# REGULAR EXPRESSION PARSER

By

Aaron Ward

Supervisor(s):

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
B.SC IN COMPUTING AND INFORMATION TECHNOLOGY
AT
INSTITUTE OF TECHNOLOGY BLANCHARDSTOWN
DUBLIN, IRELAND
2017

**Declaration**

I/We declare that this material, which I/We now submit for assessment, is entirely my/our own work and has not been taken from the work of others, except where otherwise stated. I/We have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references.

I/We understand that plagiarism, collusion, and copying are grave and serious offences and accept the penalties that would be imposed should I/we engage in plagiarism, collusion or copying. I acknowledge that copying someone elses assignment, or part of it, is wrong, and that submitting identical work to others constitutes a form of plagiarism. I/We have read and understood the colleges plagiarism policy 3AS08.

This material, or any part of it, has not been previously submitted for assessment for an academic purpose at this or any other academic institution. I have not allowed anyone to copy my work with the intention of passing it off as their own work.

Dated: 2017

Author: Aaron Ward

# Regular Expression Parser

The following report with describe a regular expression parser along with it's design and implementation. The parser shall take a given input and do an analysis of regular expressions. This report will consist of three parts. Firstly. A section of the lexical categories will be brought forward to discuss the architecture of the lexicon. A section on the implementation will be provided to describe the program being developed and how it interacts with the lexical entries. Lastly, a system architectures section will state the issues that were encountered during the process of development and how they were resolved. A conclusion will be given to summarize the points made during the report. Additionally, the project will be developed with Java and the report shall be with LaTeX.

## Lexical Categories

Firstly, the lexical categories are ordered from the sentence of *The/a man/men/woman/women bite(s)/like(s) the green dog*. Each word was specified with a part-of-speech category and a number (Singular or plural). Each abbreviation for the POS categories are taken in accordance of the results from the Stanford CoreNLP library. The categories are as follows:

- Determiner - **DT**
- Common Noun - **NN**
- Noun Plural Form - **NNS**
- Verb Base Form - **VB**
- Verb 3rd Person Singular - **VBZ**
- Adjectives - **JJ**

Therefore, the structure of the lexicon and it's parts-of-speech can be seen in table below.

| **Word** | **POS** | **Number** |
|---|---|---|
| The | DT | Singular |
| A | DT | Singular |
| Man | NN | Singular |
| Men | NNS | Plural |
| Woman | NN | Singular |
| Women | NNS | Plural |
| Bite | VB | Singular |
| Bites | VBZ | Singular |
| Like | VB | Singular |
| Likes | VBZ | Singular |
| Green | JJ | Singular |
| Dog | NN | Singular |

# Implementation

Initially, the parser program takes the input entered by the user. The input is then tokenized. These tokens are traversed to in while loop. Additionally, the lexicon file is scanned using the *java.util.Scanner* class. Each line from the lexicon is also tokenized for evaluation. Within each token of the users input, the file is searched to determine wether the word is contained

within the lexicon. If the word is found in the file, the second token (which is the part-of-speech tag) is added to an array list. This is done for all tokens in the users input. Following this, the array list is traversed and concatenated to a string. This string is then compared to a structure expected for it to be an acceptable regular expression. The result is then printed to the user interface. This can be seen in Figure 1.

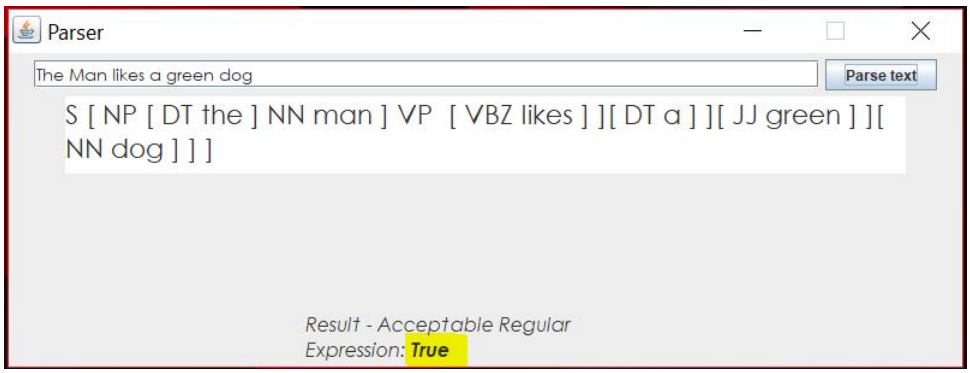


Figure 1: Result For Acceptable Input

Additionally, if a text is entered that does meet match the words used in the lexicon, or that is in the wrong order, the user interface is updated to display that it is not a suitable regular expression. This is displayed in Figure 2.

