RELATÓRIO TÉCNICO
COMPILADORES

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

LAVRAS
2024

COMPILADORES

RELATÓRIO TÉCNICO

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

LAVRAS
2024

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.

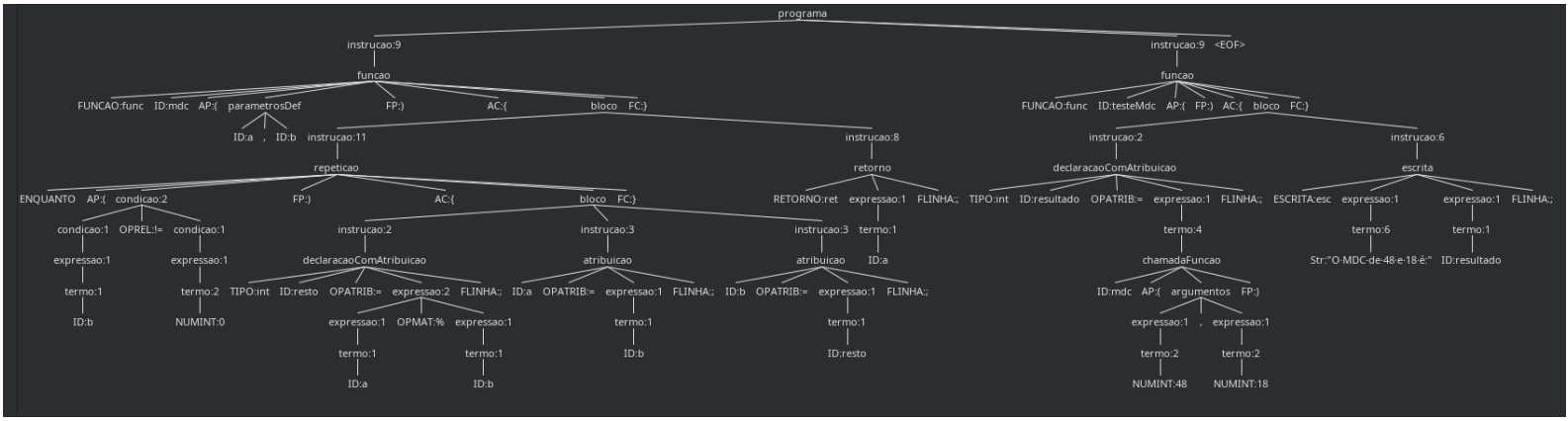
```
func mdc(a, b) {
    enquanto (b != 0) {
        int resto = a % b;
        a = b;
        b = resto;
    }
    ret a;
}

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
```

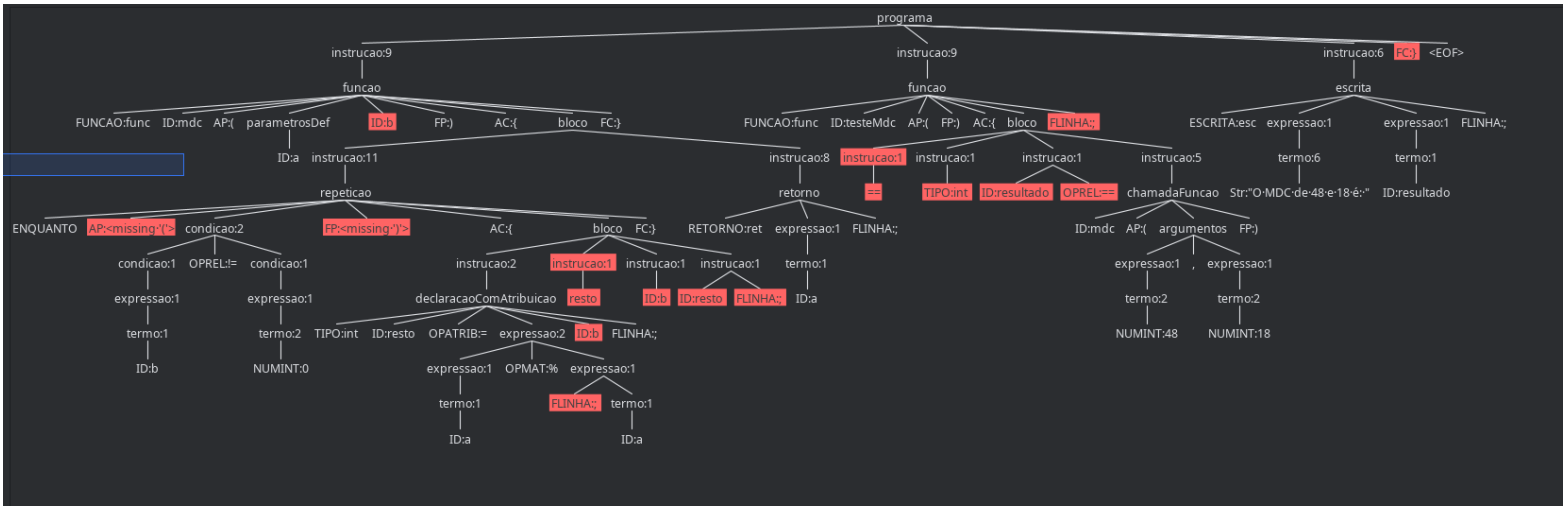
```
[139 87 52 208 112 55 40] "O MDC de 48 e 18 é:")) ([88 52 208 112 55 40] ([139 88 52 208 112 55 40] resultado)) ;))) })) <EOF>)
```



