Universidade Federal de Santa Catarina - UFSC
Departamento de Informática e Estatística
Curso de Ciências da Computação
INE5426 - Construção de Compiladores

Luiz João Carvalhaes Motta
Maria Eduarda de Melo Hang
Marina Pereira das Neves Guidolin

Relatório EP1 - Analisador Léxico

Florianópolis, 25 de Setembro de 2020

1 INTRODUÇÃO

Esse trabalho possui como objetivo a implementação de um Analisador Léxico para a linguagem CC-2020-1 conforme proposto no enunciado do Exercício Programa 1.

Para a construção do Analisador Léxico, o grupo optou pelo uso da ferramenta ANTLR versão 4.

2 IDENTIFICAÇÃO DOS TOKENS

A identificação dos tokens deu-se a partir da análise da gramática da linguagem CC-2020-1. Inicialmente, foram analisados todos os símbolos descritos como terminais. A partir disso, foi possível separar em qual categoria cada símbolo se encaixava.

Dessa maneira, os tokens foram separados em Palavras Reservadas, Tipos, Símbolos e Pontuação, Operadores Aritméticos, Operadores Relacionais e Tokens não triviais, cujos tokens são os Identificadores, Números inteiros e de ponto flutuante, e Strings. Além disso, criamos uma regra de identificação para espaços em branco.

2.1 PALAVRAS RESERVADAS

```
IF: 'if';
FOR: 'for';
ELSE: 'else';
RETURN: 'return';
READ: 'read';
PRINT: 'print';
NEW: 'new';
BREAK: 'break';
DEF: 'def';
```

2.2 TIPOS

```
TYPE_STRING: 'string';
TYPE_FLOAT: 'float';
TYPE_INT: 'int';
NULL: 'null';
```

2.3 SÍMBOLOS E PONTUAÇÃO

```
LPAREN : '(';
RPAREN : ')';
LBRACE : '{';
RBRACE : '}';
LBRACK : '[';
RBRACK : ']';
SEMI : ';';
COMMA : ',';
DOT : '.';
COLON : ':';
```

2.4 OPERADORES ARITMÉTICOS

```
ASSIGN : '=';
ADD : '+';
SUB : '-';
MUL : '*';
DIV : '/';
MOD : '%';
```

2.5 OPERADORES RELACIONAIS

```
GT : '>';
LT : '<';
EQUAL : '==';
LE : '<=';
GE : '>=';
NOTEQUAL : '!=';
```

2.6 TOKENS NÃO TRIVIAIS

Para identificar um número inteiro, definimos que esse deve ser formado por um ou mais dígitos. Para os números de ponto flutuante estabelecemos que será composto por zero ou mais dígitos, seguido por um ponto (‘.’) e, por fim, um ou mais dígitos.

```
INT: DIGITS;
FLOAT: DIGITS '.' DIGITS | '.' DIGITS;
```

As Strings foram determinadas como qualquer caractere dentro de aspas duplas, exceto o próprio caractere de aspas duplas. Por fim, para os identificadores, definimos que esse token deve começar com uma letra ou underline (‘_’) , podendo incluir depois zero ou mais letras, underlines ou dígitos.

```
STRING: '"' ~('"')* '"';
IDENT: [a-zA-Z_] ( [a-zA-Z0-9_] )*;
```

2.7 ESPAÇO EM BRANCO

Para os espaços em branco consideramos os símbolos de quebra de linha (‘\n’), tabulação (‘\t’) e nova linha (‘\r’) e aplicamos o método *skip* para o analisador léxico descartar os tokens formados.

```
WHITESPACE: [ \t\r\n]+ -> skip;
```

3 DEFINIÇÕES REGULARES

As definições regulares utilizadas no trabalho se encontram logo abaixo. As definições regulares estão em itálico e os símbolos terminais entre aspas simples, como por exemplo: ‘0’, ‘ “ ‘ e ‘new’.

