Raul Aguilar

Professor Paulding

CS 220 2148

November 27, 2020

Lab 10: JackTokenizer

SquareDanceT.xml

<tokens>

<keyword> class </keyword>

<identifier> SquareGame </identifier>

<symbol> { </symbol>

<keyword> field </keyword>

<identifier> Square </identifier>

<identifier> square </identifier>

<symbol> ; </symbol>

<keyword> field </keyword>

<keyword> int </keyword>

<identifier> direction </identifier>

<symbol> ; </symbol>

<keyword> constructor </keyword>

<identifier> SquareGame </identifier>

<identifier> new </identifier>

<symbol> ( </symbol>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> let </keyword>

<identifier> square </identifier>

<symbol> = </symbol>

<identifier> Square </identifier>

<symbol> . </symbol>

<identifier> new </identifier>

<symbol> ( </symbol>

<integerConstant> 0 </integerConstant>

<symbol> , </symbol>

<integerConstant> 0 </integerConstant>

<symbol> , </symbol>

<integerConstant> 30 </integerConstant>

<symbol> ) </symbol>

<symbol> ; </symbol>

<keyword> let </keyword>

<identifier> direction </identifier>

<symbol> = </symbol>

<integerConstant> 0 </integerConstant>

<symbol> ; </symbol>

<keyword> return </keyword>

<keyword> this </keyword>

<symbol> ; </symbol>

<symbol> } </symbol>

<keyword> method </keyword>

<keyword> void </keyword>

<identifier> dispose </identifier>

<symbol> ( </symbol>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> do </keyword>

<identifier> square </identifier>

<symbol> . </symbol>

<identifier> dispose </identifier>

<symbol> ( </symbol>

<symbol> ) </symbol>

<symbol> ; </symbol>

<keyword> do </keyword>

<identifier> Memory </identifier>

<symbol> . </symbol>

<identifier> deAlloc </identifier>

<symbol> ( </symbol>

<keyword> this </keyword>

<symbol> ) </symbol>

<symbol> ; </symbol>

<keyword> return </keyword>

<symbol> ; </symbol>

<symbol> } </symbol>

<keyword> method </keyword>

<keyword> void </keyword>

<identifier> moveSquare </identifier>

<symbol> ( </symbol>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> if </keyword>

<symbol> ( </symbol>

<identifier> direction </identifier>

<symbol> = </symbol>

<integerConstant> 1 </integerConstant>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> do </keyword>

<identifier> square </identifier>

<symbol> . </symbol>

<identifier> moveUp </identifier>

<symbol> ( </symbol>

<symbol> ) </symbol>

<symbol> ; </symbol>

<symbol> } </symbol>

<keyword> if </keyword>

<symbol> ( </symbol>

<identifier> direction </identifier>

<symbol> = </symbol>

<integerConstant> 2 </integerConstant>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> do </keyword>

<identifier> square </identifier>

<symbol> . </symbol>

<identifier> moveDown </identifier>

<symbol> ( </symbol>

<symbol> ) </symbol>

<symbol> ; </symbol>

<symbol> } </symbol>

<keyword> if </keyword>

<symbol> ( </symbol>

<identifier> direction </identifier>

<symbol> = </symbol>

<integerConstant> 3 </integerConstant>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> do </keyword>

<identifier> square </identifier>

<symbol> . </symbol>

<identifier> moveLeft </identifier>

<symbol> ( </symbol>

<symbol> ) </symbol>

<symbol> ; </symbol>

<symbol> } </symbol>

<keyword> if </keyword>

<symbol> ( </symbol>

<identifier> direction </identifier>

<symbol> = </symbol>

<integerConstant> 4 </integerConstant>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> do </keyword>

<identifier> square </identifier>

<symbol> . </symbol>

<identifier> moveRight </identifier>

<symbol> ( </symbol>

<symbol> ) </symbol>

<symbol> ; </symbol>

<symbol> } </symbol>

<keyword> do </keyword>

<identifier> Sys </identifier>

<symbol> . </symbol>

<identifier> wait </identifier>

<symbol> ( </symbol>

<integerConstant> 5 </integerConstant>

<symbol> ) </symbol>

<symbol> ; </symbol>

<keyword> return </keyword>

<symbol> ; </symbol>

<symbol> } </symbol>

<keyword> method </keyword>

<keyword> void </keyword>

<identifier> run </identifier>

<symbol> ( </symbol>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> var </keyword>

