Natural Language Parser

David Thornton

B00152842

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 else's 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 (available here).

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.

Name: David Thornton Dated: 27/11/2025

(Printing your name here will be taken as a digital signature)

Contents

Lexical Categories

The lexical categories initially followed the basic rules for verbs, nouns etc, having words categorised by:

- DT – determiner
- NN - noun (singular)
- NNS – noun (plural)
- VB – verb
- JJ – adjective

All lexical entries also stored information for whether they were a root wood for a given phrase or not, having a “y” if they were a root and an “n” if not.

Later in the implementation to solve issues with the initial solution (described in the Problems encountered section), whether or not words were singular or plural were also stored, with 1 meaning singular, 2 meaning plural and 0 meaning either plural or singular. Additionally, the noun types were also expanded with the following structure:

- DT – determiner
- NN-PER – person noun (singular)
- NN-OBJ – object noun (singular)
- NNS-PER – person noun (plural)
- VB – verb
- JJ – adjective

The final lexicon is shown below:

Word	**Part of Speech**	**Root (Y/N)**	**Singular/Plural**
The	DT	N	0
A	DT	N	1
king	NN-PER	Y	1
kings	NNS-PER	Y	2
dislike	VB	Y	2
dislikes	VB	Y	1
like	VB	Y	2
likes	VB	Y	1
the	DT	N	0
new	JJ	N	0
cat	NN-OBJ	Y	1

Grammer Rules

The grammar rules for the program followed the grammar rules of the given sentence structure:

The/A king/kings dislike(s)/like(s) the new cat

The initial rules were made for parts of speech including:

- S – Sentences

- NP – noun phrase
- VP – verb phrase

Which either contained other parts of speech, including lexical categories above.

Later in the implementation to solve issues with the initial solution (described in the Problems encountered section), whether or not noun phrases were related to subject nouns (NP-SUB) or object nouns (NP-OBJ) were also stored. The final grammar rules were as follows:

- S -> NP-SUB VP
- NP-SUB -> DT NN-PER
- NP-SUB -> DT NNS-PER
- NP-OBJ -> DT JJ NN-OBJ
- VP -> VB NP-OBJ

Parser Logic

The parser verified the validity of sentences by first ensuring that the sentence followed the grammatical rules described above, and then check to ensure that consistent use of plural and singular parts of speech were used.

Parsing Grammer

As the sentences are structured in a tree format, a tree searching algorithm was used to verify the grammar rules of input sentences. The parser, starting with the root part of speech (the sentence) would check for available rules matching that part of speech, and would then check these possible rules to see if they were valid. The rules would be assessed as valid once a part of speech associated with a word (lexical entry) was assessed. If the next word in the sentence to be accessed matched the part of speech being checked, the word would be added to a tree node and would be returned to the previous node checked. If the word didn't match, a failure message is returned instead. If all the parts of speech of a rule were valid, this rule would also be valid, in which case the rule would be added to a tree node, along with its corresponding parts of speech as child nodes, and would then be returned to the previous rule that was checked. If the sentence is valid, the parser returns the tree structure for the sentence, if not a failure message is returned.

Plural/Single Validation

If the sentence follows the grammatical rules, the parser should return a tree structure of the sentence and its parts of speech. The plural/single validator (validator) can then check to ensure that plural and single word use is consistent. In this problem, the validator only needs to check for two things:

1. That all the words in each noun phrase are consistent
2. That verbs associated with noun phrases match their noun phrase

To verify that the words within a noun phrase match, the validator first finds the root word of that noun phrase and gets its "number" associated with whether is a singular or plural word. The validator then checks to ensure that all other words in the noun phrase have the same number as the root word. If this validation step is passed, the noun phrase number is passed to check if a verb associated with is matches. If the verb number matches the noun phrase number, the

validator returns a success message. If the noun phrase is inconsistent, or if the verb phrase doesn't match, a failure message is returned.

