4-20-2011

A Bottom-Up Design and Implementation for Ambiguity-Compatible Natural Language Sentence Parsing

Elise Thrasher
Trinity University, ethrashe@trinity.edu

A BOTTOM-UP DESIGN AND IMPLEMENTATION FOR AMBIGUITY-COMPATIBLE NATURAL LANGUAGE SENTENCE PARSING

Elise Thrasher

A departmental senior thesis submitted to

the Department of Computer Science at Trinity University

in partial fulfillment of the requirements for graduation with departmental honors.

April 20, 2011

_____ _____

Thesis Advisor Department Chair

_____

Associate Vice President
For Academic Affairs

A Bottom-Up Design and Implementation for Ambiguity-Compatible Natural Language Sentence Parsing

Elise Thrasher

Abstract

Although many theory-focused computer science textbooks give a brief outline of a context-free grammar model of natural language, the approach is often vague and, in reality, greatly simplifies the English language's grammatical complexities. When applied to commonly-seen sentences, these sentence parsing models often fall short. In this paper, I detail my process of creating a programmable natural language context-free grammar that is able to parse (i.e. diagram) many common sentence forms, as well as the research which influenced the design of this project. In order to create a grammar that recognized the intricacies of the English language, I also incorporated the ability to identify and represent ambiguous sentences into my program. While the resulting program is not able to correctly parse every possible English sentence, ambiguous or not, it does function as an introduction to the field of computational linguistics and the difficulties present in this field.

Acknowledgements

The author would like to thank her parents for their support and encouragement in this great endeavor, and Dr. Paul Myers for his confidence in my abilities, as well as his guidance in this unfamiliar, though thoroughly intriguing, territory.

A Bottom-Up Design and Implementation for Ambiguity-Compatible Natural Language Sentence Parsing

Elise Thrasher

Table of Contents

List of Diagrams

1.Introduction

1.1. Motivation

The main goal of this project could be described as exploratory: attempt to create a project that is able to utilize aspects of both linguistics and computer science in an effective and demonstrative way. Rather than creating a completely new project or investigating some innovative, cutting-edge research, I felt it would be more beneficial to my goals to build some academic foundation in computational linguistics. Thus, I chose to pursue topics that were several years out of date, but were still very relevant to the field of computational linguistics. The processes of part-of-speech tagging and sentence parsing are used in many natural language processing systems to accomplish tasks like speech generation and grammar checking. In an effort to increase the complexity of the project and to produce a somewhat original resulting program, I also chose to incorporate the recognition and presentation of linguistic ambiguity into my project.

As a great deal of the work required for this project has been done previously, sometimes even decades ago, I chose to focus more on making my work understandable and accessible to a broader range of people rather than create a novel or innovative project. Once I refine the code to be more descriptive and easy to follow, I intend to release the program as a free, open-source reference tool. There are a many things that

the code I have created could grow into, if given the appropriate resources. I can envision the code being used in a database or sorting project to create an efficient dictionary, or in a human-computer interaction experiment, or even in an English project illustrating ambiguity. Because of the simplicity and versatility of the topics present in this project, the code may yet have a wide range of applications no one has yet considered.

1.2. Approach

To create a program which would not only diagram a sentence but also identify and represent some level of linguistic ambiguity, I broke the task into three main classes: Sentence Diagrammer, Organizer, and Parser. The Sentence Diagrammer creates the user interface and retrieves user input, which is sent to the Organizer to identify the part-of-speech and phrase structure, all possible permutations of which are recorded in the Parsers. The Sentence Diagrammer then retrieves the completed parse and reformats it for display on the user interface. If ambiguity is encountered in the Organizer, another Parser is created, so that the Sentence Diagrammer occasionally displays more than one diagram, if the multiple Parsers are deemed valid.

The Organizer is probably the most note-worthy function of the three because of its versatility. When an input is received, it goes through each word and identifies its part-of-speech classification based on the classification recorded for the previous word and several small dictionaries. The part-of-speech classifications are then used to determine the sentence's phrasal structure. This information is all saved in arrays, called parses, in a Parser so that, if the Organizer finds multiple part-of-speech classifications for a single word, a new Parser can be generated that includes the previous array's information but changes the last element to be the new classification. A new Parser is

also added when multiple phrase structures are identified. With multiple parses, the Organizer continues to loop through the sentences word-by-word, but now also loops through the Parsers within the sentence loop. As each word's part-of-speech classification possibilities are determined by the previous word's classification, multiple Parsers can very easily return widely varying diagrams for the same input. However, if the Organizer does not find an appropriate classification for the current word based on the previous word's classification, then the Parser is rejected and removed. This program setup not only keeps track of all identified interpretations of the input, but also removes the parses inconsistent with the grammar.

2.Background

2.1. Introduction

Although the topics of sentence parsing and part-of-speech tagging have been integral challenges in natural language processing almost since the field's beginning (and, some will argue, even before, because of the topics' relationships with the previously-established fields of linguistics and logic), most research has been with focuses other than sentence structure or sentence checking. This is probably due to the difficulty of the sentence generation problem as well as the versatility of a perfect solution, if it were found. A computer would not be able to interact with a user if it were not able to identify and define the topic of the user's sentences. For example, in the infamous Turing Test, a computer is tasked with the challenge of fooling a user into thinking it is a human speaker, using only a chat interface (Russell 2). The true level of functional intelligence that would be demonstrated by a computer able to complete this task is debatable, but, as the Turing Test was originally intended to act as a "satisfactory operational definition of intelligence", creating such a program is still viewed as a respectable goal. In the test, the computer must be able to not only produce comprehensible responses to the user's written questions, but also must produce appropriate ones. In order for a computer to produce an appropriate response, a program must be able to recognize the key words in a

sentence, most often the subject, verb, and, if one is present, the object, and create a response which refers to the user's chosen topic and, through this process, answer the user's questions appropriately.

Luckily, a great deal of work with word classification has already been performed in the field of linguistics, thus making the recognition of the subject, verb, and object relatively trivial. The idea of classifying parts-of-speech into categories, i.e. nouns for items and verbs for actions, goes back to Ancient Greece. One particular frontrunner in the field of linguistics was Dionysius Thrax of Alexandria who, circa 100 BC, created a "grammatical sketch of Greek […] which summarized the linguistic knowledge of his day" (Jurafsky 287). Although Thrax was not the first to classify words, his classifications "became the basis for practically all subsequent part-of-speech descriptions of Greek Latin, and most European languages for the next 2000 years" (Jurafsky 287). The modern lexical categories, remarkably consistent with Thrax's work, include nouns and pronouns, verbs and auxiliary verbs, adjectives, adverbs, prepositions, articles, and conjunctions. It is important to note here that some languages do not contain all parts-of-speech; Jurafsky, a professor of Linguistics at Stanford University, specifically mentions that Chinese words that perform a function analogous to English adjectives are sometimes interpreted as a subclass of verbs rather than their own category (290). Also, some researchers have added more word classifications to make their grammars more precise; most of the parts-of-speech Thrax identified contain subcategories that are occasionally treated as individual categories as the need arises. There are also categories, like interjections (Oh! Hey!), negatives (not, no) and politeness markers (please), that can be included for completeness's sake, but do not form or

contribute to the main foundations of a sentence (Jurafsky 296). The seven categories listed above (nouns, verbs, adjectives, adverbs, prepositions, articles, and conjunctions) are the most common parts-of-speech in English and thus the most studied.

2.2. Parts-of-Speech

While the process of identifying words as belonging to certain lexical categories may seem like a simple task to native speakers, defining rules that determine a word's part-of-speech can be surprisingly complex. There are two main characteristics that will help determine a word's part-of-speech: semantics and syntax. Semantics refers to a word's meaning in a given instance and syntax is the way the word relates to other words. For example, a word can be definitively classified as a noun if it adheres to both the semantic definition of a noun (i.e. representing a person, place, thing, or idea) and the syntactic definition (i.e. having the ability to be quantified, pluralized, and possessed) (Jurafsky 290). Verbs are defined semantically as "words referring to actions and processes" and can be identified syntactically through their morphological variances, as in verb tenses and conjugations which refer to the same action but designate when and by whom the action is performed (Jurafsky 290). Adjectives, another common part-of-speech, are descriptive words (semantics) that are most closely associated with nouns (syntax).

Adverbs are a great deal more complicated to quantify than the other parts-of-speech. They, like adjectives, are categorized as descriptive, modifying words, but are most often used in relation to verbs. However, they can also be combined with other adverbs and verb phrases. Jurafsky provides an example sentence from a 1985 paper by Schachter which contains multiple adverbs (italicized) surrounding a single noun and a single verb: "*Unfortunately*, John walked *home extremely slowly yesterday*" (Jurafsky

291). To make the "adverb" classification more intelligible, Jurafsky breaks the lexical category of into subcategories: directional/locative (*home*), degree (*extremely*), manner (*slowly*), and temporal (*yesterday*). By separating the category of adverb by the words' applications, it is easy to appreciate the difficulty in defining a concrete semantic and syntactic definition for this part-of-speech.

The first four lexical categories previously defined (nouns, verbs, adjectives, and adverbs) are termed "open classes" as they contain an ever-growing number of words; it would be impossible to list all elements within these categories as these lists they are constantly being revised and expanded (Russell 890). Conversely, closed classes are categories in which all elements can be listed without a great deal of effort. The remaining three parts-of-speech are identified as closed classes: prepositions, articles, and conjunctions. Jurafsky lists approximately 50 different words as prepositions, which is quite small when compared to the constantly-expanding list of nouns (292). However, articles represent the smallest distinct class discussed here, consisting of only three elements: "a", "an", and "the". However, some people include "this" and "that" as articles as well (Jurafsky 293). The number of words identified as conjunctions rests somewhere in the middle. Prepositions are words which are semantically relational, often dealing with time or space relationships, that occur preceding a noun. Articles also occur before nouns and are used to mark a noun as indefinite ("a chair", thus any instance of chair) or definite ("the chair", thus this specific instance of chair) (Jurafsky 293). Finally, conjunctions are "used to join two phrases, clauses, or sentences" and occur between the two items they are joining.