<keyword> char </keyword>

<identifier> key </identifier>

<symbol> ; </symbol>

<keyword> var </keyword>

<keyword> boolean </keyword>

<identifier> exit </identifier>

<symbol> ; </symbol>

<keyword> let </keyword>

<identifier> exit </identifier>

<symbol> = </symbol>

<keyword> false </keyword>

<symbol> ; </symbol>

<keyword> while </keyword>

<symbol> ( </symbol>

<symbol> ~ </symbol>

<identifier> exit </identifier>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> while </keyword>

<symbol> ( </symbol>

<identifier> key </identifier>

<symbol> = </symbol>

<integerConstant> 0 </integerConstant>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> let </keyword>

<identifier> key </identifier>

<symbol> = </symbol>

<identifier> Keyboard </identifier>

<symbol> . </symbol>

<identifier> keyPressed </identifier>

<symbol> ( </symbol>

<symbol> ) </symbol>

<symbol> ; </symbol>

<keyword> do </keyword>

<identifier> moveSquare </identifier>

<symbol> ( </symbol>

<symbol> ) </symbol>

<symbol> ; </symbol>

<symbol> } </symbol>

<keyword> if </keyword>

<symbol> ( </symbol>

<identifier> key </identifier>

<symbol> = </symbol>

<integerConstant> 81 </integerConstant>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> let </keyword>

<identifier> exit </identifier>

<symbol> = </symbol>

<keyword> true </keyword>

<symbol> ; </symbol>

<symbol> } </symbol>

<keyword> if </keyword>

<symbol> ( </symbol>

<identifier> key </identifier>

<symbol> = </symbol>

<integerConstant> 90 </integerConstant>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> do </keyword>

<identifier> square </identifier>

<symbol> . </symbol>

<identifier> decSize </identifier>

<symbol> ( </symbol>

<symbol> ) </symbol>

<symbol> ; </symbol>

<symbol> } </symbol>

<keyword> if </keyword>

<symbol> ( </symbol>

<identifier> key </identifier>

<symbol> = </symbol>

<integerConstant> 88 </integerConstant>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> do </keyword>

<identifier> square </identifier>

<symbol> . </symbol>

<identifier> incSize </identifier>

<symbol> ( </symbol>

<symbol> ) </symbol>

<symbol> ; </symbol>

<symbol> } </symbol>

<keyword> if </keyword>

<symbol> ( </symbol>

<identifier> key </identifier>

<symbol> = </symbol>

<integerConstant> 131 </integerConstant>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> let </keyword>

<identifier> direction </identifier>

<symbol> = </symbol>

<integerConstant> 1 </integerConstant>

<symbol> ; </symbol>

<symbol> } </symbol>

<keyword> if </keyword>

<symbol> ( </symbol>

<identifier> key </identifier>

<symbol> = </symbol>

<integerConstant> 133 </integerConstant>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> let </keyword>

<identifier> direction </identifier>

<symbol> = </symbol>

<integerConstant> 2 </integerConstant>

<symbol> ; </symbol>

<symbol> } </symbol>

<keyword> if </keyword>

<symbol> ( </symbol>

<identifier> key </identifier>

<symbol> = </symbol>

<integerConstant> 130 </integerConstant>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> let </keyword>

<identifier> direction </identifier>

<symbol> = </symbol>

<integerConstant> 3 </integerConstant>

<symbol> ; </symbol>

<symbol> } </symbol>

<keyword> if </keyword>

<symbol> ( </symbol>

<identifier> key </identifier>

<symbol> = </symbol>

<integerConstant> 132 </integerConstant>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> let </keyword>