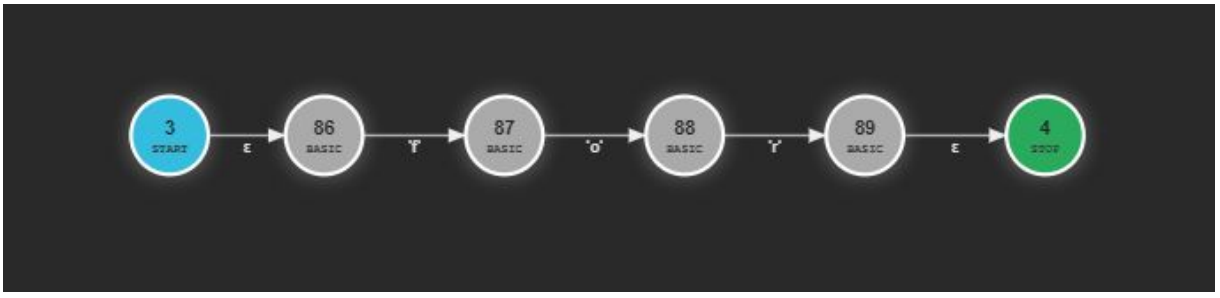
*letter_* → [a-zA-Z_]
digit → [0-9]
digits → *digit*+
int_constant → *digits*
string_constant → ‘ “ ‘ ~[‘ “ ‘]* ‘ “ ‘
float_constant → *digits* ‘.’ *digits* | ‘.’ *digits*
identifier → *letter_* (*letter_* | *digit*)*
white_space → (‘\n’ | ‘\t’ | ‘\r’)+
if → ‘if’
for → ‘for’
else → ‘else’
return → ‘return’
read → ‘read’
print → ‘print’
new → ‘new’
break → ‘break’
def → ‘def’
type_string → ‘string’
type_int → ‘int’
type_float → ‘float’
null → ‘null’
lparen → ‘(‘
rparen → ‘)’
lbrace → ‘{‘
rbrace → ‘}’
lbrack → ‘[‘
rbrack → ‘]’
semi → ‘;’
comma → ‘,’
dot → ‘.’
colon → ‘:’
assign → ‘=’
add → ‘+’
sub → ‘-’
mul → ‘*’
div → ‘/’
mod → ‘%’
gt → ‘>’
lt → ‘<’
equal → ‘==’
le → ‘<=’
ge → ‘>=’
notequal → ‘!=’

4 DIAGRAMAS DE TRANSIÇÃO

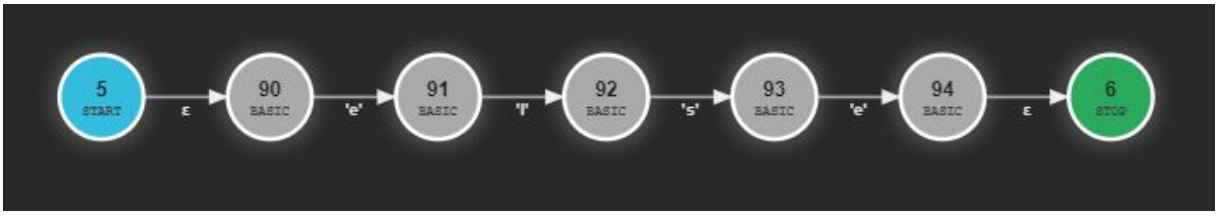
A seguir estão identificados os diagramas de transição para cada token identificado