There are two subcategories of parts-of-speech that are worth mentioning at this point in the discussion: pronouns, a subcategory of nouns, and auxiliary verbs, a subcategory of verbs. Pronouns are "forms that often act as a kind of shorthand for referring to some [understood] noun phrase or entity or event" (Jurafsky 293). The precise definition of a noun phrase will be discussed later, but for the moment it is important to note that pronouns cannot be associated with articles and adjectives, unlike traditional nouns. Jurafsky breaks the subcategory of pronouns into a few other sub-classifications which deserve some note: personal pronouns (I, me, he, she, it, etc.), possessive pronouns (her, his, my, their, etc.), and wh-pronouns (what, who, where, etc.). Auxiliary verbs are "words (usually verbs) that mark certain semantic features of a main verb, including whether an action takes place in the present, past or future (tense), whether it is completed (aspect), whether it is negated (polarity), and whether an action is necessary, possible, suggested, desired, etc. (mood)" (Jurafsky 294). In short, auxiliary verbs are used before the main verb to provide more action-related information. The auxiliaries are italicized in the following sentences:

I *must* go to the store.

I *shouldn't* go to the store.

I *have* gone to the store.

I *am* going to the store.

I *will* go to the store.

It should be noted that both pronouns and auxiliary verbs are closed classes, and thus have a relatively small and bounded number of members.

2.3. Part-of-Speech Tagging

The identification of words in a given text as distinct parts-of-speech is called part-of-speech tagging. Identifying a word's part-of-speech "gives a significant amount of information about the word and its neighbors" (Jurafsky 288). For example, if you identify a word as an adjective, there is a good probability that the next word is a noun. The method of tagging words is relatively simple and mostly involves "selecting the most likely sequence of syntactic categories for the words in a sentence" (Allen 195). The "syntactic categories" are generally called "tag sets" and are based on the parts-of-speech classifications, but, when appropriate, these traditional classifications are expanded to allow for greater accuracy in identification; the tag set outlined in one text contains 36 separate categories (Allen 196). To tag a sentence or other string of words, you input the string and the tag set into the tagging algorithm, which will output the single best tag for each word (Jurafsky 298). It is notable that the most basic algorithms are only capable of returning a single tag for each word and do not consider the word's context in creating this classification. However, they are still able to provide a decent amount of accuracy for their level of simplicity.

Tagging algorithms are generally created to functon in one of two ways: through pre-defined rules, or stochastically (Jurafsky 299). Creating a rule-based tagger involves manually writing a tag set that results in the necessary and desired syntactic relationships. These rules generally refer to the already-identified words surrounding the current item in order to narrow down the list of valid parts-of-speech. A stochastic algorithm determines the most likely word classification through the analysis of a large training sample. By

reviewing the training text, the algorithm is able to create rules based on the lexical relationships present in the sample.

The first part-of-speech tagging algorithms, mostly created in the early 1960s, were based on a two-stage architecture. The program would first "use a dictionary to assign each word a list of potential parts-of-speech", then "[apply] large lists of hand-written […] rules to winnow down this list to a single part-of-speech for each word" (Jurafsky 300). For example, the ENGTWOL tagger, introduced in 1995 by Voutilainen, follows this process relatively closely: the tagger first determines all possible parts-of-speech for each word individually, and then the context of the other identified words is considered to help eliminate inconsistent tags (Jurafsky 301). Although this is a greatly simplified explanation of the ENGTWOL tagger (Jurafsky also notes that the architecture contains other probabilistic and syntactic determiners), it does demonstrate the similarity between the basic part-of-speech tagging procedure and a relatively-recent application.

2.4. Constituents

As lexical categorization has been covered earlier in this paper, it would be well-advised to discuss possible applications of these categories. When elements from certain lexical categories are combined, they form syntactic categories, or constituents. Constituency is when "groups of words […] behave as a single unit or phrase" (Jurafsky 324). Most often, two different constituents, a noun phrase and a verb phrase, are said to comprise a sentence. A noun phrase is defined by Jurafsky as "a sequence of words surrounding at least one noun" (325) and a verb phrase "consists of a verb followed by assorted other things" (328). Although these descriptions may not appear particularly descriptive, they are entirely accurate. Noun phrases are designated by the presence of a noun, but are

categorized as a phrase due to the possibility of articles or adjectives associated with the noun also being present. For example, in the sentence “The big dog sleeps”, although “dog” is the noun, the entire phrase “the big dog” is considered a noun phrase as the article and adjective specify and modify the noun. Verb phrases contain a sentence’s verb, and also contain all elements that relate to the included verb. In the sentence above, “The big dog sleeps”, the verb phrase is made of the single word “sleeps”. However, in the sentence “The boy eats green vegetables”, the verb phrase is “eats green vegetables”, which, in turn, contains the noun phrase “green vegetables”:

Noun phrase	Verb phrase
		Noun phrase	
The boy	eats	green vegetables.	

Diagram 1: Phrasal Identification Example

In order to complete our description, there is another main constituent (phrasal structure) to introduce: the prepositional phrase. Prepositional phrases contain a preposition followed by a noun phrase and can occur within either a noun phrase or a verb phrase. The sentence “The big dog in the garden sleeps on the flowers” illustrates all phrase structures mentioned. The noun phrase (“the big dog in the garden”) is comprised of a traditional noun phrase (“the big dog”), along with a prepositional phrase (“in the garden”) which contains a noun phrase (“the garden”). The verb phrase (“sleeps on the flowers”) is comprised of a verb (“sleeps”) and another prepositional phrase (“on the flowers”) which contains another noun phrase (“the flowers”).

2.5. Context-Free Grammars

By defining how noun, verb, and prepositional phrases, along with other parts-of-speech, interlock, we are able to create a logical grammar that represents the English language with some level of accuracy. A logical grammar is quite similar to its linguistic counterpart, but much more quantifiable; a logical grammar is based on "a collection of rules that [define] a language as a set of allowable strings of words" (Russell 890). The rules consist of two types of elements: non-terminals and terminals. In the case of lexical and syntactic categorization, non-terminals are the constituents and terminals are the parts-of-speech. If one wishes to be even more specific regarding the rule format, it may be more proper to term the parts-of-speech "terminal categories" and the individual words within each part-of-speech classification as terminals, but this distinction is often ignored (Krulee 13). As there are many forms of logical grammars, we will focus on the one most often applied to work in natural language processing: context-free grammars, often abbreviated as CFGs. CFGs have a single non-terminal that can lead to any combination of non-terminals and/or terminals (Russell 889). As CFGs have a great deal of versatility in their generation, they are ideal for quantifying natural language in a simplified, comprehensible format. In fact, Russell claims that CFGs were first used by ancient Indian grammarians for the analysis of Shastric Sanskrit (919), while Jurafsky claims that the "idea of basing a grammar on constituent structure dates back to the psychologist Wilhelm Wundt" in a paper published in 1900 (327). However, both authors agree that the theory's popularity in the current field is mostly due to the work by Chomsky , published in 1956, and, independently, by Backus, published in 1959.

Context-free grammars consist of "a set of rules or productions, each of which expresses the ways that symbols of the language can be grouped and ordered together, and a lexicon of words and symbols" (Jurafsky 327). In a linguistic interpretation, the rules delineate the relationships between parts-of-speech and phrases. These rules are often referred to as a grammar. The aforementioned lexicon identifies individual words as their part-of-speech. Although the individual rules within a lexicon and grammar can be relatively simplistic, the combined grammar often gains an incredible amount of complexity with the many possible interpretations allowed by the English language. Michael Sipser, in a textbook on computational theory, introduces context-free grammars with the following English language example written in Backus Normal Form:

<SENTENCE> → <NOUN-PHRASE> <VERB-PHRASE>

<NOUN-PHRASE> → <CMPLX-NOUN> |
<CMPLX-NOUN> <PREP-PHRASE>

<VERB-PHRASE> → <CMPLEX-VERB> |
<CMPLX-VERB> <PREP-PHRASE>

<PREP-PHRASE> → <PREP> <CMPLX-NOUN>

<CMPLX-NOUN> → <ARTICLE> <NOUN>

<CMPLX-VERB> → <VERB> | <VERB> <NOUN-PHRASE>

<ARTICLE> → a | the
<NOUN> → boy | girl | flower
<VERB> → touches | likes | sees
<PREP> → with (101)

Diagram 2: Sipser Context-Free Grammar

In this example, constituents and parts-of-speech are both identified as non-terminals and are identified by their brackets (<…>) and capitalization. The symbol "→" designates that the item on the left can be replaced by the designations at the right, and the "|"

symbol allows multiple interpretations to be associated with the same left-hand non-terminal and is often read as “or”. From the designations defined above, the first six rules listed in this example comprise the grammar, where the last four rules can be identified as specifying the lexicon. With these rules, several sentences can be generated. For example:

1. a boy sees a girl
2. the boy touches the girl with a flower
3. a girl likes

Diagram 3: Examples of Sipser CFG-Derived Sentences

All three of these sentences can easily be generated by Sipser’s grammar and lexicon through the process demonstrated below with Sentence 1 (“a boy sees a girl”):

<SENTENCE> → <NOUN-PHRASE> <VERB-PHRASE>
→ <CMPLX-NOUN> <VERB-PHRASE>
→ <ARTICLE> <NOUN> <VERB-PHRASE>
→ a <NOUN> <VERB-PHRASE>
→ a boy <VERB-PHRASE>
→ a boy <CMPLX-VERB>
→ a boy <VERB> <NOUN-PHRASE>
→ a boy sees <CMPLX-NOUN>
→ a boy sees <ARTICLE> <NOUN>
→ a boy sees a <NOUN>
→ a boy sees a girl

Diagram 4: Sipser CFG Sentence Generation

However, even though Sentences 1, 2, and 3 can be generated by the Sipser’s example, it does not mean that these sentences are valid English sentences, nor does it mean that the grammar can generate all or most English sentences (even ignoring the exceptionally small example lexicon). Sentence 3, although easily generated, requires an additional noun phrase to be considered a valid English sentence, as the verb “likes” is considered a transitive verb and thus must refer to an item. This could be resolved by requiring the

<CMPLX-VERB> → <VERB> <NOUN-PHRASE> option when the <VERB> leads to "likes", but as the verb is not chosen until the later in the derivation, this proves inefficient as it requires backtracking. As for sentences that are considered "valid English" but cannot be generated through this example grammar, any sentence containing adjectives, adverbs, conjunctions, or any sentence beginning with a verb ("Eat your vegetables.") or question word ("Did you mow the lawn?") will never be created with these grammar rules. However, this is a decent example that illustrates some core principles of context-free grammars.