<identifier> direction </identifier>

<symbol> = </symbol>

<integerConstant> 4 </integerConstant>

<symbol> ; </symbol>

<symbol> } </symbol>

<keyword> while </keyword>

<symbol> ( </symbol>

<symbol> ~ </symbol>

<symbol> ( </symbol>

<identifier> key </identifier>

<symbol> = </symbol>

<integerConstant> 0 </integerConstant>

<symbol> ) </symbol>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> let </keyword>

<identifier> key </identifier>

<symbol> = </symbol>

<identifier> Keyboard </identifier>

<symbol> . </symbol>

<identifier> keyPressed </identifier>

<symbol> ( </symbol>

<symbol> ) </symbol>

<symbol> ; </symbol>

<keyword> do </keyword>

<identifier> moveSquare </identifier>

<symbol> ( </symbol>

<symbol> ) </symbol>

<symbol> ; </symbol>

<symbol> } </symbol>

<symbol> } </symbol>

<keyword> return </keyword>

<symbol> ; </symbol>

<symbol> } </symbol>

<symbol> } </symbol>

</tokens>

ArrayTestT.xml

<tokens>

<keyword> class </keyword>

<identifier> Main </identifier>

<symbol> { </symbol>

<keyword> function </keyword>

<keyword> void </keyword>

<identifier> main </identifier>

<symbol> ( </symbol>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> var </keyword>

<identifier> Array </identifier>

<identifier> a </identifier>

<symbol> ; </symbol>

<keyword> var </keyword>

<keyword> int </keyword>

<identifier> length </identifier>

<symbol> ; </symbol>

<keyword> var </keyword>

<keyword> int </keyword>

<identifier> i </identifier>

<symbol> , </symbol>

<identifier> sum </identifier>

<symbol> ; </symbol>

<keyword> let </keyword>

<identifier> length </identifier>

<symbol> = </symbol>

<identifier> Keyboard </identifier>

<symbol> . </symbol>

<identifier> readInt </identifier>

<symbol> ( </symbol>

<stringConstant> HOW MANY NUMBERS?  </stringConstant>

<symbol> ) </symbol>

<symbol> ; </symbol>

<keyword> let </keyword>

<identifier> a </identifier>

<symbol> = </symbol>

<identifier> Array </identifier>

<symbol> . </symbol>

<identifier> new </identifier>

<symbol> ( </symbol>

<identifier> length </identifier>

<symbol> ) </symbol>

<symbol> ; </symbol>

<keyword> let </keyword>

<identifier> i </identifier>

<symbol> = </symbol>

<integerConstant> 0 </integerConstant>

<symbol> ; </symbol>

<keyword> while </keyword>

<symbol> ( </symbol>

<identifier> i </identifier>

<symbol> &lt; </symbol>

<identifier> length </identifier>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> let </keyword>