Problems Encountered

The first attempt at a solution failed entirely. This solution started at the words in the sentence, and then attempted to work back up to the sentences structure. This solution (or at least my implementation of this solution) couldn't handle the recursive structure of sentences (the verb phrase holding a noun phrase).

After further thinking through the problem, I realised that if sentences follow a tree structure, and I needed to find this tree output, a tree search algorithm would likely be the answer. Since I wanted to analyse the sentence in order of the words, and not by each level in the hierarchy, a depth first search algorithm was more logical than a breath first search.

My initial attempt at this solution worked, after adding a variable within the part of speech (POS) objects to store whether they were words (leaf nodes) or not.

The initial solution, with basic noun and noun phrase types, had several issues, leading to sentences which should not pass passing. There were three main faults in the system:

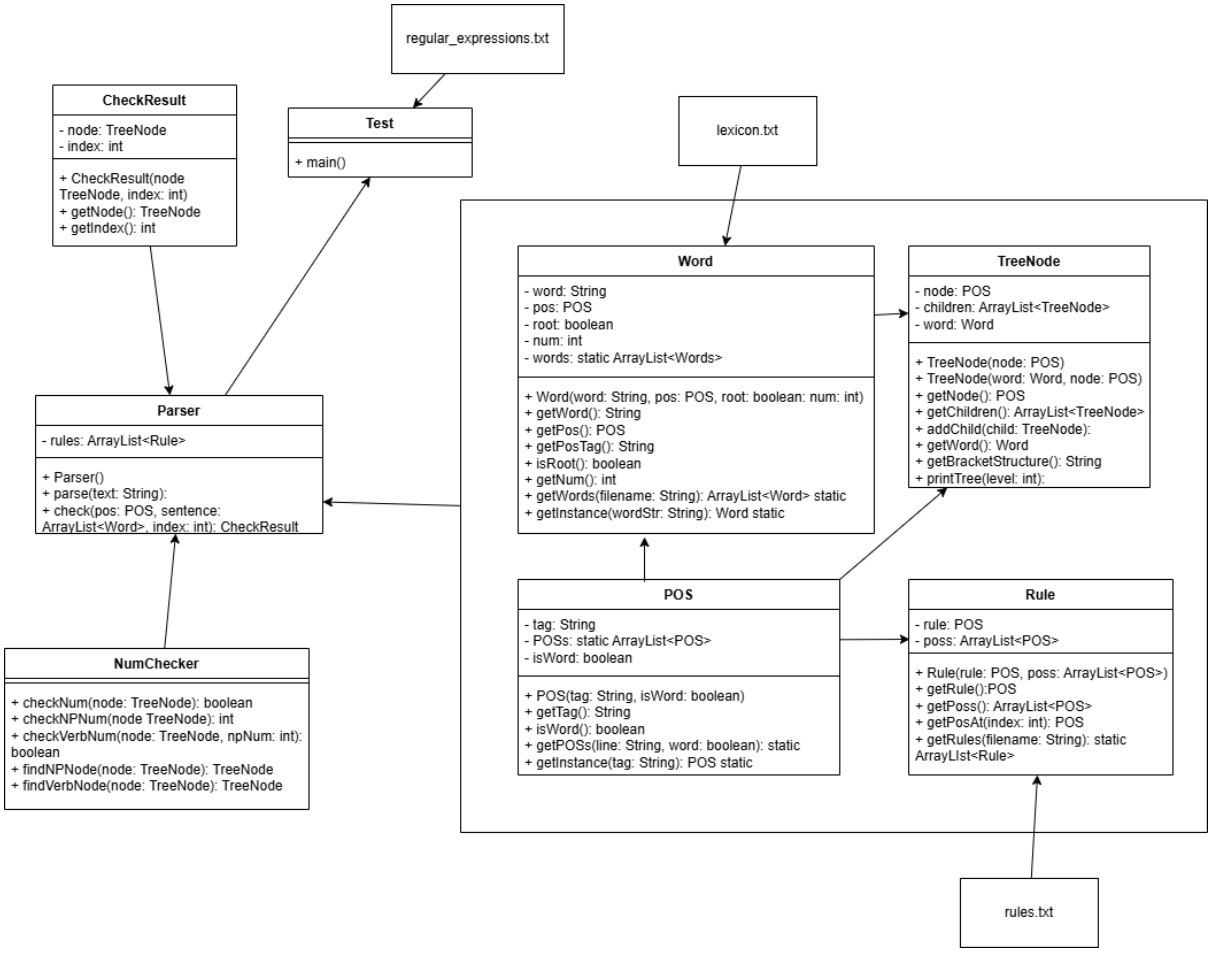
1. Plural/singular words consistency wasn't check – sentences like "A kings likes the new cat"
2. Any noun phrase could go in the place of any other – sentences like "The new cat likes the king" would pass
3. Any noun could be placed into any noun phrase – sentences like "The cat likes the new king" would pass