2.6. Parsing

Parsing is another important aspect utilized in conjunction with part-of-speech tagging to identify and understand natural language sentences. With parsing, when given an input sentence and a grammar, it can be determined whether the grammar can generate the sentence. Parsing can be described, at least in this context, as "the process of analyzing a string of words to uncover its phrase structure, according to the rules of the grammar" (Russell 892). In other words, part-of-speech tagging can be viewed as a necessary sub-task of parsing, as the tagging rules occur as part of the lexicon. The goal of parsing is to find all possible permutations that contain all words in the given input while abiding by the rules of the grammar to create a sentence; currently two main strategies exist to do so. A top-down parsing strategy begins with the knowledge that the input is a sentence, then attempts to create all possible permutations that can be derived from this interpretation and check the results against the original input to find the proper formatting. A bottom-up parsing strategy starts with the input and applies all possible rules to attempt to generate the base property.

The permutations generated from parsing are often represented in a tree form to better show the hierarchy of the items generated. The tree form includes a root, which is the single, base node of the tree, and leaves, which are the final nodes. For the purpose of assigning everything a name, the non-terminals which occur between the root and leaves will be referred to as "constituents" as they are often some derivation of a phrase structure. Connecting the root and leaves are branches, which show the path taken to generate the tree. In parsing, it is important to note that the resulting tree must have a single root node and that number of leaves must be equal to the number of words in the input. To provide an example of a parsing tree, we shall revisit the sentence "a boy sees a girl" we generated previously using Sipser's grammar and lexicon (see Diagram 3):

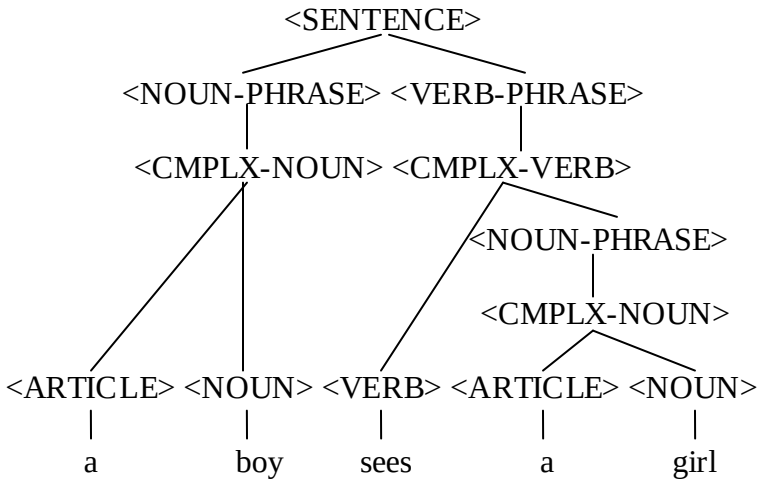

Diagram 5: Sipser CFG Derivation Tree

In this example, the root of the tree would be the <SENTENCE> node at the top, the leaves of the tree are the original input, and the branches are the lines connecting the tree's levels. Both the top-down and bottom-up parsing strategies will create the illustrated tree, although they do so using different methods. In a top-down parse strategy, the root node, <SENTENCE> is generated first and the leaves, corresponding with the words of the input, are generated last, essentially moving from the top of the tree diagram

down to the bottom. In a bottom-up parse strategy, the words of the sentence are the starting nodes and rules are applied until <SENTENCE> is generated.

Top-down parsing "starts with the [root] symbol and searches through different ways to rewrite the symbols until the input sentence is generated, or until all possibilities have been explored" (Allen 43). This leads to many dead-ends, as every possible sentence format is generated until it can be determined whether the derivation leads to the desired leaves, thus generating numerous trees that are inconsistent with the input (Jurafsky 363). However, most programs that implement a top-down parsing strategy include some way to check that the current parse is proceeding on track, but this checking is not included by default. The process of top-down parsing is detailed below:

1. Assume the desired sentence can be generated by the root symbol. In other words, assume the input is a valid sentence consistent with the example grammar
2. Expand the root symbol, creating a new derivation for each option, possibly checking the accessible nodes against the desired result to ensure continued relevance. What a symbol can be expanded to is determined by the rules in the grammar: if a rule contains the symbol on the right of the arrow, then the symbol can be expanded to the item(s) on the left of the arrow.
3. Expand the newly-added constituents using the same method as the expansion of the root node (again, possibly down accessible moves by looking ahead in the parse).

4. Continue expanding constituents until part-of-speech categories are created at the bottom of the tree, and then search the lexicon under the part-of-speech categories for the associated input words. Remember that, since the number of leaves generated by a valid tree should be equal to the number of words, each word should correspond to a part-of-speech category.
5. Reject any tree whose leaves do not match the words from the input. The remaining tree(s) will be the valid derivation(s). (Jurafsky 360)

These steps help illustrate how repetitive the parsing process can be: constituents are expanded until the lexicon has to be employed. Bottom-up parsing proceeds in almost the opposite direction as the instructions above. In bottom-up parsing, the program will "start with the words in the sentence and use the rewrite rules [(i.e. grammar)] backward to reduce the sequence of symbols until it consists solely of [the root]" (Allen 43). This technique, again, is not fast due to the number of dead-end derivations that must be tried. However, there are cases when one parsing strategy is better than the other and there are ways to modify these strategies to generate fewer trees and thus require fewer resources, but the basic techniques and goals remain the same.

2.7. Ambiguity

Assuming a user has created a reasonably-detailed grammar and lexicon and wishes to begin parsing sentences, the user will soon discover one of the main difficulties with working with natural language processing: ambiguity. Ambiguity arises when "there are multiple alternative linguistic structures [i.e. parses] that can be built" for a single input (Jurafsky 4). The wide-reaching effects of ambiguity surprised many researchers when it was first recognized. As Russell noted, "almost every utterance is highly ambiguous,

even though the alternative interpretations might not be apparent to a native speaker" (905). Russell continues to point out that the extent to which ambiguity is present in our daily conversations was not realized until the 1960s when researchers began using computers to analyze natural language. As a native speaker can use a sentence's context and a speaker's inflection to help determine a sentence's true meaning, a computer must rely on quantitative data that often cannot take such elements into account. Thus, not only does ambiguity exist, but it is present in multiple forms: lexical ambiguity (dealing with individual words), syntactic ambiguity (dealing with phrases), and semantic ambiguity (dealing with meaning).

Lexical ambiguity arises when "a word has more than one meaning", often even transcending part-of-speech categories; a popular example is the word "still" which can be an adjective ("still water"), noun ("photographic still"), adverb ("Be still!"), verb ("to still the tumult"), and even a conjunction ("It was late, still I walked."). Syntactic ambiguity is caused by "a phrase that has multiple parses" and semantic ambiguity is present in sentences where multiple parses lead to different interpretations of the original sentence's meaning (Jurafsky 905). As semantic ambiguity is often caused by lexical and syntactic ambiguity, it is not uncommon to find multiple forms of ambiguity in the same sentence. For example, one of the sentences generated by the Sipser grammar can be identified as both syntactically and semantically ambiguous: "the boy touches the girl with a flower" (see Diagram 3). Semantically, it can be interpreted as either the boy used a flower to touch the girl, or that the boy touched a girl who was holding a flower. This ambiguous sentence can also be represented by the two trees below:

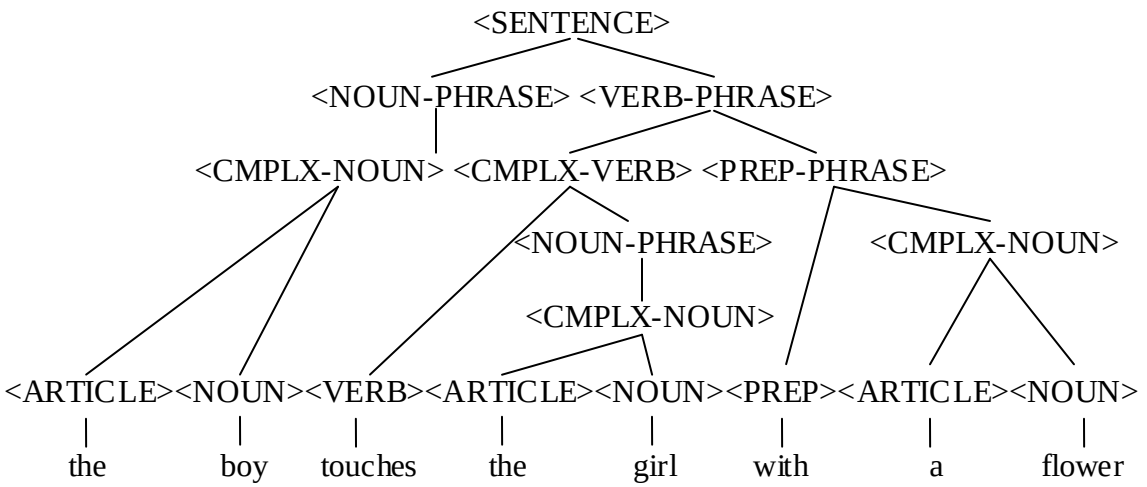

Diagram 6: Sipser CFG Ambiguous Derivation Tree #1

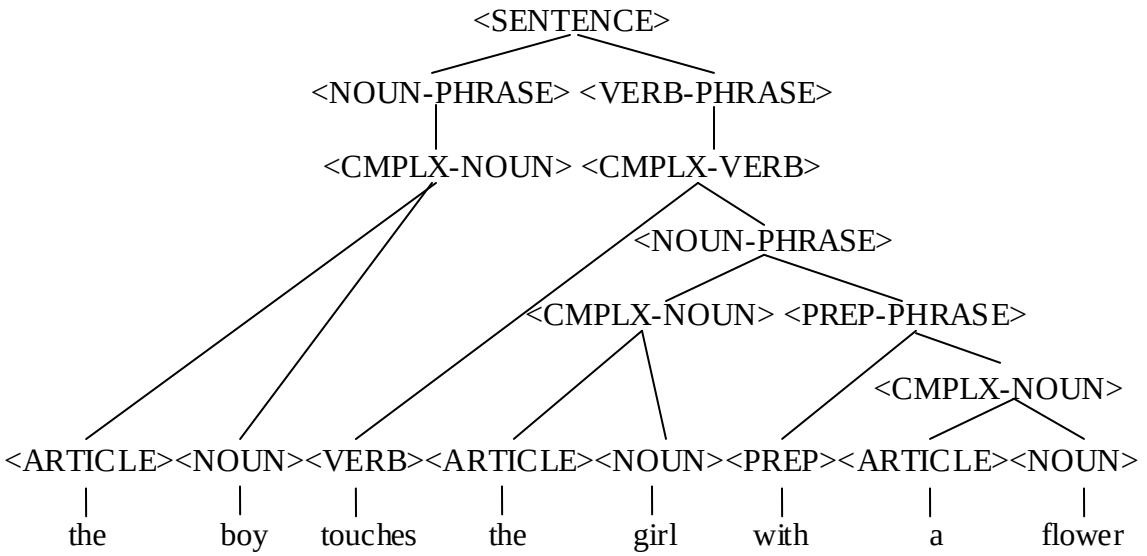

Diagram 7: Sipser CFG Ambiguous Derivation Tree #2

In the first tree, the prepositional phrase “with a flower” adjoins the verb phrase, thus identifying the boy as using the flower to touch the girl. In the second derivation tree, the prepositional phrase is adjoining the noun phrase “the girl”, thus identifying the girl of the sentence as possessing a flower. Both these interpretations are valid within the given grammar and lexicon and, although they carry different trees and, ultimately, different

meanings, it would be impossible to choose the "correct" meaning from the information present.

The problem of ambiguity is far-reaching and complex and a great many approaches have been taken toward its resolution. Jurafsky begins his book by dividing the field of speech and language processing into six focuses: phonetics and phonology, morphology, syntax, semantics, pragmatics, and discourse. After introducing ambiguity, he makes the claim that "a perhaps surprising fact about the six categories of linguistic knowledge is that most or all tasks in speech and language processing can be viewed as resolving ambiguity at one of these levels" (Jurafsky 4). Part-of-speech taggers recognize ambiguity and are often equipped with a way to record different tags for each individual word. Parsers sometimes employ charts that are able to record which rules are applied in order to keep track of all derivations without repetition. Using a chart, it becomes obvious if more than one path results in the desired state since the moves are compacted and easily viewable. There are multiple other methods that can be applied to reduce ambiguity in a sentence or to identify the most likely desired interpretation, but these still can rarely compare to the accuracy of a rational person's interpretation of the same input.

3.Project

3.1. The Plan

As mentioned previously, the intention of devising this project was to create a "sentence diagrammer" (i.e. sentence parser) that accurately recognizes and visualizes ambiguity in a given sentence. In turn, the project would demonstrate how natural language could be represented in computer code, as well as show the difficulties inherent in doing so. Originally, the project was to cover such grammar-dependent topics as syntax, ambiguity, and subject-verb agreement, but these goals changed once the scope of this undertaking became apparent.

Although this project required a great deal of background research, this project was not designed to recreate other researchers' work. The main focus in research was to accumulate the background knowledge about the field of computational linguistics and become acquainted with the basic history and principles of the field. Thus, the goal of this project was to explore the practical applications of the foundational topics of computational linguistics, rather than produce completely innovative thought or to reproduce foundational work.

3.2. Subject-Verb Identifier

The first technique applied in pursuit of a sentence diagrammer was to pare the task down to identifying the subject and verb first, then extrapolating the other words in the input from those words' placement. The identification process would compare the words to a list of nouns and a list of verbs to identify which word could be grouped under which part-of-speech. However, this technique requires a great deal of searching and checking, but without a great deal of reasoning. For example, in sentences that begin with a verb and contain an object ("Eat your vegetables"), this system could incorrectly identify the object as the subject. To remedy this, a restriction was placed on the sentence to search for the verb first, then search for the subject only in the words which proceeded the verb. But, when ambiguity was introduced, this technique quickly failed on ambiguous noun/verb inputs like "the dove dove."

3.3. Sentence-Based Parser

After recognizing that identifying a subject and verb first was not going to be a practical search method, but not wanting to delve into more complex parts-of-speech immediately, the program was modified to check the length of the sentence and guess the word order from that. For example, if the sentence was only one word long, in order for the sentence to be grammatically valid, the one word had to be a verb (e.g. "Eat"). Two word sentences were most likely either a subject and a verb (e.g. "I eat"), or a verb and an object (e.g. "Eat vegetables"), but could also be a verb and adverb (e.g. "Eat well"). A three-word sentence could be an article or an adjective accompanied by a subject and a verb, or it could be a verb followed by an article or adjective and an object, or it could be

a subject, a verb, and an object strung together, and so on, eventually going through all possible permutations of valid three-word sentence grammars. Of course, this would not be a practical solution for large sentences, but it did help organize some of the basic structure of sentences and highlight some patterns in sentence structure that were helpful in my next version.

3.4. Move Identification

After two trial permutations of unsuccessful grammars, I began drawing diagrams showing various possible permutations of valid English sentences. I have duplicated an example below:

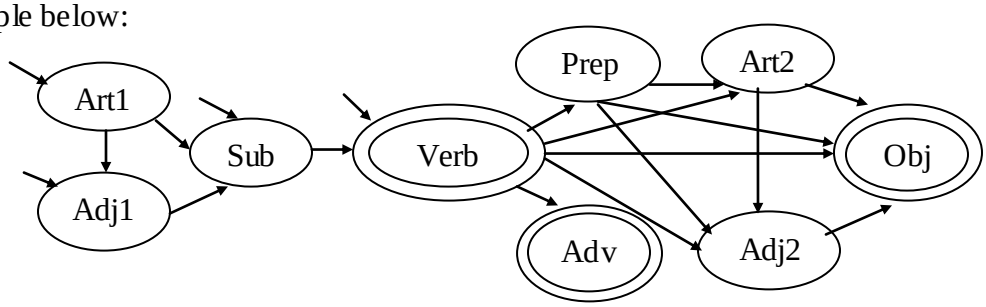

Diagram 8: Finite-State Automata-like Diagram

In the illustration, the parts-of-speech on which the sentence can end and be called valid are circled twice, and the parts-of-speech which can begin a sentence have a "floating arrow" pointing to them that is not derived from a state. All other arrows illustrate possible paths to or from a node. For example, in the illustration above, you can reach the "Prep" (preposition) node from the "Verb" node, and then you can travel to the "Art2" (article), "Adj2" (adjective), or "Obj" (object) nodes.

From making this and other similar diagrams, I quickly noticed that, even though noun phrases and verb phrases are not clearly defined in these graphs, there seems to be an order inherent in the parts-of-speech themselves due to their relationships within

phrases: subjects are nouns that occur before verbs and outside prepositional phrases, if articles and adjectives are present, a noun must appear soon, and other such properties of sentences. From these graphs, I created a small list of possible moves as a base for a context-free grammar:

```
Start   > Art1 or Adj1 or Sub or Verb
Art1    > Adj1 or Sub
Adj1    > Sub
Sub     > Verb
Verb    > Adv or Art2 or Adj2 or Obj or Prep or End
Prep    > Art2 or Adj2 or Obj
Art2    > Adj2 or Obj
Adj2    > Obj
Adv     > End
Obj     > End
```

Diagram 9: Test Moves List based on Diagram 8

In order to modify the above schema into a valid and accurate representation of English, some modifications need to be made to the list's structure. Some necessary connections are not present: a noun can lead to a preposition if there is a prepositional phrase referring to the subject of the sentence, the adverb state can be reached from a noun, and lists or loops of words are completely ignored. Also, there is no linguistic distinction between "Sub" (subject) and "Obj" (Object), nor between "Art1" and "Art2" or "Adj1" and "Adj2". By combining these states into "Noun", "Art", and "Adj", respectively, we can reduce the amount of space reserved for the lexicon drastically. Taking these modifications into account, we are able to create a more complete list of moves similar to the one detailed below:

Start > Art or Adj or Noun or Verb
Art > Adj or Noun
Adj > Adj or Noun
Noun > Verb or Prep or Adv (or End)
Verb > Adv or Art or Adj or Noun or Prep or Verb or Part or End
Prep > Art or Adj or Noun
Adv > Adj or Verb or Noun or End
Part > Verb

Diagram 10: Final Moves List

A couple notable changes exist in this list that did not occur in the original test list. First, as the Sub and Obj classifications have been combined into a single Noun classification, the End state is separated by parentheses to indicate that that state can only be reached through the previously-defined Obj state and not the Sub state. The method of making this distinction is discussed later. Also, one might observe the addition of the "Part" state in the new grammar, standing for "particle". This state was created to hold such words as infinitive-case "to" (as opposed to the prepositional-case "to"). A particle is defined as a "word that resembles a preposition or an adverb, and that often combines with a verb" (Jurafsky 292). This part-of-speech category was avoided earlier in this paper because of its general obscurity and its lack of concrete definition: some words are particles in some cases and adverbs or prepositions in others and it can be difficult to distinguish a difference when evaluating a sentence word-by-word. However, it is a necessary category as it allows the program to distinguish the prepositional phrase "to the house" from the verb phrase "to eat".

3.5. Context-Free Grammar

In order to translate the list of moves into a proper CFG (context-free grammar), one has to realize that, rather than dealing with moves, a CFG replaces the left-side non-terminal with the selected items on the right. As a CFG must be comprised of "a set of rules or productions, each of which expresses the ways that symbols of the language can be grouped and ordered together, and a lexicon of words and symbols" (Jurafsky 327), the start state will be <Sentence>, like in Sipser's grammar, and the earlier definitions of "→" and "|" remain the same. When the sentence is complete, the <End> state places a period ("."), as identified by new the <Period> state. In the example below, the terms ending with C represent the constituents where the base terms are terminal categories linked with specific terms in the lexicon (not included for simplicity's sake). Thus, the <ArtC> (constituent article) non-terminal produces the <Art> (article category) terminal and either the <AdjC> (constituent adjective) or <NounC> (constituent noun) non-terminals.

