UNIVERSIDADE FEDERAL DE ALAGOAS
INSTITUTO DE COMPUTAÇÃO
COMP263 - Compiladores

LINGUAGEM DUMA: ANALISADOR LÉXICO

Eduarda Tatiane Caetano Chagas
Marcos Gleysson Silva do Nascimento

Maceió-AL
2017.1

Sumário

CAPÍTULO 1

ANALISADOR LÉXICO

1.1 Introdução

Para fazer o analisador léxico, criamos um projeto Java no Eclipse intitulado AnalyzerDUMA. Neste projeto, existem dois pacotes principais: o **src**, onde está o código-fonte e o **files**/**input** utilizado para adicionar os arquivos fonte dos três programas a serem utilizados como teste: *hello_world.duma, fibonacci.duma e shell_sort.duma*.

No pacote **src** criamos os pacotes **main** e **lexicalAnalyzer**. No pacote **main** está a classe *AnalyzerDUMA.java* que possui um método *main* para executar o analisador léxico. Por sua vez, no pacote **src** existem cinco classes: *LexicalAnalyzer.java, LexicalMap.java, LexicalPrinter.java, Token.java e TokenCategory.java* que juntas compõem o analisador léxico propriamente dito.

1.2 Código-fonte do projeto

1.2.1 Classe AnalyzerDUMA.java

```
package main;

import lexicalAnalyzer.LexicalAnalyzer;
import lexicalAnalyzer.LexicalPrinter;

public class AnalyzerDUMA {
        private static LexicalAnalyzer lexicalAnalyzer;

        private static String filePath = "files/input/fibonacci.duma";

        public AnalyzerDUMA() {

        }

        public static void main(String[] args) {

                lexicalAnalyzer = new LexicalAnalyzer(filePath);
                lexicalAnalyzer.readFile();
```

```
                LexicalPrinter.printTokens(lexicalAnalyzer);
        }
}
```