4.1 TOKEN IF

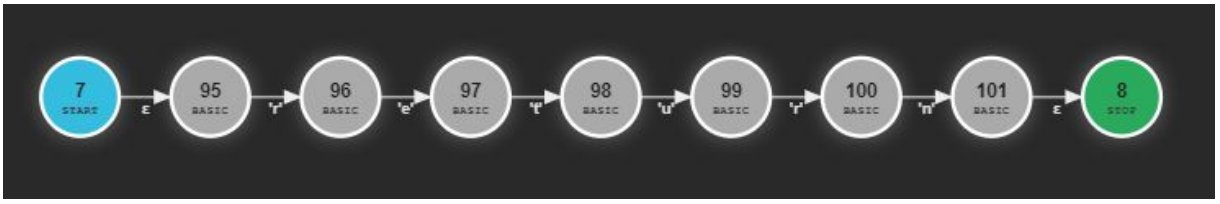
4.2 TOKEN FOR



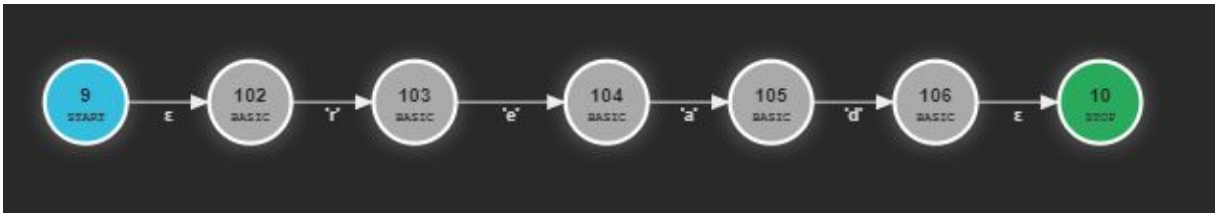
4.3 TOKEN ELSE



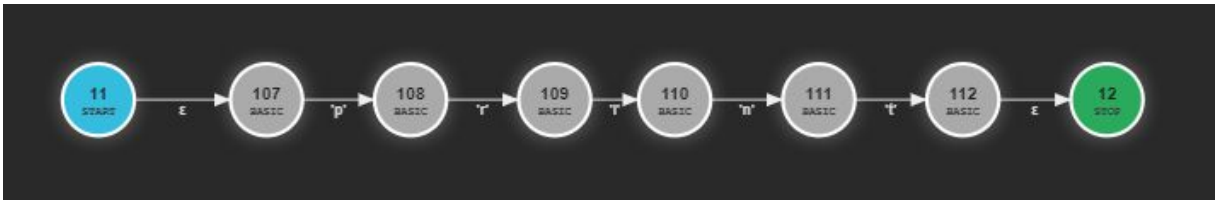
4.4 TOKEN RETURN



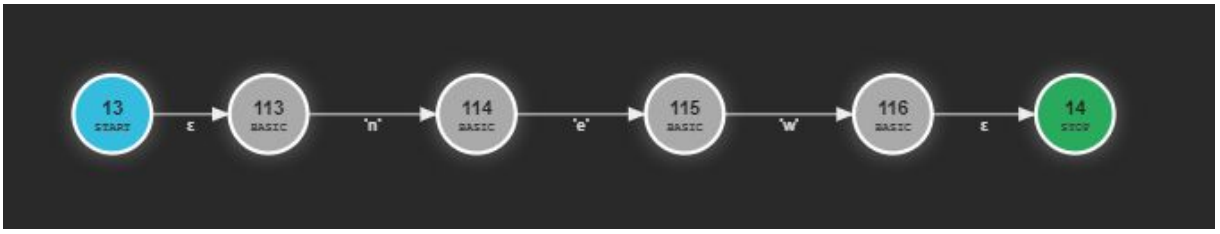
4.5 TOKEN READ



4.6 TOKEN PRINT

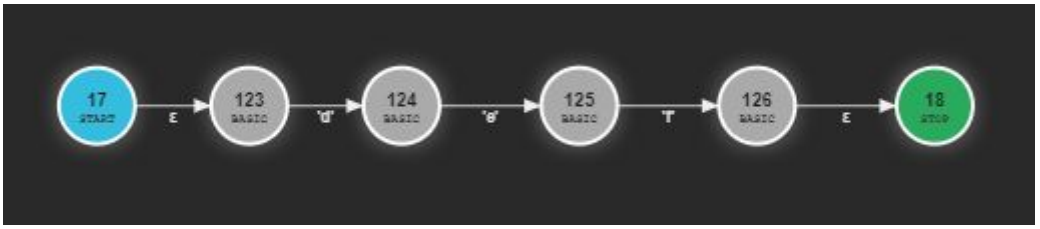


4.7 TOKEN NEW



4.8 TOKEN BREAK

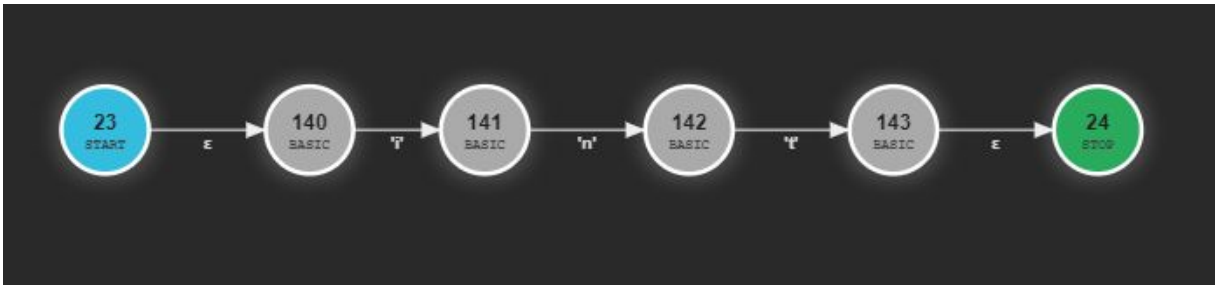
4.9 TOKEN DEF



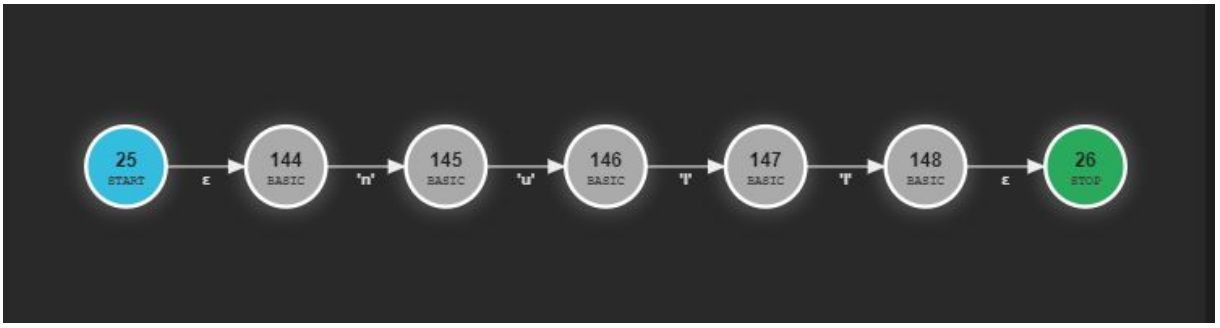
4.10 TOKEN TYPE_STRING

4.11 TOKEN TYPE_FLOAT

4.12 TOKEN TYPE_INT



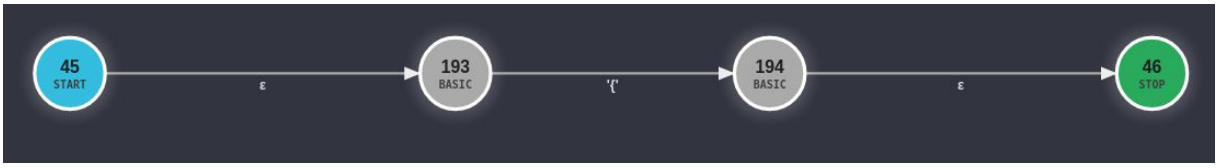
4.13 TOKEN NULL



4.14 TOKEN LPAREN