<Sentence> → <ArtC> | <AdjC> | <NounC> | <VerbC>
<ArtC> → <Art><AdjC> | <Art><NounC>
<AdjC> → <Adj><AdjC> | <Adj><NounC>
<NounC> → <Noun><VerbC> | <Noun><PrepC> |
<Noun><AdvC> | <Noun><End>
<VerbC> → <Verb><VerbC>| <Verb><AdvC> | <Verb><ArtC> |
<Verb><AdjC> | <Verb><NounC> | <Verb><PrepC>|
<Verb><PartC> | <Verb><End>
<PrepC> → <Prep><ArtC> | <Prep><AdjC> | <Prep><NounC>
<AdvC> → <Adv><AdjC> | <Adv><Verb> | <Adv><Noun> |
<Adv><End>
<PartC> → <Part><VerbC>
<End> → <Period>

Diagram 11: Context-Free Grammar

The rules from the grammar help determine under which categories in the lexicon should the program search for the desired word. The lexicon is represented in this program by text files associated with each part-of-speech category. These files contain short lists of words that are members of the desired category: due to memory and time constraints, these lists are currently unorganized stubs. When the program wishes to check whether a word from the input is included in a specific part-of-speech category, a testing command is sent which compares all words within the category list with the desired word and returns a Boolean true variable if it is found. Thus, the word would be identified as belonging to that category.

Since the phrase structures were removed from the grammar, but are still important linguistic structures, they are added back in as each word's part-of-speech is determined. For example, if a word is identified as an article, adjective, or noun, a noun phrase is initialized since those three part-of-speech categories are the elements that comprise a noun phrase. The initial noun phrase closes when the verb phrase begins, but the endings of noun phrases within the verb phrase or within prepositional phrases can sometimes be ambiguous. Prepositional phrases are bounded by the noun phrase they contain: they are created when a preposition is identified and closed after the included noun phrase completes. However, a noun phrase within a prepositional phrase can also contain a prepositional phrase, thus making the identification of a prepositional phrase's ending sometimes ambiguous as well. Verb phrases are very similar to prepositional phrases as they are created when a verb or adverb is identified and closed when all included constituents (prepositional phrases and noun phrases) have finished, often not until the end of the sentence.

The parsing strategy used to identify the constituents and parts-of-speech in this project is a bit different than the top-down and bottom-up strategies introduced earlier in this paper. As the analysis begins with the input and the identification of the words' parts-of-speech rather than an analysis of possible constituent organizations, I suppose this process is closest to the bottom-up parsing strategy. However, our measure of a successful parse is that all words from the input have been able to be identified based on the previous words' categorizations, which is not equivalent to the bottom-up parsing goal of forming the start state through the repeated applications of constituent-forming rules. In classifying the parsing strategy used in this program, it would be found most similar to a bottom-up depth-first search, where the initial state is the uncategorized input and, with each new word analyzed, the program attempts to extract as much information as possible out of the possible combinations of the new information and the previously-categorized words.

3.6. The Process

Once the program is started, it will open a window through which all interactions with the user will occur. The window and all related actions are managed through the Sentence Diagrammer class. On this window is a text box for the user to type an input sentence and an "Enter" button which signals the program to retrieve the input. All spaces and punctuation are then removed from the input and all upper-case characters are converted to lower-case. This modification is done to avoid duplicate entries in the lexicon for multiple forms of the same word. After the sentence is edited, Sentence Diagrammer creates an array with places each word in its own cell and sends the array to Organizer. Organizer initializes the first Parser and begins progressing through the sentence word-

by-word, identifying the parts of speech. As the Parser has just been initialized, Organizer is going to search through all possible part-of-speech categories that can be accessed from the start state. From the grammar defined in Section 3.5, we know the possible constituents are <ArtC>, <AdjC>, <NounC>, or <VerbC>. Organizer calls an identifier function in each of these constituent classes to check whether the current word is present in their dictionaries. Once one of the classes returns true (the current word is in their class), Organizer sets a variable in Parser that records the word's class. Once all allowable classes have been checked, organizer then initializes and ends all appropriate phrases and adds the word's classification to the current Parser. This repeats until the parser reaches the end of the sentence.

An example parse path is shown in the diagram below. This path illustrates the searches done by the program for the sentence "I am sitting on a chair." Each node shown is checked to see if the current word is present, but only the states with arrows outward are recorded in the Parser. Thus, from the Prep node, the program checks the Art, Adj, and Noun nodes, but only finds the desired word in Art, so it only checks the nodes accessible from the Art node.

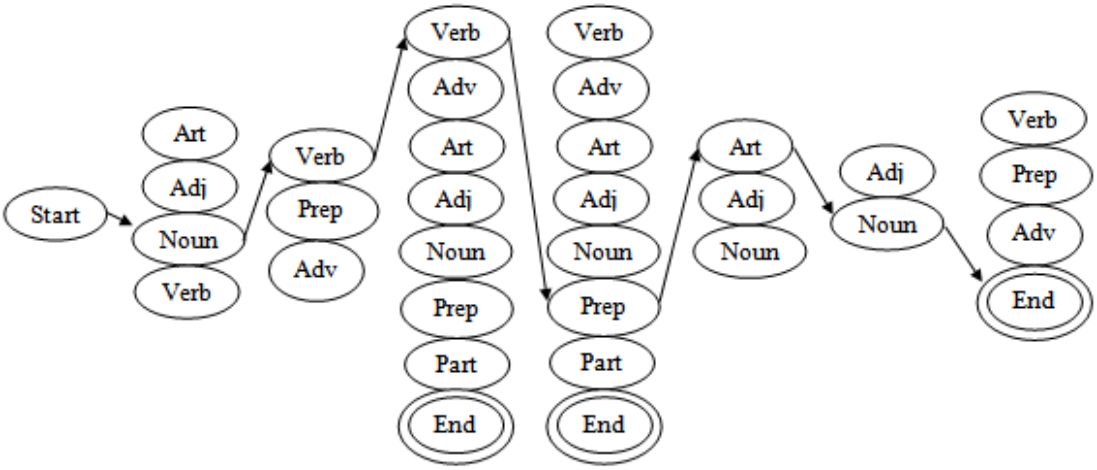

Diagram 12: Parse Path for "I am sitting on a chair."

A notable feature of program, as illustrated in the parse path, is that the End node is not accessible from the first Noun node, but it is a valid move from the second Noun node. This is because the program recognized that a verb was already identified and, thus, the second noun could not be the subject of the sentence because it was within the verb phrase. From this, the program identified the second noun as an object and enabled the move to end the sentence. This functionality is also used to identify understood-you sentences: command sentences that begin with a verb and do not contain a subject. If a verb is identified before a noun is found, the program will place a "(you)" before the verb. This allows the input to hold to the stipulation that all sentences must be comprised of a separate noun phrase and verb phrase, as mentioned in Section 2.4.

When the words identified is equal to the length of the sentence, the Parser is set to <End>, which prints a period in the parser, closes all phrases, and returns the parser to Sentence Diagrammer. This class then goes through the parse and separates the phrasal identifications from the lexical identifications and stores them each as a long string. The

size of the strings are adjusted to correspond with the size of the sentence, so items which occur due to the same input are displayed similarly.

Once all constituents and parts-of-speech have been separated, the resulting parse is displayed underneath the user's original input, as shown below. The parse's diagram ultimately consists of the phrases on the top row, surrounding the parts-of-speech classifications which sit above their associated input words, all displayed in a user-friendly, easy-to-read format.

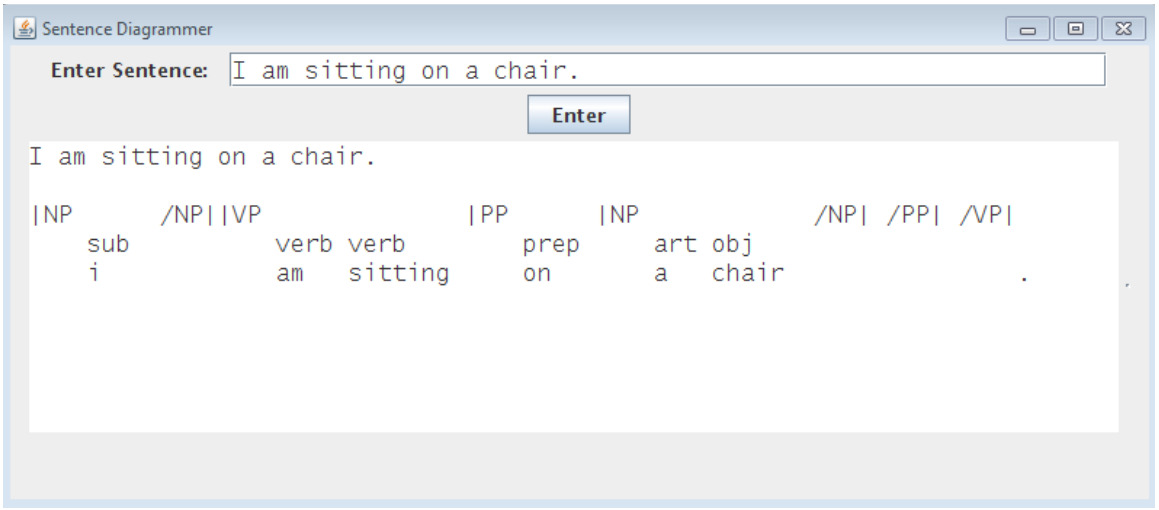

Diagram 13: Sentence Diagrammer

3.7. Lexical and Syntactic Ambiguity

Occasions arise while parsing when a word is identified as belonging to more than one part-of-speech or a phrase could have more than one relation. In these cases, the program will create a new parse, using the previous parse's derivations up until the deviation point, and then adding the valid possible moves to the end of each of the parses. The resulting parses have no subsequent interaction and are, in effect, treated as two different inputs. When the entirety of the input has been identified and parsed, the program displays all resulting parses.