To ensure that plural and singular word uses was consistent, I designed the validator described in the Parser Logic section. For this to work, I also had to add values to the lexicon entries, describing them as singular or plural, as described in the Lexical Categories section

To ensure that the correct noun phrases were used in the correct places, I changed the rules to have different types of noun phrases, one for subject noun phrases and one for object noun phrases. Although this solved the issue of incorrect noun phrase usage, incorrect nouns could still be used within these noun phrases (object nouns used in subject noun phrases and vice versa).

To ensure that the correct nouns were used in the correct places, I changes the lexical categories and their corresponding rules to reflect the different types of nouns present, person and object nouns. The combination of these two steps solved the issues related to incorrect nouns and noun phrases. No changes to the original parser were needed.

System Architecture



The main parser program has a check method which performs the grammar check. This method is called in the parser constructer.

The number checker class is what checks for consistent uses of plural and singular words. It has methods to separately evaluate the noun phrase, check the verb phrase against the noun phrase, and to find relevant noun phrases and corresponding verbs. The main check num method, which calls the methods to check the noun phase and verb, is called by the parser object.

Words, rules and POS (parts of speech) each have their own class objects to represent them, and have method convert their text representations to objects at run time, which are called by the main parser program.

The sentence structure is stored as a tree using the TreeNode object class. This class also has methods to recursively construct the bracketed phrasal structure and to print the structure of the tree itself.

The check result class is what is returned by the recursive check method in the parser. This stores the sub tree for that check and the index of the current word.

The test class has the main method and is where the parser is called.

Example Execution

First, the test class create a parser object.

```
Parser parser = new Parser();
```

This constructor calls methods to read the text files and create object for words, rules and POSs.

```
try {
    this.rules = Rule.getRules(filename: "rules.txt");
    Word.getWords(filename: "lexicon.txt");
} catch (Exception e) {
    e.printStackTrace();
}
```

Then the test class calls the parse method on the sentence to be tested.

```
parser.parse(text: "A king likes the new cat");
```

Now the parser then attempts to convert the sentence string into an array of word objects

```
ArrayList<Word> sentence = new ArrayList<>();
for (String word : text.split(regex: " ")) {
    try {
        sentence.add(Word.getInstance(word));
    } catch (Exception e) { // or throw error if word not found
        System.out.println("Word not found: " + word);
        return;
    }
}
```

If this step fails, the parser outputs the following message and stops

```
Test 2: Parsing sentence 'The quick brown fox jumps' should fail.
Word not found: quick
```

