEITURA ID FLINHA;
    else if (ctx.leitura() != null) {
        String id = ctx.leitura().ID().getText();
        if (!verificarVarNaoDeclarada(id, ctx.start)) {
            verificarTipoIncompativelTexto(id, ctx.start);
        }
    }
    // Outros casos (chamada de função, escrita, retorno, etc.) podem ser tratados conforme necessário.

    // Reinicia o tipoElemento após processar a instrução
    tipoElemento = null;
}

@Override  1 usage
public void enterExpressao(TupyParser.ExpressaoContext ctx) {
    // Caso a expressão seja simples (apenas um termo)
    if (ctx.termo() != null && ctx.getChildCount() == 1) {
        if (tipoElemento == null) {
            tipoElemento = verificarTipoTermo(ctx.termo());
        } else if (!tipoElemento.equals("indefinido")) {
            tipoElemento = verificarTipoTermo(ctx.termo());
        }
    }
```

```
88          // Expressão com operação aritmética: expressao OPMAT expressao
89          else if (ctx.OPMAT() != null && ctx.getChildCount() == 3) {
90              String tipo1 = "indefinido";
91              String tipo2 = "indefinido";
92              if (ctx.expressao( i: 0) != null && ctx.expressao( i: 0).termo() != null) {
93                  tipo1 = verificarTipoTermo(ctx.expressao( i: 0).termo());
94              }
95              if (ctx.expressao( i: 1) != null && ctx.expressao( i: 1).termo() != null) {
96                  tipo2 = verificarTipoTermo(ctx.expressao( i: 1).termo());
97              }
98              if (!tipo1.equals("indefinido") && tipo2.equals("indefinido")) {
99                  tipoElemento = tipo1;
100             } else if (tipo1.equals("indefinido") && !tipo2.equals("indefinido")) {
101                 tipoElemento = tipo2;
102             } else if (!tipo1.equals(tipo2)) {
103                 tipoElemento = "indefinido";
104             } else {
105                 tipoElemento = tipo1;
106             }
107         }
108         // Expressão parentizada: AP expressao FP
109         else if (ctx.getChildCount() == 3
110                 && ctx.getChild( i: 0).getText().equals("(")
111                 && ctx.getChild( i: 2).getText().equals(")")) {
112             // A expressão interna já será processada.
113         }
114     }
115
116     // Verifica o tipo de um termo (equivalente à antiga função verificarTipoElemento)
117 @   private String verificarTipoTermo(TupyParser.TermoContext ctx) {  4 usages
118         if (ctx.NUMINT() != null) {
119             return "int";
120         } else if (ctx.NUMREAL() != null) {
121             return "real";
122         } else if (ctx.Str() != null) {
123             return "lin";
124         } else if (ctx.ID() != null) {
125             String id = ctx.ID().getText();
126             if (verificarVarNaoDeclarada(id, ctx.start)) {
127                 return "indefinido";
128             } else {
129                 return getTipoVariavel(id);
130             }
131         }
132         return "indefinido";
```

```
133         }
134
135         // Checagem de variável duplicada (somente no escopo atual)
136         private boolean verificarVarDuplicacao(String id, Token start) {  2 usages
137             if (getEscopoAtual().containsKey(id)) {
138                 System.out.println("Declaração duplicada! Variável " + id + " já foi declarada neste escopo!");
139                 imprimirErro(start);
140                 return true;
141             } else {
142                 return false;
143             }
144         }
145
146         // Checagem de variável não declarada (procura em todos os escopos)
147         private boolean verificarVarNaoDeclarada(String id, Token start) {  4 usages
148             if (getTipoVariavel(id) == null) {
149                 System.out.println("Variável " + id + " não foi declarada!");
150                 imprimirErro(start);
151                 return true;
152             } else {
153                 return false;
154             }
155         }
156
157         // Checagem de tipos incompatíveis na atribuição/expressão
158         private boolean verificarTipoIncompativelExpressao(String id, String tipo, Token start) {  2 usages
159             if (tipo == null) {
160                 tipo = getTipoVariavel(id);
161             }
162             if (!tipo.equals(tipoElemento)) {
163                 imprimirTipo(id, tipo);
164                 System.out.println("\tTipo da expressão: " + tipoElemento);
165                 imprimirErro(start);
166                 return true;
167             } else {
168                 return false;
169             }
170         }
171
172         // Checagem de tipos incompatíveis para incremento/decremento
173         private void verificarTipoIncompativelOpIncDec(String id, Token start) {  1 usage
174             String tipo = getTipoVariavel(id);
175             if (!tipo.equals("int") && !tipo.equals("real")) {
176                 imprimirTipo(id, tipo);
177                 System.out.println("\tTipo esperado: int ou real");
```