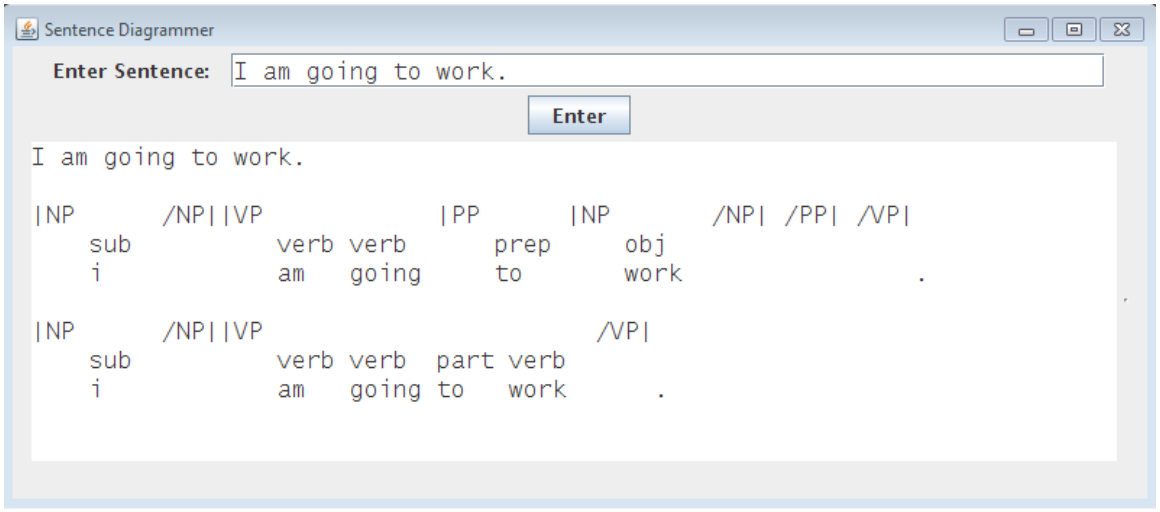

Diagram 14: Sentence Diagrammer – Lexically Ambiguous Input

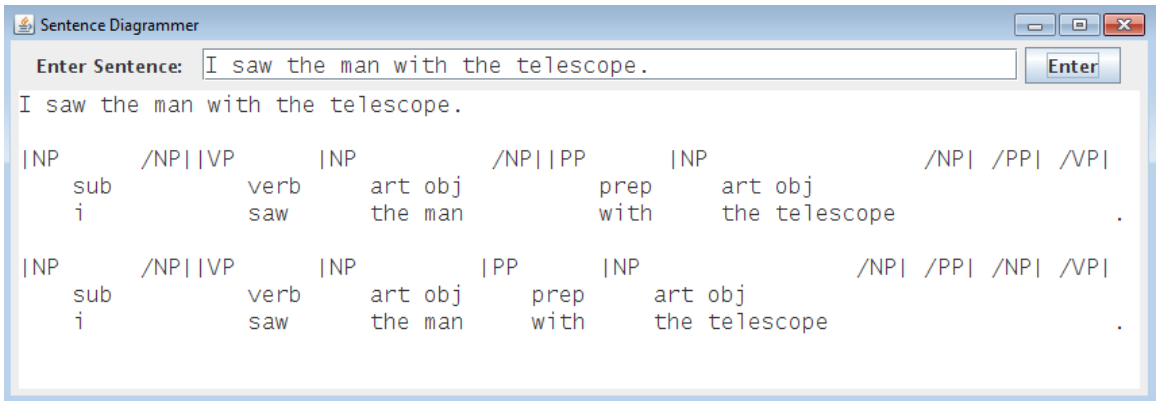

Diagram 15: Sentence Diagrammer – Syntactically Ambiguous Input

Of course, sometimes a new parse is created but, in future steps, it is found that the parse does not fit with the input. If the program finds an input word that doesn't appear in any of the categories linked with the previous word, the parse is rejected and deleted. By deleting a parse, it will no longer appear on the final screen. However, this scenario can also occur if the word from the input is simply not categorized or not categorized correctly. The command line version of the program prints out a warning about what word was unable to be identified and suggests that the user add the word to the list files, if necessary. However, in my testing, I have found that it is more often the

case that an ambiguous sentence is incorrectly interpreted than that the dictionary is missing an input.

4.Conclusion

4.1. Future Modifications

There are many things that could easily be added or improved in this project, which is part of the reason why I wish to release it as an open-source program. By allowing other people to add to my project, I think it can grow into a much bigger application than I would be able to create myself.

Some of the possible options for expansion in this project would be to add functionality for parsing questions, negatives, conjunctions, contractions, and other words and sentence arrangements that are relatively common but not as essential as the parts-of-speech and basic sentences demonstrated herein. It would also probably be wise to expand and sort the part-of-speech text files to reduce search time, as well as create functionality to allow the user to add an unknown word to the appropriate list file directly from the interface. The functionality present in the program now is almost capable of identifying words to be added, but it also has the trend of declaring a word as undefined if it has not been able to find it in the current path because of an incorrect parse turn.

Future work could also include implementation of probability factors associated with part-of-speech tagging. This would help the program more quickly identify which elements to check first and ultimately speed up the parsingprocess. Along the same lines,

it would be possible to calculate the probability of a parse being the intended interpretation of an ambiguous sentence by comparing the probabilities of each move in all possible parses and displaying the only the highest-probability result.

There are also ways in which the program can be made more accurate in its parsing and ambiguity identification. Currently, the program assumes the input is a properly-formatted English sentence. If this is not the case, the program will still run and sometimes ends up finding a completely nonsensical parsing in English that, nonetheless, is perfectly valid in the created grammar. If the program is made to check subject-verb agreement, identify the appropriate verb conjugation, and separate out auxiliary verbs and pronouns from their respective verb and noun morass, the program will become a great deal more accurate and consistent with the English language's actual rules.

4.2. Applications

In its current form, this project is unlikely to be of technical use to anyone except as a tool demonstrating the complexities and difficulties inherent in natural language processing. However, if some of more work is performed on it, it could very easily end up being a program used in school to teach students how to diagram sentences, or in English as a Second Language programs to help demonstrate how the elements of grammar are combined to create sentences. If expanded, this research would also be useful in checking for grammatical errors in word-processing software, or in providing the computer functionality to identify the subject, object, and action in a human-computer interaction study. Along the same lines, this code could be used in speech synthesis to identify a word's part-of-speech and thus determine the appropriate pronunciation. Nonetheless, in its current form, this program currently serves its intended purpose of being an illustration of the inherently ambiguous nature of the English language and the difficulties of translating natural language into a computer-programmable form.

5.Bibliography

Allen, James. *Natural Language Understanding*. 2d ed. Redwood City, California: The Benjamin/Cummings Publishing Company, Inc., 1995.

Brinton, Laurel J. *The Structure of Modern English: A Linguistic Introduction*. Amsterdam, The Netherlands: John Benjamins Publishing Co., 2000.

Jurafsky, Daniel and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Upper Saddle River, New Jersey: Prentice Hall, 2000.

Krulee, Gilbert K. *Computer Processing of Natural Language*. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

Russell, Stuart J. and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3d ed. Upper Saddle River, New Jersey: Prentice Hall, 2010.

Sipser, Michael. *Introduction to the Theory of Computation*. 2d ed. Boston: Course Technology, 2006.

6.Appendix A: Sample Test Sentences

Basic

- Eat green vegetables.
  - This sentence tests the "understood-you" case, where no subject is present.
- I am sitting on a chair.
  - This input tests phrase recognition and nested phrase order.
- The man in the garden is eating.
  - This input specifically tests nested noun phrases.
- I am going to eat.
  - This sentence tests the whether the incorrect interpretation of "to" as a preposition is deleted.
- I want to present the present.
  - This sentence checks that the proper interpretation of ambiguous words can be found. "Present" could be either a verb or noun, but this sentence ensures that the first instance be read as a verb and the second instance as a noun.

Ambiguous

- Students hate annoying professors.
  - This sentence tests lexical ambiguity as "annoying" should be identified as both a verb and an adjective.
- I am going to work.
  - The phrase "to work" can either be seen as part of the verb phrase ("work" as a verb) or a prepositional phrase ("work" as a noun).
- I saw the man with the telescope.
  - This sentence tests syntactic ambiguity with the prepositional phrase "with the telescope", which could refer to the verb "saw" or the noun phrase "the man".

7.Appendix B: Code Used

Note that this code is written in the Java programming language. To allow the program to run, the user must create classes for Article, Adjective, Adverb, Particle, and Preposition as well, as these have been left out due to space concerns. However, each of these classes contains the same code as the included Verb class (see Section 7.5) except for a new dictionary file location and some name changes.

7.1. SentenceDiagrammer

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class SentenceDiagrammer extends JFrame implements
ActionListener{

   private static final long serialVersionUID = 1L;

   int fieldLength = 75;

   JTextField textField;
   JTextArea uneditArea;
   JLabel enter;
   JButton btn;
   JScrollPane scrollPane;
   String input;
   String[] sentence;

   public SentenceDiagrammer(){
      JFrame frame = new JFrame("Sentence Diagrammer");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      createFrame(frame);
      frame.setSize(800, 350);
      frame.setVisible(true);
```

```
}

public void createFrame(JFrame frame){
   Container container = frame.getContentPane();
   container.setLayout(new FlowLayout());

   Font font1 = new Font("Lucida Sans Typewriter", Font.PLAIN, 16);
   Font font2 = new Font("Lucida Sans", Font.BOLD, 14);

   textField = new JTextField(fieldLength-15);
   textField.setFont(font1);
   enter = new JLabel("Enter Sentence:  ");
   enter.setFont(font2);
   btn = new JButton("Enter");
   btn.setFont(font2);
   uneditArea = new JTextArea(10,fieldLength);
   uneditArea.setEditable(false);
   uneditArea.setFont(font1);
   scrollPane = new JScrollPane(uneditArea);


   container.add(enter);
   container.add(textField);
   container.add(btn);
   container.add(uneditArea);
   container.add(scrollPane);

   btn.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent arg0) {
         startParsing();
      }
   });
}

public void startParsing() {
   input = textField.getText();
   uneditArea.setText(input+"\n"+"\n");
   sentence = editSentence(input);