1.2.2 Classe LexicalAnalyzer.java

```
package lexicalAnalyzer;

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.List;

import lexicalAnalyzer.LexicalMap;
import lexicalAnalyzer.Token;
import lexicalAnalyzer.TokenCategory;

public class LexicalAnalyzer {

        private List<String> linesList;
        private int currentLine, currentColumn;
    private int tkBeginColumn = 0, tkBeginLine = 0;
        private String line;
        private String filePath;

        private final char LINE_BREAK = '\n';

        public LexicalAnalyzer(String filePath) {
                linesList = new ArrayList<>();
                this.filePath = filePath;
        }

        public void readFile() {
                BufferedReader br;
                try {
                        br = new BufferedReader(new FileReader(filePath));
                        String brLine = br.readLine();

                        while (brLine != null) {
                                linesList.add(brLine);
                                brLine = br.readLine();
                        }
                        br.close();
                } catch (Exception e) {
```

```
                e.printStackTrace();
            }
        }

        public Token nextToken() {

            Token token;

            char currentChar;
            String tkValue = "";

            tkBeginColumn = currentColumn;
            tkBeginLine = currentLine;

            currentChar = line.charAt(currentColumn);

            while(currentChar == ' ' || currentChar == '\t'){
                currentChar = nextChar();
                tkBeginColumn++;
            }

            if (Character.toString(currentChar).matches("\\d")) {
                tkValue += currentChar;
                currentChar = nextChar();
                while (Character.toString(currentChar).matches("\\d")){
                    tkValue += currentChar;
                    currentChar = nextChar();
                }
                if (currentChar == '.') {
                    tkValue += currentChar;
                    currentChar = nextChar();
                    while(Character.toString(currentChar).matches("\\d")){
                        tkValue += currentChar;
                        currentChar = nextChar();
                    }
                }

                if (currentChar != ' ') {
                    while (!LexicalMap.symbolList.contains(currentChar)){
                        tkValue += currentChar;

                        // Vai para o proximo
                        currentChar = nextChar();
```

```
                        if (currentChar == LINE_BREAK) {
                                break;
                        }
                }
        }
} else {

        while(!LexicalMap.symbolList.contains(currentChar)){
                tkValue += currentChar;

                currentChar = nextChar();
                if (currentChar == LINE_BREAK) {
                        break;
                }
        }
}

if (tkValue == "") {

        switch (currentChar) {

        case '"':

                tkValue += currentChar;
                currentChar = nextChar();

                if (currentChar == '"') {
                        tkValue += currentChar;
                        currentColumn++;
                        break;
                }

                while (currentChar != LINE_BREAK) {
                        tkValue += currentChar;
                        currentChar = nextChar();

                        if (currentChar == '"') {
                                tkValue += currentChar;
                                currentColumn++;
                                break;
                        }
                }
                break;
```

```
case '/':
        tkValue += currentChar;
        currentChar = nextChar();

        if (currentChar == '#') {
                tkValue += currentChar;
                currentLine++;
                currentColumn = 0;
        }
        break;

case '#':
        tkValue += currentChar;
        currentChar = nextChar();
        if (currentChar == '/') {
                tkValue += currentChar;
                currentChar = nextChar();
        }
        break;

case '\'':

        tkValue += currentChar;

        currentChar = nextChar();
        if (currentChar != LINE_BREAK) {
                tkValue += currentChar;
        }
        currentChar = nextChar();
        if (currentChar == '\'') {
                tkValue += currentChar;
                currentColumn++;
        }
        break;

case '<':
case '>':
case '!':
case '=':

        tkValue += currentChar;
        currentChar = nextChar();
```

```
                        if (currentChar == '=') {
                                tkValue += currentChar;
                                currentColumn++;
                        }
                        break;

                case '+':
                        tkValue += currentChar;
                        currentChar = nextChar();

                        if (currentChar == '+') {
                                tkValue += currentChar;
                                currentChar = nextChar();
                        }

                        break;

                default:
                        tkValue += currentChar;
                        currentColumn++;
                        break;
                }
        }

        tkValue = tkValue.trim();

        token = new Token();

        token.setValue(tkValue);
        token.setLine(tkBeginLine);
        token.setColumn(tkBeginColumn);
        token.setCategory(analyzeCategory(tkValue));

        if (token.getCategory().equals(TokenCategory.COMMENT)) {
                if (hasMoreTokens()) {
                        return nextToken();
                }
        }
        return token;