```
            imprimirTipo(id, tipo);
            System.out.println("\tTipo esperado: int ou real");
            imprimirErro(start);
        }
    }

    // Checagem de tipos incompatíveis para operações de leitura (texto)
    private void verificarTipoIncompativelTexto(String id, Token start) {  1 usage
        String tipo = getTipoVariavel(id);
        if (!tipo.equals("lin")) {
            imprimirTipo(id, tipo);
            System.out.println("\tTipo esperado: lin");
            imprimirErro(start);
        }
    }

    // Função auxiliar para imprimir mensagens de tipos incompatíveis
    private void imprimirTipo(String id, String tipo) {  3 usages
        System.out.println("Tipo incompatível para a variável/valor: " + id);
        System.out.println("\tTipo de " + id + ": " + tipo);
    }

    // Função auxiliar para imprimir informações de erro (linha e posições)
    private void imprimirErro(Token start) {  5 usages
        System.out.println("\tLinha: " + start.getLine());
        System.out.println("\tPos Inicial: " + start.getStartIndex());
        System.out.println("\tPos Final: " + start.getStopIndex());
    }

    // Retorna o escopo atual (topo da pilha)
    private Map<String, String> getEscopoAtual() { return escopos.peek(); }

    // Procura pelo tipo de uma variável em todos os escopos (do mais interno para o mais externo)
    private String getTipoVariavel(String id) {  5 usages
        for (int i = escopos.size() - 1; i >= 0; i--) {
            if (escopos.get(i).containsKey(id)) {
                return escopos.get(i).get(id);
            }
        }
        return null;
    }

    // Retorna o escopo global (primeiro da pilha)
    public Map<String, String> getTabelaSimbolos() { return escopos.firstElement(); }
}
```

11. IMPLEMENTAÇÃO DO ANALISADOR SEMÂNTICO

Da mesma forma foi adicionado o código AnalisadorSemantico.java, que tem a função de executar a análise semântica do código fonte utilizando o MyListener.java. O principal objetivo deste código é integrar o processo de leitura e análise do código com as verificações semânticas implementadas no listener. Ele

começa recebendo o caminho de um arquivo de código-fonte, que é passado para o parser. O parser então analisa o código e gera uma árvore sintática abstrata (AST), que é a estrutura de dados usada para representar a estrutura do programa.

Após a geração da AST, o código cria uma instância do MyListener e a utiliza para realizar a verificação semântica do programa. A árvore sintática é passada para o ParseTreeWalker, que caminha por cada nó da árvore e aplica as verificações semânticas definidas no MyListener. Esse processo permite a validação de variáveis, tipos e escopos conforme o programa é analisado. Ao final, o código imprime a tabela de símbolos, que contém as variáveis e seus respectivos tipos, como um resumo do que foi validado durante a execução.

O código também trata exceções relacionadas ao arquivo de entrada, como quando o caminho do arquivo é inválido ou o arquivo não é encontrado, garantindo que o programa não falhe inesperadamente. A função getParser é responsável por configurar e inicializar o lexer e o parser, que são necessários para transformar o código-fonte em uma árvore sintática. Em resumo, o código AnalisadorSemantico.java serve para integrar o processo de análise sintática e semântica, utilizando o MyListener.java para realizar a validação semântica, conforme o código abaixo mostra.