If this is successful, the parser will begin to validate the sentence

```
// parse string
try {
    // start parsing sentence starting with the sentence pos
    CheckResult result = check(POS.getInstance(tag: "S"), sentence, index: 0);
    TreeNode root = result.getNode(); // get root of parse tree
    if (result.getIndex() == sentence.size()) { // if entire sentence parsed and sentence length is correct
        NumChecker numChecker = new NumChecker();
        // check tree for singular/plural matching
        if (!numChecker.checkNum(root)) {
            System.out.println(x: "Number agreement error.");
            return;
        }
        // if match, print success and bracket structure
        System.out.println(x: "Parse successful.");
        // print bracket structure and tree
        System.out.println(x: "Bracket Structure:");
        System.out.println(root.getBracketStructure());
        System.out.println(x: "Tree Structure:");
    } else {
        System.out.println(x: "Parse failed.");
    }
    root.printTree(level: 0);
} catch (Exception e) {
    e.printStackTrace();
}
```

The validator checks the sentence to ensure it follows the correct grammatical structure using the check method (as described in the Parser Logic, Parsing Grammer section)

```java
public CheckResult check(POS pos, ArrayList<Word> sentence, int index) {
    // if POS is a word, check if current word matches
    if (pos.isWord()) {
        if (sentence.get(index).getPosTag().equals(pos.getTag())) {
            return new CheckResult(new TreeNode(sentence.get(index), pos), index + 1); // if matches return word
                                                                                       // node and increment index
        } else {
            return new CheckResult(new TreeNode(sentence.get(index), pos), -1); // else return failure
        }
    }

    // if not word, check each possible rule for this subtree
    for (Rule rule : rules) {
        if (rule.getRule().getTag().equals(pos.getTag())) { // if rule matches current POS
            TreeNode parent = new TreeNode(pos);
            int currI = index;
            boolean match = true;
            // check all child POS rules
            for (int i = 0; i < rule.getPoss().size(); i++) {
                CheckResult child = check(rule.getPosAt(i), sentence, currI);
                // if any child fails, break and try next rule
                if (child.getIndex() == -1) {
                    match = false;
                    break;
                }

                // else add child node and update current index
                parent.addChild(child.getNode());
                currI = child.getIndex();

            }
            // if all children matched, return parent node and updated index
            if (match) {
                return new CheckResult(parent, currI);
            }
        }
    }
    // if no rules matched, return failure
    return new CheckResult(new TreeNode(pos), -1);
}
```

If this step fails, the parser outputs the following message and execution stops

```
Test 3: Parsing sentence 'A new cat likes the king' should fail.
Parse failed.
```

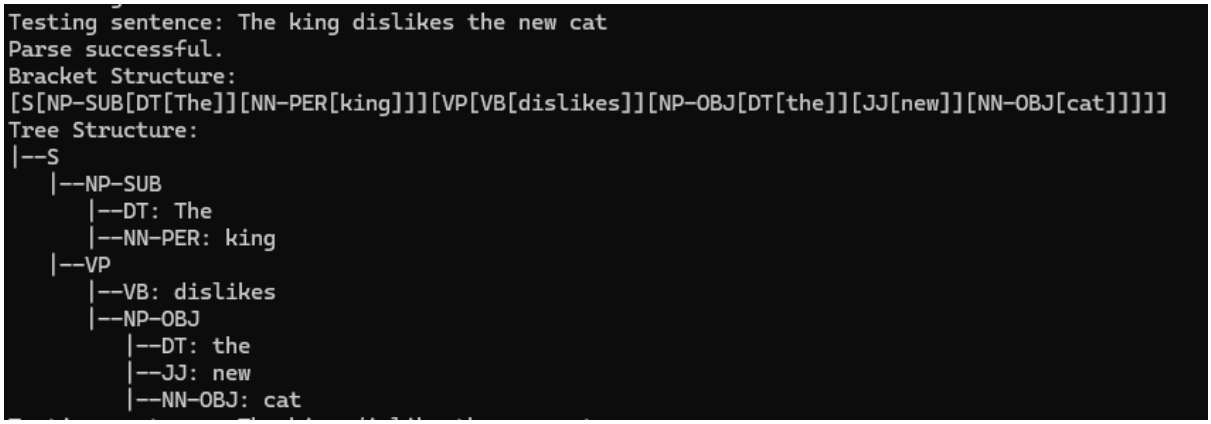
If this step succeeds, the parser will attempt to verify the consistency of singular/plural use, as described in Logic Parsing, Plural/Singular Validation section

```java
// check if the Noun Phrase is consistent
public int checkNPNum(TreeNode npNode) {
    // get the children of the NP node
    ArrayList<TreeNode> children = npNode.getChildren();
    int num = -1;

    // determine the number of the NP root
    for (TreeNode child : children) {
        if (child.getWord() != null && child.getWord().isRoot()) {
            num = child.getWord().getNum();
            break;
        }
    }
    // if no root found return -1
    if (num == -1) {
        return -1;
    }
    // check if other words match root number
    for (TreeNode child : children) {
        if (child.getWord() != null && !child.getWord().isRoot()) { // if node is a word and isnt root
            if (child.getWord().getNum() != num && child.getWord().getNum() != 0) {
                // if any dont match return -1
                return -1;
            }
        }
    }
    // if all match return num
    return num;
}

// check if verb matches the noun phrase
public boolean checkVerbNum(TreeNode verbNode, int npNum) {
    // get the verb word
    Word verbWord = verbNode.getWord();
    if (verbWord == null) {
        return false;
    }
    // get the number of the verb
    int verbNum = verbWord.getNum();

    // compare verb number with NP number
    return verbNum == npNum;
}
```