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
```



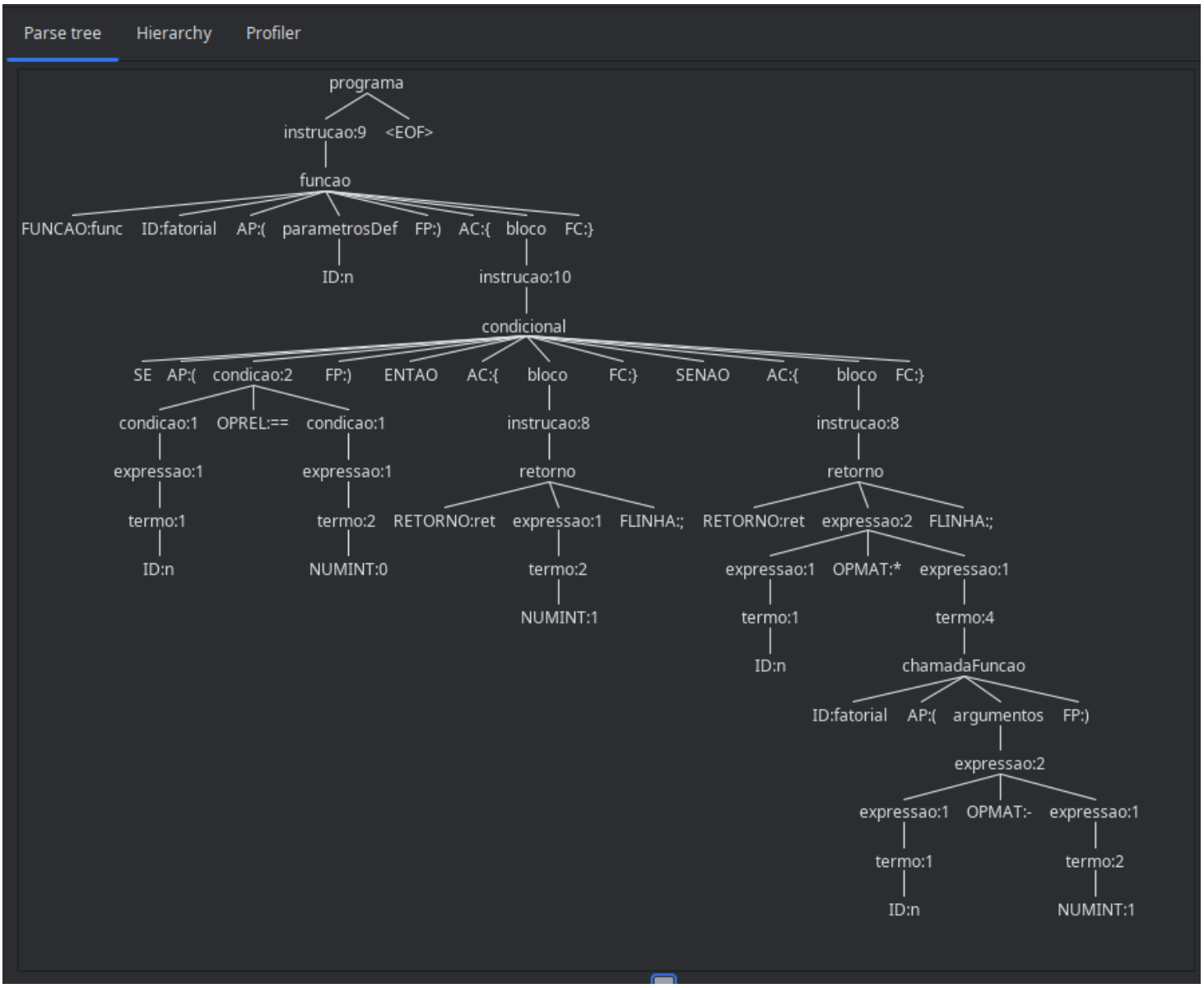

```
func mdc(a b) { // Erro:Falta a vírgula entre os parâmetros.
    enquanto b != 0 { // Erro: Parênteses ausentes na expressão condicional.
        int resto = a %; // Erro: Expressão incompleta.
        a b; // Erro: Operador de atribuição ausente.
        b resto; // Erro: Operador de atribuição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