```java
import java.io.IOException;
import java.nio.file.NoSuchFileException;

import org.antlr.v4.runtime.CommonTokenStream;
import org.antlr.v4.runtime.tree.ParseTree;
import org.antlr.v4.runtime.tree.ParseTreeWalker;
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.CharStreams;

public class AnalisadorSemantico {
    public static void main(String[] args) {

        String filename = "insira caminho";

        TupyParser parser = getParser(filename);

        if(parser != null) {
            ParseTree ast = parser.programa();

            MyListener listener = new MyListener();

            ParseTreeWalker walker = new ParseTreeWalker();

            walker.walk(listener, ast);

            System.out.println(listener.getTabelaSimbolos().toString());
        }

    }
```

```
30
31      private static TupyParser getParser(String filename) {  1 usage
32          TupyParser parser = null;
33
34          try {
35              CharStream input = CharStreams.fromFileName(filename);
36              TupyLexer lexer = new TupyLexer(input);
37              CommonTokenStream tokens = new CommonTokenStream(lexer);
38              parser = new TupyParser(tokens);
39          }
40          catch(NoSuchFileException e) {
41              System.out.println("Nome inválido ou não existe!");
42          }
43          catch(IOException e) {
44              e.printStackTrace();
45          }
46
47          return parser;
48      }
49  }
```

12. CASOS DE TESTE – SEMÂNTICO

Na parte de testes, foram utilizados dois códigos, chamados Fatorial e Fatorial (Com erros), para validar a eficácia do analisador semântico. O primeiro código, Fatorial, está correto e não apresenta problemas semânticos. Nele, todas as variáveis estão devidamente declaradas, os tipos estão compatíveis e não há conflitos relacionados a escopos ou declarações duplicadas. Como resultado, o código é processado sem erros, e a tabela de símbolos gerada pelo MyListener contém as informações corretas das variáveis, como esperado.

Já no segundo código, Fatorial (Com erros), são introduzidos dois tipos de erros. O primeiro é um erro de checagem de tipo, onde ocorre uma incompatibilidade de tipo inteiro na variável “n”, da qual recebe um valor não inteiro. O segundo erro é uma checagem de variáveis não declaradas, em que a variável “n”, por ter um erro já estabelecido, não é declarada, o que gera uma falha semântica. Esses erros são detectados pelo MyListener, que emite mensagens detalhadas de erro, indicando a linha e a posição do código onde os problemas ocorrem.

As imagens que serão apresentadas a seguir ilustram como essas verificações funcionam e como o sistema identifica os erros nos testes realizados.

- **Fatorial**

```
int n = 0;

func fatorial(n){
    se (n == 0) entao {
        ret 1;
    }
    senao {
        ret n * fatorial(n - 1);
    }
}

```

```
/usr/lib/jvm/java-23-jdk/bin/java -javaagent:/usr/share/idea/lib/idea_rt.jar=33017:/usr/share/idea/bi
Instrução: intn=0;
Instrução: retl;
Instrução: retn*fatorial(n-1);
Instrução: se(n==0)entao{retl;}senao{retn*fatorial(n-1);}
Instrução: funcfatorial(n){se(n==0)entao{retl;}senao{retn*fatorial(n-1);}}
{n=int}

Process finished with exit code 0
```

- **Fatorial (Com erros)**

```
int n = 0,5;

func fatorial(n){
    se (n == 0) entao {
        ret 1;
    }
    senao {
        ret n * fatorial(n - 1);
    }
}

```