If this step fails, the parser outputs the following message and execution stops

```
Testing sentence: The king like the new cat
Number agreement error.
```

If this step succeeds, the parser will output a message saying so, along with the bracket phrase structure of the sentence and the sentence tree structure



Appendix

lexicon.txt

The DT n 0

A DT n 1

king NN-PER y 1

kings NNS-PER y 2

dislike VB y 2

dislikes VB y 1

like VB y 2

likes VB y 1

the DT n 0

new JJ n 0

cat NN-OBJ y 1

rules.txt

S NP-SUB NP-OBJ VP

DT NN-PER NNS-PER NN-OBJ JJ VB

S -> NP-SUB VP

NP-SUB -> DT NN-PER

NP-SUB -> DT NNS-PER

NP-OBJ -> DT JJ NN-OBJ

VP -> VB NP-OBJ

regular_expressions.txt

The king likes the new cat

The king like the new cat

The king dislikes the new cat

The king dislike the new cat

The kings likes the new cat

The kings like the new cat

The kings dislikes the new cat

The kings dislike the new cat

A king likes the new cat

A king like the new cat

A king dislikes the new cat

A king dislike the new cat

A kings likes the new cat

A kings like the new cat

A kings dislikes the new cat

A kings dislike the new cat

CheckResults.java

```
// class to hold results from recursive checks
// holds the current word index and current sentence subtree root associated with current node


public class CheckResult {
   private TreeNode node;
   private int index;


   public CheckResult(TreeNode node, int index) {
      this.node = node;
      this.index = index;
   }
```

```
    public TreeNode getNode() {
        return node;
    }

    public int getIndex() {
        return index;
    }

}
```