4.15 TOKEN RPAREN

4.16 TOKEN LBRACE



4.17 TOKEN RBRACE

4.18 TOKEN LBRACK

4.19 TOKEN RBRACK

4.20 TOKEN SEMI

4.21 TOKEN COMMA

4.22 TOKEN DOT

4.23 TOKEN COLON

4.24 TOKEN ASSIGN

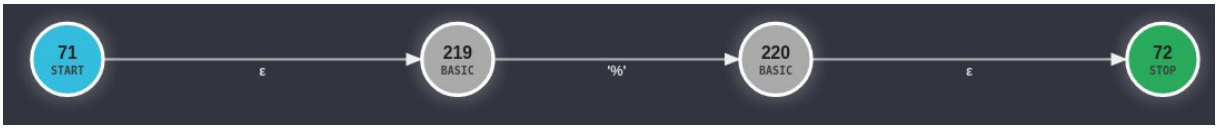
4.25 TOKEN ADD

4.26 TOKEN SUB

4.27 TOKEN MUL

4.28 TOKEN DIV

4.29 TOKEN MOD



4.30 TOKEN GT

4.31 TOKEN LT

4.32 TOKEN EQUAL

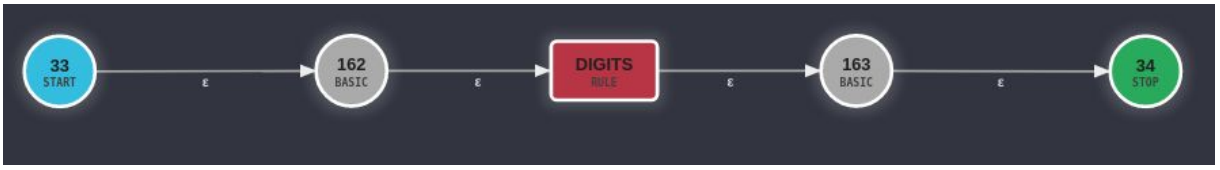
4.33 TOKEN LE

4.34 TOKEN GE

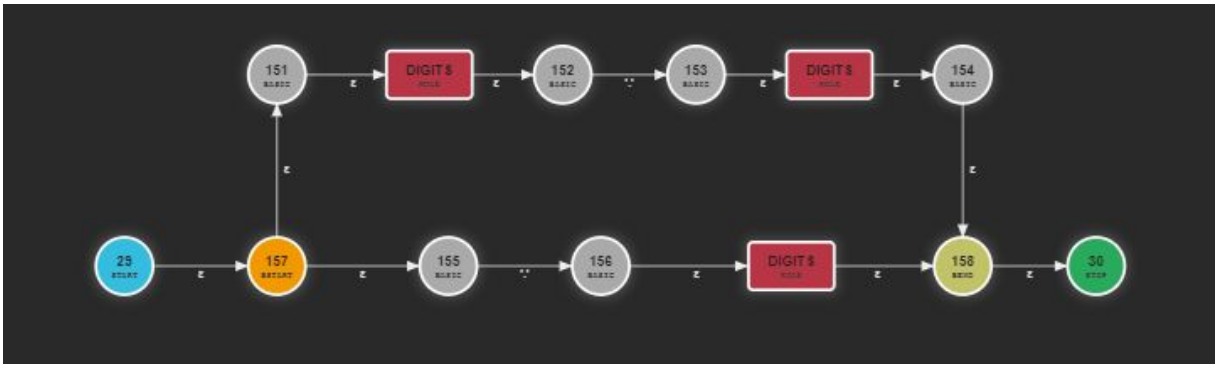
4.35 TOKEN NOT EQUAL

4.36 TOKENS NÃO TRIVIAIS

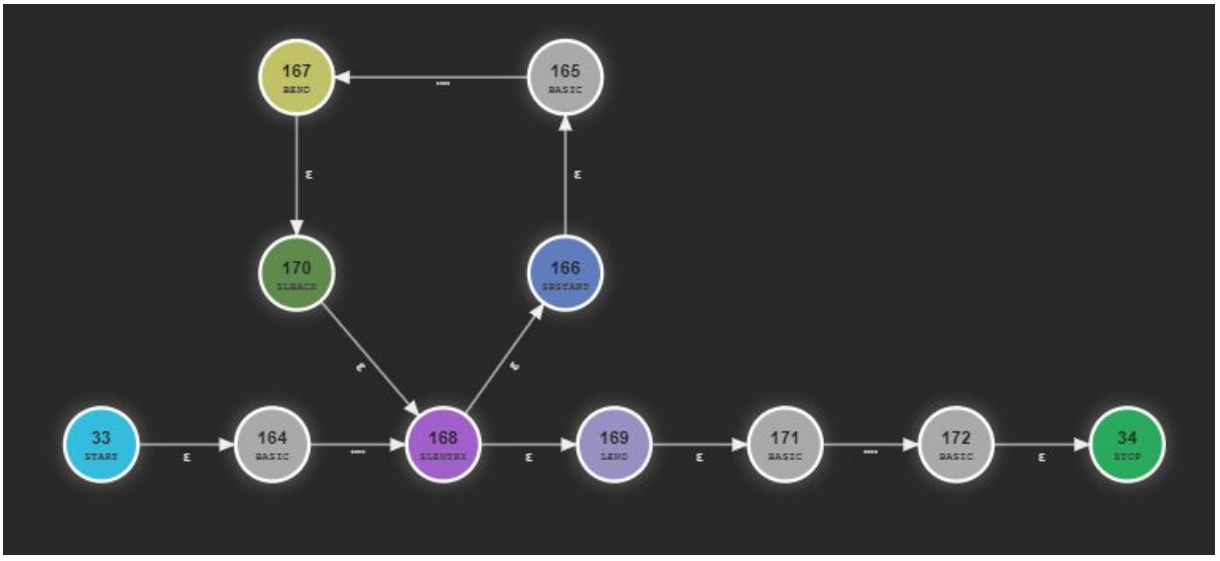
4.36.1 TOKEN INT