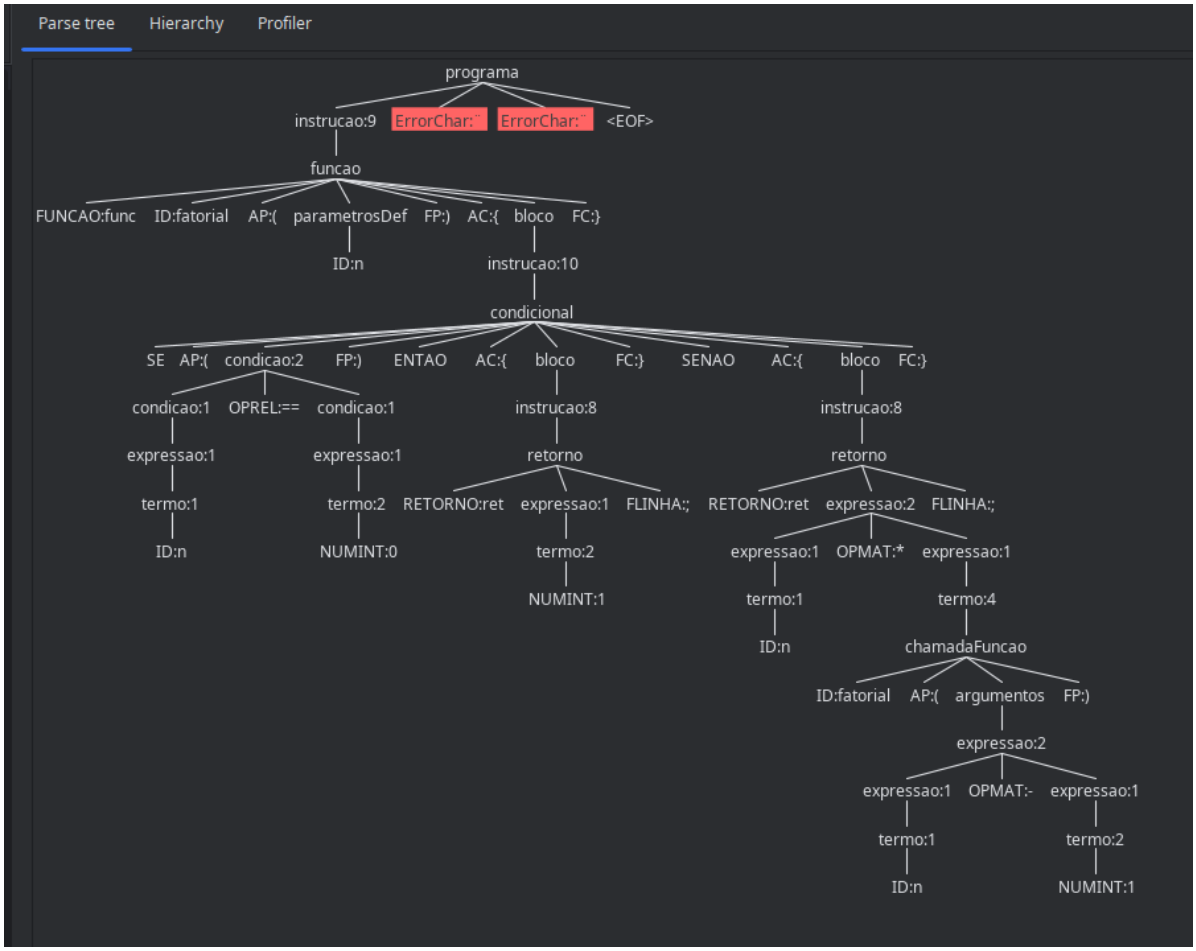
```
/usr/lib/jvm/java-23-jdk/bin/java -javaagent:/usr/share/idea/lib/idea_rt.jar=43567:/usr/share/idea/bin -
([] ([40] ([55 40] func fatorial ( ([107 55 40] n) ) { ([112 55 40] ([208 112 55 40] ([56 208 112 55 40]

Process finished with exit code 0
```




```
/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
```


Parse tree
Hierarchy
Profiler
programa
instrucao:9
ErrorChar:¨
ErrorChar:¨
<EOF>
funcao
FUNCAO:func
ID:fatorial
AP:(
parametrosDef
FP:)
AC:{
bloco
FC:}
ID:n
instrucao:10
condicional
SE
AP:(
condicao:2
FP:)
ENTAO
AC:{
bloco
FC:}
SENAO
AC:{
bloco
FC:}
condicao:1
OPREL:==
condicao:1
instrucao:8
instrucao:8
expressao:1
expressao:1
retorno
retorno
termo:1
termo:2
RETORNO:ret
expressao:1
FLINHA:;
RETORNO:ret
expressao:2
FLINHA:;
ID:n
NUMINT:0
termo:2
expressao:1
OPMAT:*
expressao:1
NUMINT:1
termo:1
termo:4
ID:n
chamadaFuncao
ID:fatorial
AP:(
argumentos
FP:)
expressao:2
expressao:1
OPMAT:-
expressao:1
termo:1
termo:2
ID:n
NUMINT:1

10. LINK DO GITHUB

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