}

public boolean hasMoreTokens() {
```

```
            if (!linesList.isEmpty()) {
                    if (currentLine < linesList.size()) {

                            line = linesList.get(currentLine);
                            line = line.replace('\t', ' ');

                            if (line.substring(currentColumn).matches("\\s*")) {
                                    currentLine++;
                                    currentColumn = 0;
                                    while (currentLine < linesList.size()) {
                                            line = linesList.get(currentLine);
                                            if (line.matches("\\s*")) {
                                                    currentLine++;
                                            } else {
                                                    return true;
                                            }
                                    }
                            } else if (currentColumn < line.length()) {
                                    return true;
                            } else {
                                    currentLine++;
                                    currentColumn = 0;
                                    while (currentLine < linesList.size()) {
                                            line = linesList.get(currentLine);
                                            if (line.matches("\\s*")) {
                                                    currentLine++;
                                            } else {
                                                    return true;
                                            }
                                    }
                            }
                    }
            }

            return false;

    }

    private TokenCategory analyzeCategory(String tkValue) {

            if (isOpNegUnary(tkValue)) {
                    return TokenCategory.OPARITUN;
```

```
            } else
                    if (LexicalMap.lexemMap.containsKey(tkValue)) {
                    return LexicalMap.lexemMap.get(tkValue);

            } else if (isCchar(tkValue)) {
                    return TokenCategory.CTESERMO;

            } else if (isChar(tkValue)) {
                    return TokenCategory.CTELIT;

            } else if (isConstInt(tkValue)) {
                    return TokenCategory.CTENUMINT;

            } else if (isConstDec(tkValue)) {
                    return TokenCategory.CTENUMREAL;

            } else if (isIdentifier(tkValue)) {
                    return TokenCategory.ID;
            }

            return TokenCategory.UNKNOWN;
    }

    private Character nextChar() {

            currentColumn++;

            if (currentColumn < line.length()) {
                    return line.charAt(currentColumn);
            } else {
                    return LINE_BREAK;
            }

    }

    private boolean isOpNegUnary(String tkValue) {

            if (tkValue.equals("-")) {

                    Character previousChar = previousNotBlankChar();
                    if ((previousChar != null) &&
        Character.toString(previousChar).matches("[_a-zA-Z0-9]")) {
```

```
                        return false;
                } else {
                        return true;
                }
        }
        return false;
}

private Character previousNotBlankChar() {

        int previousColumn = tkBeginColumn - 1;
        char previousChar;

        while (previousColumn >= 0) {
                previousChar = line.charAt(previousColumn);
                if (previousChar != ' ' && previousChar != '\t') {
                        return previousChar;
                }
                previousColumn--;
        }
        return null;
}

private boolean isConstDec(String tkValue) {
        if (tkValue.matches("(\\d)+\\.(\\d)+")) {
                return true;
        } else if (tkValue.matches("(\\d)+\\.")) {
                printError("constante decimal em formato errado.", tkValue);
        }
        return false;
}

private boolean isConstInt(String tkValue) {
        if (tkValue.matches("(\\d)+")) {
                return true;
        }
        return false;
}

private boolean isString(String tkValue) {
        if (tkValue.startsWith("\"") && tkValue.endsWith("\"")) {
                return true;
        } else if (tkValue.startsWith("\"")) {
```

```
                printError("String nao fechada corretamente com '\"'.",
    tkValue);
            }
            return false;
    }

    private boolean isChar(String tkValue) {
            if (tkValue.matches("'(.?)'")) {
                    return true;
            } else if (tkValue.startsWith("'")) {
                    printError("caracter nao fechado corretamente com '.",
    tkValue);
            }
            return false;
    }

    private boolean isIdentifier(String tkValue) {

            if (tkValue.matches("[_a-zA-Z][_a-zA-Z0-9]*")) {
                    if (tkValue.length() < 16) {
                            return true;
                    } else {
                            printError("identificador muito longo.", tkValue);
                    }

            } else if (tkValue.matches("[^_a-zA-Z=\"'].*")) {
                    printError("identificador nao iniciado com letra ou '_'.",
    tkValue);

            } else if (tkValue.matches("[_a-zA-Z].*")) {
                    printError("identificador contem caracter invalido.",
    tkValue);

            }
            return false;
    }

    private void printError(String string, String token) {
            System.err.println("Erro na <linha, coluna> " + "= <" + currentLine
                            + "," + currentColumn + ">. " + "'" + token + "'"
            + " " + string);
            System.exit(1);
    }
```

```
}
```