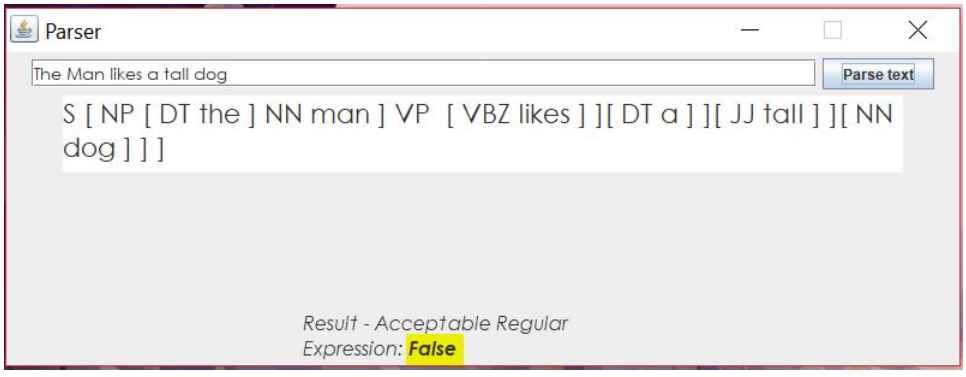


Figure 2: Result For Non-acceptable Input

# System Architecture

### Parser

The program's initial state is to display the user interface. As seen in Figure 1 and Figure 2, the view consists of a JTextField for the users input. A JTextArea can be seen beneath which displayed the users input in the bracket phrasal structure. This is performed by the **PrintBracket** Class, which checks the users input and print the POS tags and words entered. The interface also includes a button which calls **ParserButtonListener** when clicked for parsing the user input. At the bottom of the user interface a result is displayed, which is changed by the **ValidateInput** class. Each of the interactive components are static types so they may be modified by other classes.

### ParserButtonListener

This class implements the ActionListener interface to react to a users interaction with the "Parse Text" button. When clicked, the text from the input field is taken and converted to lower case characters. The string is tokenized into tokens for validation of an acceptable regular expression. Additionally, the string is tagged using the Stanford CoreNLP library. A ValidateInput and a PrintBracket object are created and the inputs are passed accordingly to each object to update the user interface and validate the input string. Initially, the validation and the output display were done within the same class, but this ran into problems when ever an input that did not satisfy the regular expression as it would only print the words that were found in the lexicon. Therefore, in order to get around this, the user input was validated and formatted for the bracket structure separately.

### ValidateInput

This class is used to evaluate the string entered by the user and determine if it is valid or not. The tokenized input is passed as a parameter in the validate() method and traversed in a while loop. Each token is then compared to the tokenized lexicon entries by line to see if the word is part of the systems lexicon. If it is, the words POS tag (second token in the **lexiconTokens** tokens) is added to a categories array list. After all tokens are traversed, the array list is converted to a string and evaluated if the parts-of-speech meet the requirements

to be an acceptable expression. If all tags are met, then the result is displayed as **true**, other it is **false**. See code below.

```
if(posTags.equals("DT NN VBZ DT JJ NN ") ||
    posTags.equals("DT NNS IN DT JJ NN ") ||
    posTags.equals("DT NNS VBZ DT JJ NN ") ||
    posTags.equals("DT NN IN DT JJ NN ")){

    Parser.acceptable.setText("<html>Result - Acceptable
    Regular Expression: <b>True</b></html>" );
}
else{
    Parser.acceptable.setText("<html>Result - Acceptable
    Regular Expression: <b>False</b></html>" );
}
```

**PrintBracket**

In this class, the used input is also tokenized and traversed with a while loop. In this loop two strings are made using regular expression to separate the tag from the tokenized input from the user. This is done because the output for each token from the Stanford CoreNLP library results in a **word_TAG** format. For example, if the word "dog" was entered, the tagged value is "dog_NN". See code below.

```
while(taggedTokens.hasMoreTokens()){
  String t = taggedTokens.nextToken();
  String s = t;
  t= t.replaceAll("[a-z]+_", "");
  s =s.replaceAll("_[A-Z]+", "");
  posTags.add(t);
  words.add(s);
}
```

The words and tags are added to their own respective array list. After all tokens are properly separated, the then evaluated to be printed into a bracket phrasal structure. This is done by

checking the size of the POS tag array list to give a pedicted structure output. For further illustration of the system architecture, please refer to the UML diagram displayed in Figure 3.