<identifier> a </identifier>

<symbol> [ </symbol>

<identifier> i </identifier>

<symbol> ] </symbol>

<symbol> = </symbol>

<identifier> Keyboard </identifier>

<symbol> . </symbol>

<identifier> readInt </identifier>

<symbol> ( </symbol>

<stringConstant> ENTER THE NEXT NUMBER:  </stringConstant>

<symbol> ) </symbol>

<symbol> ; </symbol>

<keyword> let </keyword>

<identifier> i </identifier>

<symbol> = </symbol>

<identifier> i </identifier>

<symbol> + </symbol>

<integerConstant> 1 </integerConstant>

<symbol> ; </symbol>

<symbol> } </symbol>

<keyword> let </keyword>

<identifier> i </identifier>

<symbol> = </symbol>

<integerConstant> 0 </integerConstant>

<symbol> ; </symbol>

<keyword> let </keyword>

<identifier> sum </identifier>

<symbol> = </symbol>

<integerConstant> 0 </integerConstant>

<symbol> ; </symbol>

<keyword> while </keyword>

<symbol> ( </symbol>

<identifier> i </identifier>

<symbol> &lt; </symbol>

<identifier> length </identifier>

<symbol> ) </symbol>

<symbol> { </symbol>

<keyword> let </keyword>

<identifier> sum </identifier>

<symbol> = </symbol>

<identifier> sum </identifier>

<symbol> + </symbol>

<identifier> a </identifier>

<symbol> [ </symbol>

<identifier> i </identifier>

<symbol> ] </symbol>

<symbol> ; </symbol>

<keyword> let </keyword>

<identifier> i </identifier>

<symbol> = </symbol>

<identifier> i </identifier>

<symbol> + </symbol>

<integerConstant> 1 </integerConstant>

<symbol> ; </symbol>

<symbol> } </symbol>

<keyword> do </keyword>

<identifier> Output </identifier>

<symbol> . </symbol>

<identifier> printString </identifier>

<symbol> ( </symbol>

<stringConstant> THE AVERAGE IS:  </stringConstant>

<symbol> ) </symbol>

<symbol> ; </symbol>

<keyword> do </keyword>

<identifier> Output </identifier>

<symbol> . </symbol>

<identifier> printInt </identifier>

<symbol> ( </symbol>

<identifier> sum </identifier>

<symbol> / </symbol>

<identifier> length </identifier>

<symbol> ) </symbol>

<symbol> ; </symbol>

<keyword> do </keyword>

<identifier> Output </identifier>

<symbol> . </symbol>

<identifier> println </identifier>

<symbol> ( </symbol>

<symbol> ) </symbol>

<symbol> ; </symbol>

<keyword> return </keyword>

<symbol> ; </symbol>

<symbol> } </symbol>

<symbol> } </symbol>

</tokens>

JackTokenizer.java

/\*\*

 \* JackTokenizer.java

 \* Author   Raul Aguilar

 \* Date     November 14, 2020

 \* CS 220 2148

 \* JackTokenizer: Removes all comments and white space from the input

 \* stream and breaks it into Jack-language tokens, as specified by the

 \* Jack grammar.

 \*/

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.PrintWriter;

import java.io.FileOutputStream;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.Scanner;

public class JackTokenizer {

    private static String VALID\_SYMBOLS = "{}()[].,;+-\*/&|<>=~";

    private static String VALID\_IDENTIFIER = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789\_.$:";

    private PrintWriter outputFile = null;

    private Scanner inputFile;

    private KeywordTable keywords = new KeywordTable();

    private ArrayList<String> tokens = new ArrayList<>();

    private String cleanLine, token, symbol, stringConst;

    private int lineNumber, keyword, intConst;

    private TokenType tokenType;

    /\*\*

     \* Opens the input .jack file and gets ready to tokenize it

     \* @param fileName Name of the jack file

     \*/

    public JackTokenizer(String fileName) {

        try {

            inputFile = new Scanner(new FileReader(fileName));

        } catch (FileNotFoundException e) {

            System.err.println("File could not be found. Exiting program...");

            System.exit(0);

        }

    }

    /\*\*

     \* Checks if the .jack file has more line to read

     \*/

    public boolean hasMoreLines() {

        if(inputFile.hasNextLine()) {

            return true;

        } else {

            inputFile.close();

            return false;

        }

    }

    /\*\*

     \* Increments the line number by one. Intakes a sinle line from the

     \* file. First attempts to clean any single line comments, if there

     \* are none, then returns the line back. Then attempts to clean any

     \* syntax for multiline comments, otherwise returns back the line.

     \* After the line is clean of comments or whitespaces. Then

     \* tokenize the clean line.

     \*/

    public void parseNextLine() {

        lineNumber++;

        String rawLine = inputFile.nextLine();

        cleanLine = cleanMultiLineComments(cleanSingleLineComments(rawLine));

        parseTokens(cleanLine);

    }

    /\*\*

     \* Scans the line for "//" single line comments. Returns the line

     \* from the beginning up to the comment index. If there are no

     \* comments, returns the line back trimmed. However, there could

     \* still be multiline comments present.

     \* @param line The raw line taken from the file

     \* @return Line of jack code without single line comments

     \*/

    private String cleanSingleLineComments(String line) {

        int commentIndex = line.indexOf("//");

        if(commentIndex >= 0) {

            return line.substring(0, commentIndex).trim();

        }

        return line.trim();

    }

    /\*\*

     \* Cleans the line of multiline commments by scanning for "/\*",

     \* "\*", or "\*"/". If the start of a multiline comment is present in

     \* the line, then return the line from beginning up to index of"\\*"

     \* If the line starts with "\*", then line is in multiline comment

     \* and return empty line.