1.2.3 Classe LexicalMap.java

```
package lexicalAnalyzer;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class LexicalMap {

        public static Map<String, TokenCategory> lexemMap = new HashMap<>();
        public static Map<String, TokenCategory> delimitadorMap = new HashMap<>();
        public static List<Character> symbolList = new ArrayList<>();

        static {


                lexemMap.put("+", TokenCategory.OPARITAD);
                lexemMap.put("-", TokenCategory.OPARITAD);
                lexemMap.put("*", TokenCategory.OPARITMUL);
                lexemMap.put("/", TokenCategory.OPARITMUL);

                lexemMap.put("<", TokenCategory.OPREL1);
                lexemMap.put(">", TokenCategory.OPREL1);
                lexemMap.put("<=", TokenCategory.OPREL1);
                lexemMap.put(">=", TokenCategory.OPREL1);
                lexemMap.put("==", TokenCategory.OPREL2);
                lexemMap.put("!=", TokenCategory.OPREL2);

                lexemMap.put("!", TokenCategory.OPLOGNEG);
                lexemMap.put("&&", TokenCategory.OPLOGAND);
                lexemMap.put("||", TokenCategory.OPLOGOR);

                lexemMap.put(".", TokenCategory.OPCON);

                lexemMap.put("(", TokenCategory.PARAMBEGIN);
                lexemMap.put(")", TokenCategory.PARAMEND);

                lexemMap.put("{", TokenCategory.ESCBEGIN);
                lexemMap.put("}", TokenCategory.ESCEND);
```

```
lexemMap.put("[", TokenCategory.VETBEGIN);
lexemMap.put("]", TokenCategory.VETEND);

lexemMap.put("#", TokenCategory.COMMENT);

lexemMap.put(";", TokenCategory.TERMCMD);

lexemMap.put(",", TokenCategory.SEPVIRG);

lexemMap.put("litterae", TokenCategory.TDLIT);
lexemMap.put("sermo", TokenCategory.TDSERMO);
lexemMap.put("inanis", TokenCategory.TDINANIS);
lexemMap.put("integer", TokenCategory.TDINT);
lexemMap.put("realem", TokenCategory.TDREAL);
lexemMap.put("boolean", TokenCategory.TDBOOL);

lexemMap.put("si", TokenCategory.SELSI);
lexemMap.put("aliud", TokenCategory.SELALIUD);
lexemMap.put("sialiud", TokenCategory.SELSIALIUD);

lexemMap.put("quia", TokenCategory.REPQUIA);
lexemMap.put("dum", TokenCategory.REPDUM);
lexemMap.put("in", TokenCategory.REPIN);
lexemMap.put("facite", TokenCategory.REPFACITE);
lexemMap.put("spatium", TokenCategory.REPSPATIUM);

lexemMap.put("true", TokenCategory.CTEBOOL);
lexemMap.put("false", TokenCategory.CTEBOOL);

lexemMap.put("duma", TokenCategory.PRDUMA);
lexemMap.put("const", TokenCategory.PRCONST);
lexemMap.put("fun", TokenCategory.PRFUN);
lexemMap.put("var", TokenCategory.PRVAR);
lexemMap.put("initium", TokenCategory.PRINITIUM);
lexemMap.put("reditus", TokenCategory.PRREDITUS);
lexemMap.put("matrix", TokenCategory.PRMATRIX);
lexemMap.put("scribo", TokenCategory.PRSCRIBO);
lexemMap.put("scriboln", TokenCategory.PRSCRIBOLN);
lexemMap.put("lectio", TokenCategory.PRLECTIO);


symbolList.add(' ');
symbolList.add(',');
```

```
                symbolList.add(';');
                symbolList.add('+');
                symbolList.add('-');
                symbolList.add('*');
                symbolList.add('\\');
                symbolList.add('/');
                symbolList.add('#');
                symbolList.add('$');
                symbolList.add('<');
                symbolList.add('>');
                symbolList.add('=');
                symbolList.add('~');
                symbolList.add('(');
                symbolList.add(')');
                symbolList.add('[');
                symbolList.add(']');
                symbolList.add('{');
                symbolList.add('}');
                symbolList.add('\'');
                symbolList.add('"');

        }

}
```