   Organizer org = new Organizer(sentence);
   Parser parse = org.sentenceParse(org.parse, sentence);
   for(int j = 0; j <= parse.count; j++){
      String[] text = createDiagram(parse.parsing[j], sentence);
      for(int i = 0; i < 3; i++){
         uneditArea.append(text[i]+"\n");
      }
      uneditArea.append("\n");
   }

uneditArea.setCaretPosition(uneditArea.getDocument().getLength());
}

public String[] editSentence(String s){
   s = s.trim();
   s = s.toLowerCase();

   s = s.replaceAll("[\\p{Punct}]", " ");
```

```
      System.out.println("Input: " + s);
      return s.split(" ");
   }
   public String[] createDiagram(String[] parsing, String[] sentence){
      String[] diagram = new String[3];
      diagram[0] = "";
      diagram[1] = "";
      diagram[2] = "";

      int sentenceCounter = 0;

      for(int i = 0; parsing[i]!= null; i++){
         if(parsing[i].equals("NP")){
            diagram[0] = diagram[0].concat("|NP ");
         }
         else if(parsing[i].equals("/NP")){
            diagram[0] = diagram[0].concat(" /NP|");
         }
         else if(parsing[i].equals("VP")){
            diagram[0] = diagram[0].concat("|VP ");
         }
         else if(parsing[i].equals("/VP")){
            diagram[0] = diagram[0].concat(" /VP|");
         }
         else if(parsing[i].equals("PP")){
            diagram[0] = diagram[0].concat("|PP ");
         }
         else if(parsing[i].equals("/PP")){
            diagram[0] = diagram[0].concat(" /PP|");
         }

         else if(parsing[i].equals("sub")){
            diagram[1] = diagram[1].concat("sub ");
            if(Verb.isVerb(sentence[sentenceCounter])){
               diagram[2] = diagram[2].concat("(you) ");
            }
            else{
               diagram[2] =
diagram[2].concat(sentence[sentenceCounter]+" ");
               sentenceCounter = sentenceCounter+1;
            }
         }
         else if(parsing[i].equals("obj")){
            diagram[1] = diagram[1].concat("obj ");
            diagram[2] = diagram[2].concat(sentence[sentenceCounter]+"
");
            sentenceCounter = sentenceCounter+1;
         }
         else if(parsing[i].equals("verb")){
            diagram[1] = diagram[1].concat("verb ");
            diagram[2] = diagram[2].concat(sentence[sentenceCounter]+"
");
            sentenceCounter = sentenceCounter+1;
         }
         else if(parsing[i].equals("art")){
            diagram[1] = diagram[1].concat("art ");
```

```
            diagram[2] = diagram[2].concat(sentence[sentenceCounter]+"
");
            sentenceCounter = sentenceCounter+1;
         }
         else if(parsing[i].equals("adj")){
            diagram[1] = diagram[1].concat("adj ");
            diagram[2] = diagram[2].concat(sentence[sentenceCounter]+"
");
            sentenceCounter = sentenceCounter+1;
         }
         else if(parsing[i].equals("adv")){
            diagram[1] = diagram[1].concat("adv ");
            diagram[2] = diagram[2].concat(sentence[sentenceCounter]+"
");
            sentenceCounter = sentenceCounter+1;
         }
         else if(parsing[i].equals("prep")){
            diagram[1] = diagram[1].concat("prep ");
            diagram[2] = diagram[2].concat(sentence[sentenceCounter]+"
");
            sentenceCounter = sentenceCounter+1;
         }
         else if(parsing[i].equals("part")){
            diagram[1] = diagram[1].concat("part ");
            diagram[2] = diagram[2].concat(sentence[sentenceCounter]+"
");
            sentenceCounter = sentenceCounter+1;
         }
         else if(parsing[i].equals(".")){
            diagram[2] = diagram[2].concat(". ");
            sentenceCounter = sentenceCounter+1;
         }

         int zero = diagram[0].length();
         int one = diagram[1].length();
         int two = diagram[2].length();

         int maxLength = zero;
         if(one > maxLength){
            maxLength = one;
         }
         if(two > maxLength){
            maxLength = two;
         }

         while(diagram[0].length() < maxLength){
            diagram[0] = diagram[0].concat(" ");
         }
         while(diagram[1].length() < maxLength){
            diagram[1] = diagram[1].concat(" ");
         }
         while(diagram[2].length() < maxLength){
            diagram[2] = diagram[2].concat(" ");
         }

      }
```

```
      return diagram;
   }

   @Override
   public void actionPerformed(ActionEvent e) {}

}
```

7.2. Organizer

```
public class Organizer {

   Parser parse;

   public Organizer (String[] sentence)
   {
      parse = new Parser (sentence);
   }
   public Parser sentenceParse(Parser parse, String[] sentence){

      /* start > article, adjective, noun, verb
       * article > adjective, noun
       * adjective > adjective, noun
       * noun > verb, preposition, adverb, end
       * verb > adverb, article, adjective, noun, preposition, verb,
particle, end
       * preposition > article, adjective, noun
       * adverb > adjective, noun, verb, end
       * particle > verb
       */

      int wordCount = 0;
      int parseCounter = 0;

      for(wordCount = 0; wordCount <= sentence.length; wordCount++){

         for(parseCounter = 0; parseCounter <= parse.count;
parseCounter++){
            if(parse.prevWord[parseCounter] == 1){
               //start > article, adjective, noun, verb
               parse.prevWord[parseCounter] = 0;
               if(Noun.isNoun(sentence[wordCount])){
                  parseCounter = newParseTest(parseCounter);
                  parse.prevWord[parseCounter] = 4;
               }
               if(Adjective.isAdj(sentence[wordCount])){
                  parseCounter = newParseTest(parseCounter);
                  parse.prevWord[parseCounter] = 3;
               }
               if(Article.isArt(sentence[wordCount])){
                  parseCounter = newParseTest(parseCounter);
                  parse.prevWord[parseCounter] = 2;
               }
```

```
                if(Verb.isVerb(sentence[wordCount])){
                    parseCounter = newParseTest(parseCounter);
                    parse.prevWord[parseCounter] = 5;
                }
            }
            else if(parse.prevWord[parseCounter] == 2){
                //article > adjective, noun
                parse.prevWord[parseCounter] = 0;
                if(Noun.isNoun(sentence[wordCount])){
                    parseCounter = newParseTest(parseCounter);
                    parse.prevWord[parseCounter] = 4;
                }
                if(Adjective.isAdj(sentence[wordCount])){
                    parseCounter = newParseTest(parseCounter);
                    parse.prevWord[parseCounter] = 3;
                }
            }
            else if(parse.prevWord[parseCounter] == 3){
                //adjective > adjective, noun
                parse.prevWord[parseCounter] = 0;
                if(Noun.isNoun(sentence[wordCount])){
                    parseCounter = newParseTest(parseCounter);
                    parse.prevWord[parseCounter] = 4;
                }
                if(Adjective.isAdj(sentence[wordCount])){
                    parseCounter = newParseTest(parseCounter);
                    parse.prevWord[parseCounter] = 3;
                }
            }
            else if(parse.prevWord[parseCounter] == 4){
                //noun > verb, preposition, adverb, end
                parse.prevWord[parseCounter] = 0;
                if(wordCount == sentence.length){
                    if(parse.hasBeenVerb[parseCounter]){
                        parse.prevWord[parseCounter] = 9;
                    }
                }
                else{
                    if(Verb.isVerb(sentence[wordCount])){
                        parseCounter = newParseTest(parseCounter);
                        parse.prevWord[parseCounter] = 5;
                    }
                    if(Preposition.isPrep(sentence[wordCount])){
                        parseCounter = newParseTest(parseCounter);
                        parse.prevWord[parseCounter] = 6;
                    }
                    if(Adverb.isAdv(sentence[wordCount])){
                        parseCounter = newParseTest(parseCounter);
                        parse.prevWord[parseCounter] = 7;
                    }
                }
            }
            else if(parse.prevWord[parseCounter] == 5){
                //verb > adverb, article, adjective, noun, preposition,
verb, particle, end
                parse.prevWord[parseCounter] = 0;
                if(wordCount == sentence.length){
```

```
            if(parse.hasBeenVerb[parseCounter]){
               parse.prevWord[parseCounter] = 9;
            }
         }
         else{
            if(Adverb.isAdv(sentence[wordCount])){

               parseCounter = newParseTest(parseCounter);
               parse.prevWord[parseCounter] = 7;
            }
            if(Article.isArt(sentence[wordCount])){
               parseCounter = newParseTest(parseCounter);
               parse.prevWord[parseCounter] = 2;
            }
            if(Adjective.isAdj(sentence[wordCount])){
               parseCounter = newParseTest(parseCounter);
               parse.prevWord[parseCounter] = 3;
            }
            if(Noun.isNoun(sentence[wordCount])){
               parseCounter = newParseTest(parseCounter);
               parse.prevWord[parseCounter] = 4;
            }
            if(Preposition.isPrep(sentence[wordCount])){
               parseCounter = newParseTest(parseCounter);
               parse.prevWord[parseCounter] = 6;
            }
            if(Verb.isVerb(sentence[wordCount])){
               parseCounter = newParseTest(parseCounter);
               parse.prevWord[parseCounter] = 5;
            }
            if(Particle.isPart(sentence[wordCount])){
               parseCounter = newParseTest(parseCounter);
               parse.prevWord[parseCounter] = 8;
            }
         }
      }
      else if(parse.prevWord[parseCounter] == 6){
         //preposition > article, adjective, noun
         parse.prevWord[parseCounter] = 0;
         if(Noun.isNoun(sentence[wordCount])){
            parseCounter = newParseTest(parseCounter);
            parse.prevWord[parseCounter] = 4;
         }
         if(Adjective.isAdj(sentence[wordCount])){
            parseCounter = newParseTest(parseCounter);
            parse.prevWord[parseCounter] = 3;
         }
         if(Article.isArt(sentence[wordCount])){
            parseCounter = newParseTest(parseCounter);
            parse.prevWord[parseCounter] = 2;
         }
      }
      else if(parse.prevWord[parseCounter] == 7){
         //adverb > adjective, noun, verb, end
         parse.prevWord[parseCounter] = 0;
         if(wordCount == sentence.length){
            if(parse.hasBeenVerb[parseCounter]){
```

