RELATÓRIO TÉCNICO - ANALISADOR LÉXICO COMPILADORES

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

LAVRAS
2024

COMPILADORES

ANALISADOR LÉXICO

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 abrangente de uma linguagem de programação que será criada pelo grupo, destacando suas características principais e aspectos léxicos. Serão abordados os elementos fundamentais que compõem a linguagem, incluindo a definição de lexemas e suas classificações. Através de exemplos práticos, o relatório ilustrará a aplicação da linguagem em algoritmos específicos desenvolvidos pelo grupo. Também será discutida a construção de um analisador léxico, incluindo suas etapas de implementação e os resultados obtidos em casos de teste. O objetivo é fornecer uma compreensão clara das capacidades e da estrutura da linguagem proposta.

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 java.io.IOException;

public class Main {

    public static void main(String[] args) {
        String filename = "C:\\Compiladores\\v1\\codigo.txt";
        try{
            CharStream input = CharStreams.fromFileName(filename);
            TupyLexer lexer = new TupyLexer(input);
            Token token;
            while (!lexer._hitEOF){
                token = lexer.nextToken();
                System.out.println("Token: <Classe: "
                + lexer.getVocabulary().getSymbolicName(token.getType())
                +" ,Lexema: "+ token.getText() +">");
            }

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

6. CASOS DE TESTE

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: }>
```

Por fim, 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>>
```