```
/usr/lib/jvm/java-23-jdk/bin/java -javaagent:/usr/share/idea/lib/idea_rt.jar=3699
Instrução: intn=0,5;
Tipo incompatível para a variável/valor: n
    Tipo de n: int
    Tipo da expressão: real
    Linha: 1
    Pos Inicial: 0
    Pos Final: 2
Variável n não foi declarada!
    Linha: 4
    Pos Inicial: 40
    Pos Final: 40
Instrução: retl;
Variável n não foi declarada!
    Linha: 8
    Pos Inicial: 88
    Pos Final: 88
Variável n não foi declarada!
    Linha: 8
    Pos Inicial: 101
    Pos Final: 101
Variável n não foi declarada!
    Linha: 8
    Pos Inicial: 101
    Pos Final: 101
Instrução: retn*fatorial(n-1);
Instrução: se(n==0)entao{retl;}senao{retn*fatorial(n-1);}
Instrução: funcfatorial(n){se(n==0)entao{retl;}senao{retn*fatorial(n-1);}}
{}

Process finished with exit code 0
```

De maneira semelhante aos testes anteriores, foi realizada uma nova série de testes para validar a checagem de declarações duplicadas de variáveis e a checagem de escopo de variáveis, utilizando os códigos MDC e MDC (Com erros). No código MDC, que está correto, não foram encontrados problemas semânticos. Porém, no código MDC (Com erros), foram introduzidos dois tipos de erros. O primeiro erro identificado foi o de declaração duplicada de variáveis, em que a variável "b" foi declarada novamente dentro do mesmo escopo, o que violou as regras da linguagem. O segundo erro foi um erro de escopo de variável, no qual a variável "resto" foi acessada fora do escopo da função "mdc" onde ela foi originalmente declarada. Esse tipo de erro ocorre quando uma variável é usada em

um contexto onde não deveria ser acessível, como no caso de variáveis locais a uma função, que não podem ser acessadas fora dela.

As imagens a seguir ilustram os sucessos e erros encontrados nos testes realizados.

- **MDC**

```
int a = 2;
int b = 5;

func mdc(a, b) {
    enquanto (b != 0) {
        int resto = a % b;
        a = b;
        b = resto;
    }
    ret a;
}
```

```
/usr/lib/jvm/java-23-jdk/bin/java -javaagent:/usr/share/idea/lib/idea_rt.jar=3423
Instrução: inta=2;
Instrução: intb=5;
Instrução: intresto=a%b;
Instrução: a=b;
Instrução: b=resto;
Instrução: enquanto(b!=0){intresto=a%b;a=b;b=resto;}
Instrução: reta;
Instrução: funcmdc(a,b){enquanto(b!=0){intresto=a%b;a=b;b=resto;}reta;}
{a=int, b=int}

Process finished with exit code 0
```

- **MDC (Com erros)**

```
int a = 2;
int b = 5;

func mdc(a, b) {
    enquanto (b != 0) {
        int resto = a % b;
        a = b;
        b = resto;
    }
    ret a;
}

int b = 13;
resto = 10;
```

```
/usr/lib/jvm/java-23-jdk/bin/java -javaagent:/usr/share/idea/lib/idea_rt.ja
Instrução: inta=2;
Instrução: intb=5;
Instrução: intresto=a%b;
Instrução: a=b;
Instrução: b=resto;
Instrução: enquanto(b!=0){intresto=a%b;a=b;b=resto;}
Instrução: reta;
Instrução: funcmdc(a,b){enquanto(b!=0){intresto=a%b;a=b;b=resto;}reta;}
Instrução: intb=13;
Declaração duplicada! Variável b já foi declarada neste escopo!
    Linha: 13
    Pos Inicial: 157
    Pos Final: 159
Instrução: resto=10;
Variável resto não foi declarada!
    Linha: 14
    Pos Inicial: 170
    Pos Final: 174
{a=int, b=int}

Process finished with exit code 0
```

13. LINK DO GITHUB

https://github.com/Gabseek/Projeto-Compiladores