NumChecker.java

```
import java.util.ArrayList;

// Class to check if singular/plural usage stays consistent

public class NumChecker {
    // check if singular/plural usage is consistent between subject and verb
    public boolean checkNum(TreeNode node) {
        // check if the NP is consistent
        int npNum = checkNPNum(findNPNode(node));
        if (npNum == -1) {
            return false;
        }

        // if it is, check if the verb matches the NP
        return checkVerbNum(findVerbNode(node), npNum);
    }

    // check if the Noun Phrase is consistent
    public int checkNPNum(TreeNode npNode) {
        // get the children of the NP node
```

```
    ArrayList<TreeNode> children = npNode.getChildren();
    int num = -1;

    // determine the number of the NP root
    for (TreeNode child : children) {
        if (child.getWord() != null && child.getWord().isRoot()) {
            num = child.getWord().getNum();
            break;
        }
    }
    // if no root found return -1
    if (num == -1) {
        return -1;
    }
    // check if other words match root number
    for (TreeNode child : children) {
        if (child.getWord() != null && !child.getWord().isRoot()) { // if node is a word and isnt root
            if (child.getWord().getNum() != num && child.getWord().getNum() != 0) {
                // if any dont match return -1
                return -1;
            }
        }
    }
    // if all match return num
    return num;
}

// check if verb matches the noun phrase
public boolean checkVerbNum(TreeNode verbNode, int npNum) {
    // get the verb word
    Word verbWord = verbNode.getWord();
```

```
    if (verbWord == null) {
        return false;
    }
    // get the number of the verb
    int verbNum = verbWord.getNum();

    // compare verb number with NP number
    return verbNum == npNum;
}

// find the subject noun phrase associated with the verb
public TreeNode findNPNode(TreeNode node) {
    // if current node matches, return it
    if (node.getNode().getTag().equals("NP-SUB")) {
        return node;
    }
    // else search children
    for (TreeNode child : node.getChildren()) {
        TreeNode result = findNPNode(child);
        if (result != null) { // if child node matches, return it
            return result;
        }
    }
    return null; // otherwise return null
}

// find the main verb
public TreeNode findVerbNode(TreeNode node) {
    // if current node matches, return it
    if (node.getNode().getTag().equals("VB")) {
        return node;
```

```
        }
        // else search children
        for (TreeNode child : node.getChildren()) {
            TreeNode result = findVerbNode(child);
            if (result != null) { // if child node matches, return it
                return result;
            }
        }
        return null; // otherwise return null
    }
}
```

Parser.java

```
import java.lang.reflect.Array;
import java.util.ArrayList;
import java.util.Iterator;

// main parser class, uses rules and lexicon to parse input text

public class Parser {
    private ArrayList<Rule> rules;

    // constructor takes in input text to be parsed
    public Parser() {
        // get rules and words from files
        try {
            this.rules = Rule.getRules("rules.txt");
            Word.getWords("lexicon.txt");
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
}

public void parse(String text) {
  // convert input text to list of Word instances
  ArrayList<Word> sentence = new ArrayList<>();
  for (String word : text.split(" ")) {
    try {
      sentence.add(Word.getInstance(word));
    } catch (Exception e) { // or throw error if word not found
      System.out.println("Word not found: \"" + word + "\"");
      return;
    }
  }

  // parse string
  try {
    // start parsing sentence starting with the sentence pos
    CheckResult result = check(POS.getInstance("S"), sentence, 0);
    TreeNode root = result.getNode(); // get root of parse tree
    if (result.getIndex() == sentence.size()) { // if entire sentence parsed and sentence length is
correct
      NumChecker numChecker = new NumChecker();
      // check tree for singular/plural matching
      if (!numChecker.checkNum(root)) {
        System.out.println("Number agreement error.");
        return;
      }
      // if match, print success and bracket structure
      System.out.println("Parse successful");
      // print bracket structure and tree
```

```
            System.out.println("Bracket Structure:");
            System.out.println(root.getBracketStructure());
            System.out.println("Tree Structure:");
            root.printTree(0);
        } else {
            System.out.println("Parse failed");
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

// main method to check for correct structure
// recursivly calls itself to check sub-structures
// depth first search of rules to find matching structure
// breaks if rule doesnt match
public CheckResult check(POS pos, ArrayList<Word> sentence, int index) {
    // if POS is a word, check if current word matches
    if (pos.isWord()) {
        if (sentence.get(index).getPosTag().equals(pos.getTag())) {
            return new CheckResult(new TreeNode(sentence.get(index), pos), index + 1); // if
matches return word
                                                    // node and increment index
        } else {
            return new CheckResult(new TreeNode(pos), -1); // else return failure
        }
    }

    // if not word, check each possible rule for this subtree
    for (Rule rule : rules) {
        if (rule.getRule().getTag().equals(pos.getTag())) { // if rule matches current POS
```

```
            TreeNode parent = new TreeNode(pos);
            int currI = index;
            boolean match = true;
            // check all child POS rules
            for (int i = 0; i < rule.getPoss().size(); i++) {
              CheckResult child = check(rule.getPosAt(i), sentence, currI);
              // if any child fails, break and try next rule
              if (child.getIndex() == -1) {
                match = false;
                break;
              }

              // else add child node and update current index
              parent.addChild(child.getNode());
              currI = child.getIndex();

            }
            // if all children matched, return parent node and updated index
            if (match) {
              return new CheckResult(parent, currI);
            }
          }
        }
        // if no rules matched, return failure
        return new CheckResult(new TreeNode(pos), -1);
    }
}
```