     \* If the end of a multiline comment is present, return everything

     \* after the index of occurance.

     \* Otherwise if no multiline comment is in line, then return the

     \* line as is.

     \* @param line Line taken after being cleaned of single line

     \* comments

     \* @return A clean line of multiline comments.

     \*/

    private String cleanMultiLineComments(String line) {

        int start = line.indexOf("/\*");

        int end = line.indexOf("\*/");

        if(start >= 0) {

            return line.substring(0, start);

        }

        if(line.startsWith("\*")) {

            return "";

        }

        if(end >= 0) {

            return line.substring(end+2);

        }

        return line;

    }

    /\*\*

     \* Recursively parse the line into tokens split up by symbols.

     \* Adds each token into an ArrayList containing all the tokens

     \* parsed from the file so far.

     \* @param line Current line from jack file being parsed into tokens

     \*/

    private void parseTokens(String line) {

        boolean foundSymbol = false;

        int symbolIndex = 0;

        if(line.length() == 1) {

            tokens.add(line.trim());

        }

        if(line.length() > 1) {

            for(int i = 0; i < line.length(); i++) {

                if(VALID\_SYMBOLS.contains(""+line.charAt(i))) {

                    // current character is a symbol

                    symbolIndex = i;

                    foundSymbol = true;

                    break;

                }

            }

            if(foundSymbol) {

                if(symbolIndex != 0) {

                    // add everything before the symbol if it doesn't

                    // have quotation marks

                    // Split the line first

                    if(line.charAt(0) != '"') {

                        tokens.addAll(Arrays.asList(line.substring(0, symbolIndex).trim().split(" ")));

                    } else {

                        tokens.add(line.substring(0, symbolIndex).trim());

                    }

                }

                // Add the symbol

                tokens.add(line.charAt(symbolIndex)+"".trim());

                // Continue parsing after the symbol

                parseTokens(line.substring(symbolIndex+1).trim());

            } else {

                // If no symbol is found in line, add the line

                tokens.add(line);

            }

        }

    }

    public void advance() {

        parseNextLine();

    }

    /\*\*

     \* Open the output XML file for writing

     \* @param fileName Name of the xml output file

     \*/

    public void xmlWriter(String fileName) {

        try {

            outputFile = new PrintWriter(new FileOutputStream(fileName));

        } catch(FileNotFoundException e) {

            System.err.println("Could not open output file " + fileName);

            System.err.println("Run program again, make sure to have write permissions, etc.");

            System.err.println("Program exiting...");

            System.exit(0);

        }

    }

    /\*\*

     \* Writes all of the tokens parsed from the input .jack file into

     \* xml tags to the output xml file

     \*/

    public void writeToXML() {

        outputFile.println("<tokens>");

        for(String t:tokens) {

            token = t;

            tokenType = tokenType(token);

            writeTokenTag(tokenType, token);

        }

        outputFile.println("</tokens>");

        outputFile.close();

    }

    private TokenType tokenType(String token) {

        try {

            intConst = Integer.parseInt(token);

            return TokenType.INT\_CONST;

        } catch (NumberFormatException notAnInteger) {

            keywords.KeywordTable();

            if(token.charAt(0) == '"') {

                stringConst = token.substring(1, token.length()-1);

                return TokenType.STRING\_CONST;

            } else if(VALID\_SYMBOLS.contains(token)) {

                switch(token) {

                    case "<":

                        symbol = "&lt;";

                        return TokenType.SYMBOL;

                    case ">":

                        symbol = "&gt;";

                        return TokenType.SYMBOL;

                    case "\"":

                        symbol = "&quot;";

                        return TokenType.SYMBOL;

                    case "'":

                        symbol = "&apos;";

                        return TokenType.SYMBOL;

                    case "&":

                        symbol = "&amp;";

                        return TokenType.SYMBOL;

                    default:

                        symbol = token;

                        return TokenType.SYMBOL;

                }

            } else if(keywords.contains(token)) {

                keyword = keywords.getKeyword(token);

                return TokenType.KEYWORD;