1.2.4 Classe LexicalPrinter.java

```
package lexicalAnalyzer;

public class LexicalPrinter {


        public static void printTokens(LexicalAnalyzer lexicalAnalyzer) {
                Token token;
                while (lexicalAnalyzer.hasMoreTokens()) {
                        token = lexicalAnalyzer.nextToken();
                        System.out.println(token.toString());
                }
                System.out.println();
                System.out.println();
                lexicalAnalyzer.readFile();
        }

}
```

1.2.5 Classe Token.java

```
package lexicalAnalyzer;

public class Token {

	private String value;
	private TokenCategory category;
	private int line;
	private int column;

	@Override
	public String toString() {
		return "<" + line + "," + column + "> " + category + " = '" + value +
	}

	public String getValue() {
		return value;
	}

	public void setValue(String value) {
		this.value = value;
	}

	public TokenCategory getCategory() {
		return category;
	}

	public void setCategory(TokenCategory category) {
		this.category = category;
	}

	public int getLine() {
		return line;
	}

	public void setLine(int line) {
		this.line = line;
	}

	public int getColumn() {
		return column;
	}
```

```
        public void setColumn(int column) {
                this.column = column;
        }

}
```

1.2.6 Enum TokenCategory.java

```
package lexicalAnalyzer;

public enum TokenCategory {

        PRDUMA(1), PRINITIUM(2), PRCONST(3), PRVAR(4), PRFUN(5),
        PRSCRIBO(6), PRSCRIBOLN(7), PRLECTIO(8), PRREDITUS(9),
    ID(10), TDINANIS(11), TDINT(12), TDREAL(13), TDLIT(14), TDBOOL(15),
    TDSERMO(16), PRMATRIX(17), ESCBEGIN(18), ESCEND(19), PARAMBEGIN(20),
    PARAMEND(21), COMMENT(22), SEPVIRG(23), TERMCMD(24), CTENUMINT(25),
    CTENUMREAL(26), CTELIT(27), CTESERMO(28), CTEBOOL(29), SELSI(30),
    SELALIUD(31), SELSIALIUD(32), REPQUIA(33), REPDUM(34), REPFACITE(35),
    REPSPATIUM(36), REPIN(37), OPARITAD(38), OPARITMUL(39), OPLOGAND(40),
    OPLOGOR(41), OPLOGNEG(42), OPARITUN(43), OPREL1(44), OPREL2(45),
    OPCON(46), VETBEGIN(47), VETEND(48), UNKNOWN(49);


        private int value;

        private TokenCategory(int value) {
                this.value = value;
        }

        public int getCategoryValue() {
                return value;
        }

}
```

1.3 Testes realizados

1.3.1 Hello Word: hello_world.duma

```
duma hello_world
inanis initium()
{
```

```
    scriboln("Alo mundo");

}
```

Saída do teste do programa Hello Word

```
<1,0> PRDUMA = 'duma'
<1,5> ID = 'hello_word'
<2,0> TDINANIS = 'inanis'
<2,7> PRINITIUM = 'initium'
<2,14> PARAMBEGIN = '('
<2,15> PARAMEND = ')'
<3,0> ESCBEGIN = '{'
<4,4> PRSCRIBOLN = 'scriboln'
<4,12> PARAMBEGIN = '('
<4,13> CTESERMO = '"Alo mundo"'
<4,24> PARAMEND = ')'
<4,25> TERMCMD = ';'
<6,0> ESCEND = '}'
```

1.3.2 Fibonacci - fibonacci.duma

```
duma fibonacci
var{
        integer a, b, auxiliar, i, n;
}
inanis initium()
{
  a = 0;
  b = 1;
  i = 0;

  scriboln("Digite um numero: ");
  lectio(n);
  scriboln("Serie de Fibonacci:\n");
  scriboln(b , "\n");

  dum(i<n)
  {
    auxiliar = a + b;
    a = b;
    b = auxiliar;
    i = i + 1;
```

```
    scriboln(auxiliar , "\n");
  }
}
```