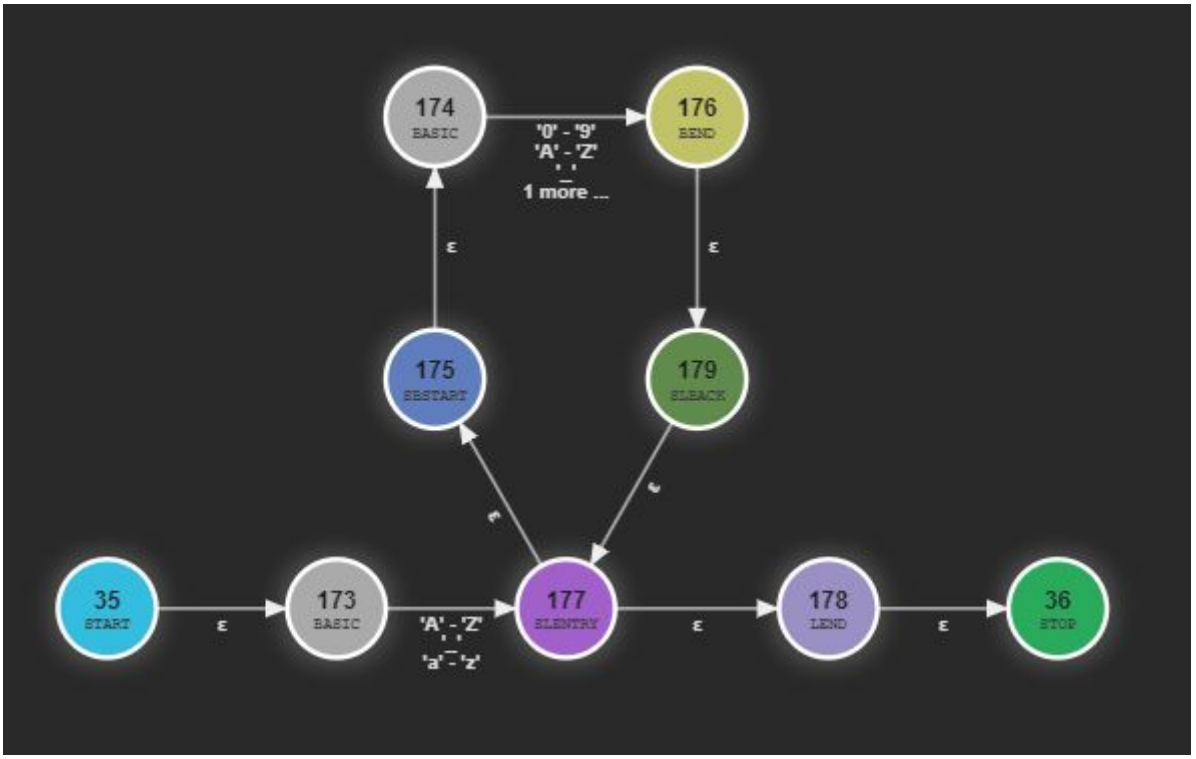
4.36.2 TOKEN FLOAT



4.36.3 TOKEN STRING



4.36.4 TOKEN IDENT



5 A FERRAMENTA

A ferramenta escolhida pelo grupo foi o ANTLR versão 4. O objetivo dessa escolha deu-se pelo fato de a ferramenta possuir integração com a linguagem escolhida para o desenvolvimento do trabalho, facilidade de uso e compreensão das funcionalidades e possibilidade de gerar os diagramas de transição automaticamente.

5.1 DESCRIÇÃO DA ENTRADA DA FERRAMENTA

Para utilizar a ferramenta, é preciso que um arquivo de extensão *.g4 seja criado. Nesse arquivo, é necessário que a gramática seja declarada na sintaxe da ferramenta como mostra a imagem.

```
program
  :
    ( statement
    | funclist
    )?
  ;
```

A partir do diretório principal do trabalho, seguindo o caminho /src/main/antlr4, pode-se encontrar o arquivo CC2020.g4, que foi utilizado para a definição da gramática da linguagem CC-2020-1.

5.2 DESCRIÇÃO DA SAÍDA DA FERRAMENTA

Ao executar o programa, a partir do arquivo CC2020.g4, a ferramenta gera como saída a classe CC2020Lexer.java**.** Tal classe é a responsável por implementar os métodos referentes ao analisador léxico. Sua responsabilidade é, a partir do código de entrada, fazer a identificação dos tokens recebidos. Com ela, foi possível a criação de um objeto do tipo CommonTokenStream, que foi utilizado para iterar pelos tokens identificados.

Com a possibilidade de iteração do Stream de tokens fornecido pelo objeto CommonTokenStream, foi possível a criação dos métodos *createLexemeSet, exportSymbolTable, writeTokens* e *exportTokens,* que podem ser encontrados na classe Utils.java e que são responsáveis pela criação da Tabela de Símbolos e de uma key para que cada Token possa identificar qual sua entrada na Tabela de Símbolos.