POS.java

```
import java.util.ArrayList;
```

```
// class to represent a Part of Speech (POS) tag

public class POS {
   private String tag;
   private static ArrayList<POS> POSs = new ArrayList<>();
   private boolean isWord;

   private POS(String tag, boolean isWord) {
      this.tag = tag;
      this.isWord = isWord;
   }

   public String getTag() {
      return tag;
   }

   public boolean isWord() {
      return isWord;
   }

   // creates POS instances from a line of tags read in from rules file
   public static void getPOSs(String line, boolean word) {
      String[] parts = line.split(" ");
      for (String part : parts) {
         POSs.add(new POS(part, word));
      }
   }

   // returns POS instance for a given tag
   // (or exception if not found)
   public static POS getInstance(String tag) throws Exception {
```

```
    for (POS pos : POSs) {
      if (pos.tag.equals(tag)) {
        return pos;
      }
    }
    throw new Exception("POS tag not found: " + tag);
  }
}
```

Rule.java

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

// class to represent a grammar rule

public class Rule {
  private POS rule;
  private ArrayList<POS> poss;

  public Rule(POS rule, ArrayList<POS> poss) {
    this.rule = rule;
    this.poss = poss;
  }

  public POS getRule() {
    return rule;
  }

  public ArrayList<POS> getPoss() {
    return poss;
  }
```

```
    public POS getPosAt(int index) {
        return poss.get(index);
    }

    // reads rules from rules file
    public static ArrayList<Rule> getRules(String filename) throws Exception {
        FileReader fr = new FileReader(filename);
        BufferedReader br = new BufferedReader(fr);
        ArrayList<Rule> rules = new ArrayList<>();
        POS.getPOSs(br.readLine(), false); // read non-word POS tags
        POS.getPOSs(br.readLine(), true); // read word POS tags
        String line;
        // read each rule line
        while ((line = br.readLine()) != null) {
            String[] parts = line.split(" -> ");
            POS rule = POS.getInstance(parts[0].trim());
            ArrayList<POS> poss = new ArrayList<>();
            for (String pos : parts[1].split(" ")) {
                poss.add(POS.getInstance(pos));
            }
            rules.add(new Rule(rule, poss));
        }
        br.close();
        return rules;
    }
}
```

Test.java

```
import java.io.BufferedReader;
import java.io.FileReader;
```

```
// test class
// checks some strings to check parser functionality, and all combinations of singular/plural sentences from a file to test number agreement

public class Test {
   public static void main(String[] args) {
      Parser parser = new Parser();
      System.out.println("Starting Parser Test");
      System.out.println("--------------------");
      System.out.println("Test 1: Parsing sentence 'A king likes the new cat' should succeed.");
      parser.parse("A king likes the new cat");
      System.out.println();
      System.out.println("Test 2: Parsing sentence 'The quick brown fox jumps' should fail.");
      parser.parse("The quick brown fox jumps");
      System.out.println();
      System.out.println("Test 3: Parsing sentence 'A new cat likes the king' should fail.");
      parser.parse("A new cat likes the king");
      System.out.println();
      System.out.println("Test 4: Parsing sentence 'The king likes the cat' should fail.");
      parser.parse("The king likes the cat");
      System.out.println();
      System.out.println("Test 5: Parsing sentence 'A cat likes the new king' should fail.");
      parser.parse("A cat likes the new king");
      System.out.println();