Saída do teste para o programa Fibonacci

```
<1,0> PRDUMA = 'duma'
<1,5> ID = 'fibonacci'
<2,0> PRVAR = 'var'
<2,3> ESCBEGIN = '{'
<3,2> TDINT = 'integer'
<3,10> ID = 'a'
<3,11> SEPVIRG = ','
<3,13> ID = 'b'
<3,14> SEPVIRG = ','
<3,16> ID = 'auxiliar'
<3,24> SEPVIRG = ','
<3,26> ID = 'i'
<3,27> SEPVIRG = ','
<3,29> ID = 'n'
<3,30> TERMCMD = ';'
<4,0> ESCEND = '}'
<5,0> TDINANIS = 'inanis'
<5,7> PRINITIUM = 'initium'
<5,14> PARAMBEGIN = '('
<5,15> PARAMEND = ')'
<6,0> ESCBEGIN = '{'
<7,2> ID = 'a'
<7,4> UNKNOWN = '='
<7,6> CTENUMINT = '0'
<7,7> TERMCMD = ';'
<8,2> ID = 'b'
<8,4> UNKNOWN = '='
<8,6> CTENUMINT = '1'
<8,7> TERMCMD = ';'
<9,2> ID = 'i'
<9,4> UNKNOWN = '='
<9,6> CTENUMINT = '0'
<9,7> TERMCMD = ';'
<11,2> PRSCRIBOLN = 'scriboln'
<11,10> PARAMBEGIN = '('
<11,11> CTESERMO = '"Digite um numero: "'
<11,31> PARAMEND = ')'
<11,32> TERMCMD = ';'
```

```
<12,2> PRLECTIO = 'lectio'
<12,8> PARAMBEGIN = '('
<12,9> ID = 'n'
<12,10> PARAMEND = ')'
<12,11> TERMCMD = ';'
<13,2> PRSCRIBOLN = 'scriboln'
<13,10> PARAMBEGIN = '('
<13,11> CTESERMO = '"Serie de Fibonacci:\n"'
<13,34> PARAMEND = ')'
<13,35> TERMCMD = ';'
<14,2> PRSCRIBOLN = 'scriboln'
<14,10> PARAMBEGIN = '('
<14,11> ID = 'b'
<14,13> SEPVIRG = ','
<14,15> CTESERMO = '"\n"'
<14,19> PARAMEND = ')'
<14,20> TERMCMD = ';'
<16,2> REPDUM = 'dum'
<16,5> PARAMBEGIN = '('
<16,6> ID = 'i'
<16,7> OPREL1 = '<'
<16,8> ID = 'n'
<16,9> PARAMEND = ')'
<17,2> ESCBEGIN = '{'
<18,4> ID = 'auxiliar'
<18,13> UNKNOWN = '='
<18,15> ID = 'a'
<18,17> OPARITAD = '+'
<18,19> ID = 'b'
<18,20> TERMCMD = ';'
<19,4> ID = 'a'
<19,6> UNKNOWN = '='
<19,8> ID = 'b'
<19,9> TERMCMD = ';'
<20,4> ID = 'b'
<20,6> UNKNOWN = '='
<20,8> ID = 'auxiliar'
<20,16> TERMCMD = ';'
<21,4> ID = 'i'
<21,6> UNKNOWN = '='
<21,8> ID = 'i'
<21,10> OPARITAD = '+'
<21,12> CTENUMINT = '1'
```

```
<21,13> TERMCMD = ';'
<23,4> PRSCRIBOLN = 'scriboln'
<23,12> PARAMBEGIN = '('
<23,13> ID = 'auxiliar'
<23,22> SEPVIRG = ','
<23,24> CTESERMO = '"\n"'
<23,28> PARAMEND = ')'
<23,29> TERMCMD = ';'
<24,2> ESCEND = '}'
<25,0> ESCEND = '}'
```

1.3.3 Shell sort: shell_sort.duma

```
duma shell_sort

var{
        integer n, a, i, j, value, gap;
}

inanis initium(){
    scriboln("Digite a quantidade de numeros a serem ordenados: ");
    lectio(n);
    matrix integer vetor[n];
    scriboln("Digite os numeros:");
        quia a in spatium(1,n,1){
        lectio(vetor[a]);
    }

    gap = 1;
        facite{
                gap = 3*gap+1;
        }dum(gap < n);

        facite{
                gap = gap / 3;
                quia a in spatium(gap,n,1){
                        value = a[i];
                        j = i - gap;

                        dum(j >= 0 && value < a[j]) {
                        a[j + gap] = a[j];
                        j = j - gap;
                        }
                        a[j + gap] = value;
```

```
                }
        }dum(gap > 1);

    scriboln("Numeros ordenados:");
        quia a in spatium(1,n,1){
                scriboln(vetor[a]);
        }
}
```