Para a criação da Tabela de símbolos foram usados dois métodos. O primeiro é chamado *createLexemeSet,* e possui a função de, para cada token do tipo IDENT, adicionar seu lexema à um HashSet chamado lexemeSet, que armazena todos os tokens que precisam de uma identificação na Tabela de Símbolos.

```java
public static HashSet<String> createLexemeSet(CommonTokenStream tokens) {
        HashSet<String> lexemeSet = new HashSet<String>();

        for (Token token : tokens.getTokens()) {
                int type = token.getType();
                String typeName = CC2020Lexer.VOCABULARY.getSymbolicName(type);

                if (typeName.equals("IDENT")) {
                        lexemeSet.add(token.getText());
                }
        }

        return lexemeSet;
}
```

Após a execução do *createLexemeSet,* o método *exportSymbolTable* é invocado. Sua responsabilidade é, a partir do lexemeSet criado pelo método anterior, dar um formato à Tabela de Símbolos, sendo ela composta por um valor único e um lexema. O valor único que a tabela possui é representado por uma key, que possui o objetivo de possibilitar que os tokens consigam encontrar sua entrada na Tabela de Símbolos, visto que eles também são compostos por uma key de mesmo valor. Além disso, o método também exporta a Tabela de Símbolos para um arquivo .txt.

```java
static public void exportSymbolTable(String filePath, HashSet<String> lexemeSet)
                throws IOException {
        String fileName = extractFileName(filePath);
        String newFileName = fileName + ".txt";
        String directoryPath = createDirectory("/Out/SymbolTable/", filePath);
        String newFilePath = directoryPath + newFileName;
        File newFile = new File(newFilePath);
        FileWriter writer = new FileWriter(newFile);

        writer.write("============> Symbol Table from the file " + fileName + ".ccc\n\n");

        for (String lexeme : lexemeSet) {
                String s = String.format("Entry: <key=%s, %s>\n", lexeme, "lexeme = " + lexeme);
                writer.write(s);
        }

        System.out
                        .println("A tabela de símbolos do arquivo " + fileName + ".ccc se encontra no arquivo: " + newFilePath);

        writer.close();
}
```

Já para a representação dos Tokens, foram utilizados os métodos *exportTokens e writeTokens.* O primeiro é responsável pela adição de uma key para cada token, que servirá para que eles possam identificar qual sua entrada na Tabela de Símbolos.

```
static public void exportTokens(String filePath, CommonTokenStream tokens,
                HashSet<String> lexemeSet) throws IOException {
        ArrayList<String> tokensList = new ArrayList<>();

        for (Token token : tokens.getTokens()) {
                int type = token.getType();
                String typeName = CC2020Lexer.VOCABULARY.getSymbolicName(type);
                String lexeme = token.getText();

                int line = token.getLine();
                int startIndex = token.getStartIndex();
                int stopIndex = token.getStopIndex();
                int tokenIndex = token.getTokenIndex();

                String stringOfToken = "";

                if (lexemeSet.contains(lexeme)) {
                        stringOfToken = String.format(
                                        "\nToken #%d: <key=%s, type=%s, lexeme=%s, line=%d, startIndex=%d, stopIndex=%d>\n", tokenIndex,
                                        lexeme, typeName, lexeme, line, startIndex, stopIndex);
                } else {
                        stringOfToken = String.format("\nToken #%d: <type=%s, lexeme=%s, line=%d, startIndex=%d, stopIndex=%d>\n",
                                        tokenIndex, typeName, lexeme, line, startIndex, stopIndex);
                }

                tokensList.add(stringOfToken);
        }

        writeTokens(filePath, tokensList);
}
```

Em seu final, é feita a invocação do segundo método, que faz a exportação dos tokens para um arquivo .txt.

```
static public void writeTokens(String filePath, ArrayList<String> tokens) throws IOException {
        String fileName = extractFileName(filePath);
        String newFileName = fileName + ".txt";
        String directoryPath = createDirectory("/Out/Tokens/", filePath);
        String newFilePath = directoryPath + newFileName;
        File newFile = new File(newFilePath);
        FileWriter writer = new FileWriter(newFile);

        writer.write("============> Tokens from the file " + fileName + ".ccc\n\n");

        for (String token : tokens) {
                writer.write(token);
        }

        System.out.println("Os tokens do arquivo " + fileName + ".ccc se encontram no arquivo: " + newFilePath);

        writer.close();
}
```