            } else {

                return TokenType.IDENTIFIER;

            }

        }

    }

    private void writeTokenTag(TokenType tokenType, String token) {

        if(tokenType == TokenType.STRING\_CONST) {

            outputFile.print("<stringConstant> ");

            outputFile.print(getStringConst());

            outputFile.println(" </stringConstant>");

        } else if(tokenType == TokenType.INT\_CONST) {

            outputFile.print("<integerConstant> ");

            outputFile.print(getIntConst());

            outputFile.println(" </integerConstant>");

        } else if(tokenType == TokenType.SYMBOL) {

            outputFile.print("<symbol> ");

            outputFile.print(getSymbol());

            outputFile.println(" </symbol>");

        } else {

            outputFile.print("<"+tokenType.name().toLowerCase()+"> ");

            outputFile.print(token);

            outputFile.println(" </"+tokenType.name().toLowerCase()+">");

        }

    }

    /\*\* Getters \*\*/

    public int getKeyword() {

        return keyword;

    }

    public String getSymbol() {

        return symbol;

    }

    public int getIntConst() {

        return intConst;

    }

    public String getStringConst() {

        return stringConst;

    }

    public int getLineNumber() {

        return lineNumber;

    }

    /\*\* Main \*\*/

    public static void main(String[] args) {

        Scanner keyboard = new Scanner(System.in);

        String inputFileName;

        // Get input file name

        System.out.println("Enter the .jack file name to compile:");

        inputFileName = keyboard.nextLine();

        keyboard.close();

        // Open jack file to read and tokenize

        JackTokenizer jackTokenizer = new JackTokenizer(inputFileName);

        while(jackTokenizer.hasMoreLines()) {

            jackTokenizer.advance();

        }

        // After tokenizing, write to xml file

        jackTokenizer.xmlWriter(inputFileName.substring(0, inputFileName.lastIndexOf('.'))+"T.xml");

        jackTokenizer.writeToXML();

        System.out.println("Done compiling. Program exiting...");

    }

}

TokenType.java

/\*\*

 \* TokenType.java

 \* Author   Raul Aguilar

 \* Date     November 13, 2020

 \* CS 220 2148

 \*/

public enum TokenType {

    KEYWORD, SYMBOL, INT\_CONST, STRING\_CONST, IDENTIFIER;

}

Kind.java

/\*\*

 \* Kind.java

 \* Author   Raul Aguilar

 \* Date     November 13, 2020

 \* CS 220 2148

 \*/

public enum Kind {

    FIELD, STATIC, LOCAL, ARGUMENT;

}

KeywordTable.java

/\*\*

 \* KeywordTable.java

 \* Author   Raul Aguilar

 \* Date     November 13, 2020

 \* CS 220 2148

 \*/

import java.util.HashMap;

public class KeywordTable {

    private HashMap<String, Integer> keywords = new HashMap<>();

    /\*\*

     \* Instantiate the table with reserved keywords and the integer

     \* constants representing them

     \*/

    public void KeywordTable() {

        keywords.put("class", 0);

        keywords.put("method", 1);

        keywords.put("function", 2);

        keywords.put("constructor", 3);

        keywords.put("int", 4);

        keywords.put("boolean", 5);

        keywords.put("char", 6);

        keywords.put("void", 7);

        keywords.put("var", 8);

        keywords.put("static", 9);

        keywords.put("field", 10);

        keywords.put("let", 11);

        keywords.put("do", 12);

        keywords.put("if", 13);

        keywords.put("else", 14);

        keywords.put("while", 15);

        keywords.put("return", 16);

        keywords.put("true", 17);

        keywords.put("false", 18);

        keywords.put("null", 19);

        keywords.put("this", 20);

    }

    /\*\*

     \* Return boolean if the current token is a keyword in the table

     \*/

    public boolean contains(String token) {

        return keywords.containsKey(token);

    }

    /\*\*

     \* Returns the integer constant associated with the current token,

     \* a keyword.

     \*/

    public int getKeyword(String keyword) {

        return keywords.get(keyword);

    }

}