Saída do teste do programa Shell Sort

```
<1,0> PRDUMA = 'duma'
<1,5> ID = 'shell_sort'
<3,0> PRVAR = 'var'
<3,3> ESCBEGIN = '{'
<4,1> TDINT = 'integer'
<4,9> ID = 'n'
<4,10> SEPVIRG = ','
<4,12> ID = 'a'
<4,13> SEPVIRG = ','
<4,15> ID = 'i'
<4,16> SEPVIRG = ','
<4,18> ID = 'j'
<4,19> SEPVIRG = ','
<4,21> ID = 'value'
<4,26> SEPVIRG = ','
<4,28> ID = 'gap'
<4,31> TERMCMD = ';'
<5,0> ESCEND = '}'
<7,0> TDINANIS = 'inanis'
<7,7> PRINITIUM = 'initium'
<7,14> PARAMBEGIN = '('
<7,15> PARAMEND = ')'
<7,16> ESCBEGIN = '{'
<8,4> PRSCRIBOLN = 'scriboln'
<8,12> PARAMBEGIN = '('
<8,13> CTESERMO = '"Digite a quantidade de numeros a serem ordenados: "'
<8,65> PARAMEND = ')'
<8,66> TERMCMD = ';'
<9,4> PRLECTIO = 'lectio'
<9,10> PARAMBEGIN = '('
<9,11> ID = 'n'
<9,12> PARAMEND = ')'
<9,13> TERMCMD = ';'
```

```
<10,4> PRMATRIX = 'matrix'
<10,11> TDINT = 'integer'
<10,19> ID = 'vetor'
<10,24> VETBEGIN = '['
<10,25> ID = 'n'
<10,26> VETEND = ']'
<10,27> TERMCMD = ';'
<11,4> PRSCRIBOLN = 'scriboln'
<11,12> PARAMBEGIN = '('
<11,13> CTESERMO = '"Digite os numeros:"'
<11,33> PARAMEND = ')'
<11,34> TERMCMD = ';'
<12,1> REPQUIA = 'quia'
<12,6> ID = 'a'
<12,8> REPIN = 'in'
<12,11> REPSPATIUM = 'spatium'
<12,18> PARAMBEGIN = '('
<12,19> CTENUMINT = '1'
<12,20> SEPVIRG = ','
<12,21> ID = 'n'
<12,22> SEPVIRG = ','
<12,23> CTENUMINT = '1'
<12,24> PARAMEND = ')'
<12,25> ESCBEGIN = '{'
<13,5> PRLECTIO = 'lectio'
<13,11> PARAMBEGIN = '('
<13,12> ID = 'vetor'
<13,17> VETBEGIN = '['
<13,18> ID = 'a'
<13,19> VETEND = ']'
<13,20> PARAMEND = ')'
<13,21> TERMCMD = ';'
<14,4> ESCEND = '}'
<16,4> ID = 'gap'
<16,8> UNKNOWN = '='
<16,10> CTENUMINT = '1'
<16,11> TERMCMD = ';'
<17,1> REPFACITE = 'facite'
<17,7> ESCBEGIN = '{'
<18,2> ID = 'gap'
<18,6> UNKNOWN = '='
<18,8> CTENUMINT = '3'
<18,9> OPARITMUL = '*'
```

```
<18,10> ID = 'gap'
<18,13> OPARITAD = '+'
<18,14> CTENUMINT = '1'
<18,15> TERMCMD = ';'
<19,1> ESCEND = '}'
<19,2> REPDUM = 'dum'
<19,5> PARAMBEGIN = '('
<19,6> ID = 'gap'
<19,10> OPREL1 = '<'
<19,12> ID = 'n'
<19,13> PARAMEND = ')'
<19,14> TERMCMD = ';'
<21,1> REPFACITE = 'facite'
<21,7> ESCBEGIN = '{'
<22,2> ID = 'gap'
<22,6> UNKNOWN = '='
<22,8> ID = 'gap'
<22,12> OPARITMUL = '/'
<22,14> CTENUMINT = '3'
<22,15> TERMCMD = ';'