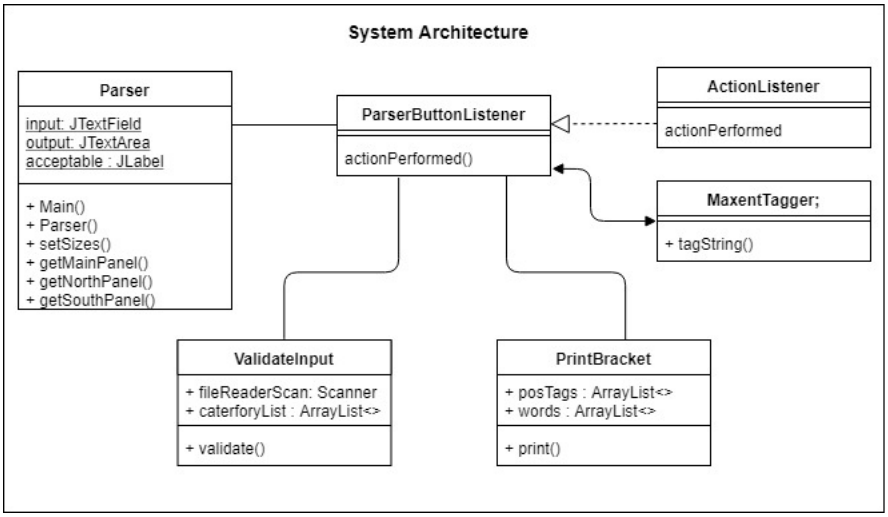


Figure 3: Result For Non-acceptable Input

# Conclusion

In conclusion, the program was able to take a users input, print a bracketed phrasal structure output to the user interface even if the the input was a valid regular expression or not, and was able to validate the input of the user in accordance to the sentence specified in the requirements.

# Appendices

## Appendix A
## Parser

```
public class Parser extends JFrame {
public static JPanel mainPanel ;
public JPanel mainNorthPanel;
public JPanel mainSouthPanel;

public static JTextField input;
public static JTextArea output;
public static JLabel acceptable;


public static void main(String [] args){
new Parser();
}

/**
* Constructer to initialise GUI components
*/
public Parser(){
        super.setTitle("Parser");
```

```
        add(getMainPanel());
        setSizes();
}


private void setSizes() {
        setSize(800, 300);
        setVisible(true);
        setResizable(false);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
}

/**
* Main JPanel
* @return
*/
public JPanel getMainPanel(){
        mainPanel = new JPanel(new BorderLayout());
        mainPanel.add(getNorthPanel(), BorderLayout.NORTH);
        mainPanel.add(getSouthPanel(), BorderLayout.SOUTH);
        return mainPanel;
}

/**
* Panel for text input, parse button and bracketed output
* @return
*/
public JPanel getNorthPanel(){

        JPanel northPanel = new JPanel();
```

```
        northPanel.setPreferredSize(new Dimension(300, 100));

        input = new JTextField(50);
        output = new JTextArea();
        output.setLineWrap(true);

        input.setFont(new Font("Century Gothic", Font.PLAIN , 14));
        output.setPreferredSize(new Dimension(700, 200));
        output.setFont(new Font("Century Gothic", Font.PLAIN ,22));
        output.setBackground(Color.WHITE);
        output.setEditable(false);

        JButton parse = new JButton("Parse text");
        parse.addActionListener(new ParseButtonListener());

        northPanel.add(input);
        northPanel.add(parse);
        northPanel.add(output);

        return northPanel;
}

/**
* For displaying results
* @return
*/
public JPanel getSouthPanel(){
        JPanel southPanel = new JPanel();
        acceptable = new JLabel("<html><b>Result - </b>
        Acceptable Regular Expression:</html>" + true);
        acceptable.setFont(new Font("Century Gothic", Font.ITALIC , 16
```

```
        acceptable.setPreferredSize(new Dimension(300, 40));

        southPanel.add(acceptable);
        return southPanel;
        }
}
```