      System.out.println();
      System.out.println("--------------------");
      System.out.println();
      System.out.println("Now test all possible combinations of singular/plural.");
      // reads sentences from regular_expressions.txt and tests each one
      try {
```

```
            FileReader fr = new FileReader("regular_expressions.txt");
            BufferedReader br = new BufferedReader(fr);
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println("Testing sentence: " + line);
                parser.parse(line);
                System.out.println();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

TreeNode.java

```
import java.util.ArrayList;

// class to hold results from recursive checks in tree structure

public class TreeNode {
    private POS node;
    private ArrayList<TreeNode> children;
    private Word word;

    // constructor for non word/leaf nodes
    public TreeNode(POS node) {
        this.node = node;
        this.children = new ArrayList<>();
        this.word = null;
    }
```

```
// constructor for word/leaf nodes
public TreeNode(Word word, POS node) {
   this.node = node;
   this.children = new ArrayList<>();
   this.word = word;
}

public POS getNode() {
   return node;
}

public ArrayList<TreeNode> getChildren() {
   return children;
}

public void addChild(TreeNode child) {
   children.add(child);
}

public Word getWord() {
   return word;
}

// returns bracket structure representation of the tree
public String getBracketStructure() {
   if (children.isEmpty()) { // if leaf node
      return "[" + node.getTag() + "[" + word.getWord() + "]" + "]"; // return [POS[word]]
   } else { // else non-leaf node
      StringBuilder sb = new StringBuilder();
      sb.append("[").append(node.getTag()); // start with [POS
      for (TreeNode child : children) { // get all children bracket structures
```

```
            sb.append(child.getBracketStructure());
        }
        sb.append("]"); // close the POS bracket
        return sb.toString();
    }
  }

  // prints the tree structure in a readable format
  public void printTree(int level) {
    // print indentation based on level
    for (int i = 0; i < level; i++) {
      System.out.print("   ");
    }
    // print current node
    if (word != null) {
      System.out.println("|--" + node.getTag() + ": " + word.getWord());
    } else {
      System.out.println("|--" + node.getTag());
    }
    // recursively print children
    for (TreeNode child : children) {
      child.printTree(level + 1);
    }

  }
}
```

Word.java

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

```
// holds information about a word in the lexicon
// including its part of speech, if it is a root word, its number (singular/plural) and the word itself

public class Word {
   private String word;
   private POS pos;
   private boolean root;
   private int num;
   private static ArrayList<Word> words = new ArrayList<>();

   public Word(String word, POS pos, boolean root, int num) {
      this.word = word;
      this.pos = pos;
      this.root = root;
      this.num = num;
   }

   public String getWord() {
      return word;
   }

   public POS getPos() {
      return pos;
   }

   public String getPosTag() {
      return pos.getTag();
   }

   public boolean isRoot() {
      return root;
```

```
}

public int getNum() {
   return num;
}

// reads words from lexicon file
public static ArrayList<Word> getWords(String filename) throws Exception {
   FileReader fr = new FileReader(filename);
   BufferedReader br = new BufferedReader(fr);
   String line;
   while ((line = br.readLine()) != null) {
      String[] parts = line.split(" ");
      String wordStr = parts[0].trim();
      POS pos = POS.getInstance(parts[1].trim());
      boolean root = parts.length > 2 && parts[2].trim().equals("y");
      int num = parts.length > 3 ? Integer.parseInt(parts[3].trim()) : 0;
      words.add(new Word(wordStr, pos, root, num));
   }
   br.close();
   return words;
}

// gets a Word instance from the list by matching the word string
// (or exception if not found)
public static Word getInstance(String wordStr) throws Exception {
   for (Word word : words) {
      if (word.word.equals(wordStr)) {
         return word;
      }
   }
```

```
        throw new Exception("Word not found: " + wordStr);
    }
}
```