```
                        parse.prevWord[parseCounter] = 9;
                    }
                }
                else{
                    if(Adjective.isAdj(sentence[wordCount])){
                        parseCounter = newParseTest(parseCounter);
                        parse.prevWord[parseCounter] = 3;
                    }
                    if(Noun.isNoun(sentence[wordCount])){
                        parseCounter = newParseTest(parseCounter);
                        parse.prevWord[parseCounter] = 4;
                    }
                    if(Verb.isVerb(sentence[wordCount])){
                        parseCounter = newParseTest(parseCounter);
                        parse.prevWord[parseCounter] = 5;
                    }
                }
            }
            else if(parse.prevWord[parseCounter] == 8){
                //particle > verb
                parse.prevWord[parseCounter] = 0;
                if(Verb.isVerb(sentence[wordCount])){
                    parseCounter = newParseTest(parseCounter);
                    parse.prevWord[parseCounter] = 5;
                }
            }

            /* Previous Word Numbers
             * 1 = start
             * 2 = article
             * 3 = adjective
             * 4 = noun
             * 5 = verb
             * 6 = preposition
             * 7 = adverb
             * 8 = particle ("to" as in "to eat")
             * 9 = end
             */
        }
        for(parseCounter = 0; parseCounter <= parse.count;
parseCounter++){
            if(parse.prevWord[parseCounter] == 0){
                System.out.println("Word not identified. If necessary,
add '"+sentence[wordCount]+"' to dictionary.");
                parse.deleteParse(parseCounter);
            }
            else if(parse.prevWord[parseCounter] == 1){
                System.out.println("Problem: Should have identified move
from start state.");
            }

            else if(parse.prevWord[parseCounter] == 2){
                if(parse.PP[parseCounter])
                    parse.startPrepNounPhrase(parseCounter);
                else
                    parse.startNounPhrase(parseCounter);
```

```
                new Article (sentence[wordCount], parse, parseCounter);
             }
             else if(parse.prevWord[parseCounter] == 3){
                if(parse.PP[parseCounter])
                   parse.startPrepNounPhrase(parseCounter);
                else
                   parse.startNounPhrase(parseCounter);

                new Adjective (sentence[wordCount], parse,
parseCounter);
             }
             else if(parse.prevWord[parseCounter] == 4){
                if(parse.PP[parseCounter])
                   parse.startPrepNounPhrase(parseCounter);
                else
                   parse.startNounPhrase(parseCounter);

                new Noun (sentence[wordCount], parse, parseCounter);
             }
             else if(parse.prevWord[parseCounter] == 5){
                if(!parse.hasSubject[parseCounter]){
                   parse.startNounPhrase(parseCounter);
                }
                parse.endPrepNounPhrase(parseCounter);
                parse.endPrepPhrase(parseCounter);
                parse.endNounPhrase(parseCounter);
                parse.startVerbPhrase(parseCounter);
                parse.hasBeenVerb[parseCounter] = true;
                new Verb (sentence[wordCount], parse, parseCounter);
             }
             else if(parse.prevWord[parseCounter] == 6){
                if((parse.NP[parseCounter] || parse.PNP[parseCounter])
&& parse.VP[parseCounter]){
                   parse.createNewSentence(parseCounter);
                   parseCounter = parse.count;
                   parse.endNounPhrase(parseCounter-1);
                   parse.startPrepPhrase(parseCounter-1);
                   new Preposition (sentence[wordCount], parse,
parseCounter-1);
                }

                parse.startPrepPhrase(parseCounter);
                new Preposition (sentence[wordCount], parse,
parseCounter);
             }
             else if(parse.prevWord[parseCounter] == 7){
                parse.startVerbPhrase(parseCounter);
                new Adverb (sentence[wordCount], parse, parseCounter);
             }
             else if(parse.prevWord[parseCounter] == 8){
                new Particle (sentence[wordCount], parse, parseCounter);
             }
             else if(parse.prevWord[parseCounter] == 9){
                parse.endPrepNounPhrase(parseCounter);
                parse.endPrepPhrase(parseCounter);
                parse.endNounPhrase(parseCounter);
                parse.endVerbPhrase(parseCounter);
```

```
                parse.addToParsing(".", parseCounter);
                parse.printParser(parseCounter);
             }
          }
       }
       return parse;
    }
    public int newParseTest(int parseCounter){
       if(parse.prevWord[parseCounter] != 0){
          parse.createNewSentence(parseCounter);
          parseCounter = parse.count;
       }
       return parseCounter;
    }
}
```

7.3. Parser

```
public class Parser {
   int margin = 15;
   boolean[] NP = new boolean [margin];
   boolean[] VP = new boolean [margin];
   boolean[] PP = new boolean [margin];
   boolean[] PNP = new boolean [margin];
   boolean[] hasBeenVerb = new boolean [margin];
   boolean[] hasSubject = new boolean [margin];
   int[] prevWord = new int[margin]; // last word type
   int count; //number of parses
   int length; //number of words in sentence
   String[][] parsing;
   int[] parsingIndex = new int[margin];//number of items in parsing

   public Parser(String[] sentence){
      count = 0; //first parse
      NP[0] = false; //0 from count
      VP[0] = false;
      PP[0] = false;
      PNP[0] = false;
      hasBeenVerb[0] = false;
      hasSubject[0] = false;
      prevWord[0] = 1; //start sentence
      length = sentence.length;
      parsing = new String[margin][length+margin]; //number of parses x
number of items in each parse
      parsingIndex[0] = 0;
   }
   //copy sentence
   public void createNewSentence(int counter){
      if(counter + 1 < margin){
         count = count+1;
         NP[count] = NP[counter];
         VP[count] = VP[counter];
```

```
            PP[count] = PP[counter];
            PNP[count] = PNP[counter];
            hasBeenVerb[count] = hasBeenVerb[counter];
            hasSubject[count] = hasSubject[counter];
            prevWord[count] = prevWord[counter];
            for(int i = 0; i <= parsingIndex[counter]; i++){
               parsing[count][i] = parsing[counter][i];
            }
            parsingIndex[count] = parsingIndex[counter];
         }
         else{
            System.out.println("Margin not large enough to accomodate
another parse. Please remedy.");
            System.exit(0);
         }
      }
      public void deleteParse(int counter){

         if(count > 0){ //count == 0 means one parse
            if(counter == count){
               NP[count] = false;
               VP[count] = false;
               PP[count] = false;
               PNP[count] = false;
               hasBeenVerb[count] = false;
               hasSubject[count] = false;
               prevWord[count] = 1; //start sentence
               for(int i = 0; i <= parsingIndex[count]; i++){
                  parsing[count][i] = null;
               }
               parsingIndex[count] = 0;
            }
            else{
               NP[counter] = NP[count];
               VP[counter] = VP[count];
               PP[counter] = PP[count];
               PNP[counter] = PNP[count];
               hasBeenVerb[counter] = hasBeenVerb[count];
               hasSubject[counter] = hasSubject[count];
               prevWord[counter] = prevWord[count];
               for(int i = 0; i <= parsingIndex[count]; i++){
                  parsing[counter][i] = parsing[count][i];
               }
               parsingIndex[counter] = parsingIndex[count];
            }
            count = count-1;
         }
         else{ //count == 0 - clear parse
            count = 0;
            NP[0] = false;
            VP[0] = false;
            PP[0] = false;
            PNP[0] = false;
            hasBeenVerb[0] = false;
            hasSubject[0] = false;
            prevWord[0] = 1; //start sentence
            parsing = new String[margin][length+margin];
```

```
      parsingIndex[0] = 0;
   }
}

public void startNounPhrase(int i){
   if(NP[i] == false)
      addToParsing("NP", i);
   NP[i] = true;
}
public void endNounPhrase(int i){
   if(!hasSubject[i])
      new Noun ("you", this, i); //understood "you" case - no verb
   if(NP[i] == true)
      addToParsing("/NP", i);
   NP[i] = false;
}
public void startPrepNounPhrase(int i){
   if(PNP[i] == false)
      addToParsing("NP", i);
   PNP[i] = true;
}
public void endPrepNounPhrase(int i){
   if(PNP[i] == true)
      addToParsing("/NP", i);
   PNP[i] = false;
}
public void startVerbPhrase(int i){
   if(VP[i] == false)
      addToParsing("VP", i);
   VP[i] = true;
}
public void endVerbPhrase(int i){
   if(VP[i] == true)
      addToParsing("/VP", i);
   VP[i] = false;
}
public void startPrepPhrase(int i){
   if(PP[i] == false)
      addToParsing("PP", i);
   PP[i] = true;
}
public void endPrepPhrase(int i){
   if(PP[i] == true)
      addToParsing("/PP", i);
   PP[i] = false;
}
public void addToParsing(String x, int i){
   parsing[i][parsingIndex[i]] = x;
   parsingIndex[i] = parsingIndex[i]+1;
}
public void printParser(int parseNum){
   System.out.print(parseNum+": ");
   for(int i = 0; i < parsingIndex[parseNum]; i++){
      System.out.print(" "+parsing[parseNum][i]+" ");
   }
   System.out.println();
}
```

```
}
```

7.4. Noun

Please note that this function is slightly different than other part-of-speech tagging functions created in this project because of the distinction created between the subject of a sentence (“sub”) and the object (“obj”).

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;


public class Noun {
   public Noun(String noun, Parser p, int i) {
      if(!p.hasSubject[i]){
         p.addToParsing("sub", i);
         p.hasSubject[i] = true;
      }
      else
         p.addToParsing("obj", i);
   }

   public static boolean isNoun(String word){
      File nounList = new File ("C:\\NounList"); //change to Noun file
location
      try{
         FileInputStream nfis = new FileInputStream(nounList);
         BufferedReader nbr = new BufferedReader(new
InputStreamReader(nfis));
         String nextline = nbr.readLine();
         while(nextline != null){
            if(nextline.equals(word)){
               return true;
            }
            else{
               nextline = nbr.readLine();
            }
         }
      }
      catch (FileNotFoundException e){}
      catch (IOException e){}
      return false;
   }
}
```

7.5. Verb

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;


public class Verb {
   public Verb(String verb, Parser p, int i) {
      p.addToParsing("verb", i);
   }

   public static boolean isVerb(String word){
      File verbList = new File ("C:\\VerbList"); //change to Verb file
location
      try{
         FileInputStream vfis = new FileInputStream(verbList);
         BufferedReader vbr = new BufferedReader(new
InputStreamReader(vfis));
         String nextline = vbr.readLine();
         while(nextline != null){
            if(nextline.equals(word)){
               return true;
            }
            else{
               nextline = vbr.readLine();
            }
         }
      }
      catch (FileNotFoundException e){}
      catch (IOException e){}
      return false;
   }
}
```