<23,2> REPQUIA = 'quia'
<23,7> ID = 'a'
<23,9> REPIN = 'in'
<23,12> REPSPATIUM = 'spatium'
<23,19> PARAMBEGIN = '('
<23,20> ID = 'gap'
<23,23> SEPVIRG = ','
<23,24> ID = 'n'
<23,25> SEPVIRG = ','
<23,26> CTENUMINT = '1'
<23,27> PARAMEND = ')'
<23,28> ESCBEGIN = '{'
<24,6> ID = 'value'
<24,12> UNKNOWN = '='
<24,14> ID = 'a'
<24,15> VETBEGIN = '['
<24,16> ID = 'i'
<24,17> VETEND = ']'
<24,18> TERMCMD = ';'
<25,6> ID = 'j'
<25,8> UNKNOWN = '='
<25,10> ID = 'i'
<25,12> OPARITAD = '-'
```

```
<25,14> ID = 'gap'
<25,17> TERMCMD = ';'
<27,6> REPDUM = 'dum'
<27,9> PARAMBEGIN = '('
<27,10> ID = 'j'
<27,12> OPREL1 = '>='
<27,15> CTENUMINT = '0'
<27,17> OPLOGAND = '&&'
<27,20> ID = 'value'
<27,26> OPREL1 = '<'
<27,28> ID = 'a'
<27,29> VETBEGIN = '['
<27,30> ID = 'j'
<27,31> VETEND = ']'
<27,32> PARAMEND = ')'
<27,34> ESCBEGIN = '{'
<28,7> ID = 'a'
<28,8> VETBEGIN = '['
<28,9> ID = 'j'
<28,11> OPARITAD = '+'
<28,13> ID = 'gap'
<28,16> VETEND = ']'
<28,18> UNKNOWN = '='
<28,20> ID = 'a'
<28,21> VETBEGIN = '['
<28,22> ID = 'j'
<28,23> VETEND = ']'
<28,24> TERMCMD = ';'
<29,7> ID = 'j'
<29,9> UNKNOWN = '='
<29,11> ID = 'j'
<29,13> OPARITAD = '-'
<29,15> ID = 'gap'
<29,18> TERMCMD = ';'
<30,6> ESCEND = '}'
<31,6> ID = 'a'
<31,7> VETBEGIN = '['
<31,8> ID = 'j'
<31,10> OPARITAD = '+'
<31,12> ID = 'gap'
<31,15> VETEND = ']'
<31,17> UNKNOWN = '='
<31,19> ID = 'value'
```

```
<31,24> TERMCMD = ';'
<32,4> ESCEND = '}'
<33,1> ESCEND = '}'
<33,2> REPDUM = 'dum'
<33,5> PARAMBEGIN = '('
<33,6> ID = 'gap'
<33,10> OPREL1 = '>'
<33,12> CTENUMINT = '1'
<33,13> PARAMEND = ')'
<33,14> TERMCMD = ';'
<35,4> PRSCRIBOLN = 'scriboln'
<35,12> PARAMBEGIN = '('
<35,13> CTESERMO = '"Numeros ordenados:"'
<35,33> PARAMEND = ')'
<35,34> TERMCMD = ';'
<36,1> REPQUIA = 'quia'
<36,6> ID = 'a'
<36,8> REPIN = 'in'
<36,11> REPSPATIUM = 'spatium'
<36,18> PARAMBEGIN = '('
<36,19> CTENUMINT = '1'
<36,20> SEPVIRG = ','
<36,21> ID = 'n'
<36,22> SEPVIRG = ','
<36,23> CTENUMINT = '1'
<36,24> PARAMEND = ')'
<36,25> ESCBEGIN = '{'
<37,5> PRSCRIBOLN = 'scriboln'
<37,13> PARAMBEGIN = '('
<37,14> ID = 'vetor'
<37,19> VETBEGIN = '['
<37,20> ID = 'a'
<37,21> VETEND = ']'
<37,22> PARAMEND = ')'
<37,23> TERMCMD = ';'
<38,1> ESCEND = '}'
<39,0> ESCEND = '}'
```