# Appendix B
# ParserButtonListener

```
import edu.stanford.nlp.tagger.maxent.MaxentTagger;
public class ParseButtonListener implements ActionListener {

@Override
public void actionPerformed(ActionEvent arg0) {

        String input = Parser.input.getText();

        /**
        * Guard to make sure the user has entered
        * an input to be parsed
        */
        if(input.equals("")){
        JOptionPane.showMessageDialog(null, ""
        + "                    Please enter in some text",
        "Oops", JOptionPane.PLAIN_MESSAGE);
        }
        else{
        //Guard incase the user uses capital letters
        input = input.toLowerCase();
```

```
        StringTokenizer tokens = new StringTokenizer(input);
        ValidateInput validateInput = new ValidateInput();
        validateInput.validate(tokens);

        //Tag the users input using the stanford library
        MaxentTagger tagger = new MaxentTagger("src/english-left3words
        -distsim.tagger");
        String tagged = tagger.tagString(input);

        //calls the print method to print the bracketed structure to th
        PrintBracket printBracket = new PrintBracket();
        printBracket.print(tagged);
        }
    }
}
```

# Appendix C
# ValidateInput

```
public class ValidateInput {

public void validate(StringTokenizer tokens){

        Scanner fileReaderScan = null;
        ArrayList<String> categoryList = new ArrayList<String>();

        /**
        * traverse each token from the input
        */
        while(tokens.hasMoreTokens()){
        //Read the lexicon file
```

```
fileReaderScan = new Scanner(getClass().
getResourceAsStream("lexicon.txt"));
String thisToken = tokens.nextToken();

/**
* Traverse lines in the lexicon.txt
*/
while(fileReaderScan.hasNextLine())
{
//Tokenize the words of each line in the lexicon.txt
String scan = fileReaderScan.nextLine().toString();
StringTokenizer lexiconTokens = new StringTokenizer(scan);

/**
* traverse the tokens and add them to
*the list of POS categories
*/
while(lexiconTokens.hasMoreTokens()){
if(thisToken.equals(lexiconTokens.nextToken())){
        String thisLexToken = lexiconTokens.nextToken();
        categoryList.add(thisLexToken);
    }
   }
  }
}
fileReaderScan.close();

// Add each POS to a concatenated string
String posTags = "";
for(int i = 0; i < categoryList.size(); i++){
        posTags += categoryList.get(i) + " ";
```

```
        }
        System.out.println(posTags);

        /**
        * if valid then print true, else print false
        */
        if(posTags.equals("DT NN VBZ DT JJ NN ") ||
        posTags.equals("DT NNS IN DT JJ NN ") ||
        posTags.equals("DT NNS VBZ DT JJ NN ") ||
        posTags.equals("DT NN IN DT JJ NN ")){

            Parser.acceptable.setText("<html>Result - Acceptable
            Regular Expression: <b>True</b></html>" );
        }
        else{
            Parser.acceptable.setText("<html>Result -
            Acceptable Regular Expression: <b>False</b></html>" );
        }
    }
}
```

# Appendix D

# PrintBracket

```
public class PrintBracket {


public void print(String input){
        //Tagged output from the MaxentTagger is tokenized
        StringTokenizer taggedTokens = new StringTokenizer(input);
```

```
//Lists for the tags and words seperated
ArrayList<String> posTags = new ArrayList<String>();
ArrayList<String> words = new ArrayList<String>();

/**
* uses regular expressions seperate the users input
* from the tagged result
* from the Stanford library and adds them to two
* separate array lists
*/
while(taggedTokens.hasMoreTokens()){
    String t = taggedTokens.nextToken();
    String s = t;
    t= t.replaceAll("[a-z]+_", "");
    s =s.replaceAll("_[A-Z]+", "");
    posTags.add(t);
    words.add(s);
}

/**
* Print the bracketed results to the GUI
*/
if(posTags.size() < 2){
    Parser.output.setText("S [ " + posTags.get(0) + " " +
    words.get(0) + " ] ");
}
else if(posTags.size() == 2){
    Parser.output.setText("S [ NP [ " + posTags.get(0) + " "
    + words.get(0) + " ] " + posTags.get(1) + " " +
     words.get(1) + " ] ] ");
}
```

```
        else{
         String out = "S [ NP [ " + posTags.get(0) + " " +
         words.get(0) + " ] " + posTags.get(1) + " " +
         words.get(1) + " ] VP ";

         for(int i = 2; i < posTags.size(); i++){
              out += "[ " + posTags.get(i) + " " +  words.get(i) + " ]
          }
          Parser.output.setText(out + " ] ");
        }
    }
}
```