Interactive Chart Parsing With Python

Kathryn Nichols

December 12, 2012

Abstract

Natural language processing often requires the building of syntactic structures from raw input, but this process can be enormously complex with large data. Chart parsing is a method that efficiently memoizes likely syntactic constituents, returning a complete summarization of sentence structure at the termination of the algorithm. This project discusses the linguistic components, data structures and algorithms used in chart parsing, then gives an overview of the Interactive Parser: an implementation of chart parsing that allows a user to input a grammar and lexicon at loading or run time, and will give a detailed explanation of each step in the algorithm, providing an interactive and informative parsing experience to the user.

1 Introduction

One of the very first steps in natural language processing (NLP) is mapping the structure of a sentence. This is an important component in question-and-answering, part-of-speech tagging, language modeling, semantic analysis, and numerous other advanced NLP techniques.

While much annotation used to be done by hand, modern computational linguistics provides the ability to parse automatically. Brute force methods of parsing tend to be extremely inefficient, as the basic algorithm must process all conceivable combinations of constituency. While the complexity will depend in part on the depth of the grammar (that is, how deep of a tree the rules allow us to build), the exploration of every combination across set of parts of speech T on a string of words N yields a lower complexity bound of $\Omega(|N|^{|T|})$, an intractable figure for most NLP applications.

A dynamic approach, however, allows the parser to keep a running list of only the viable constituents of a sentence, considerably reducing complexity. There are several versions of this technique, among them the Earley algorithm, CKY parsing and Kay's chart parsing (Jurafsky 2009). This project implements a version of the latter. In the following sections, I will give an overview of Kay's chart parsing, including the data structures required and the algorithm itself. Then I will introduce the current program, its basic requirements to run and special features, followed by a discussion of how I approached the implementation of chart parsing. Lastly, I will review the aims and success of the project overall and potential improvements.

2 Chart Parsing

There are several chart parsing algorithms, but the one implemented here is attributed to Kay (1982). The algorithm maintains a queue of *keys* (constituents), a list of *arcs* (possible constituents in the form of a grammar rule, with a symbol or index tracking the prescence of subconstituents), and a *chart*: a data structure that keeps records of a completed constituent's position in the sentence, as well as what parts of speech or subconstituents generated it. If a single constituent spanning the entire sentence is present in the chart then a successful parse has been found and a simple recursive backtrace will yield the structure.

The two basic data structures—keys and arcs—and three complex ones—key list, arc list and chart—will be discussed in further detail below. First, however, I will go over the two linguistic structures necessary for chart parsing: a grammar and a lexicon.

2.1 Linguistic Structures

In order to successfully parse a sentence, the algorithm relies on access to a lexicon detailing the part(s) of speech (POS[s]) of, at a minimum, every word in the sentence, and a set of Context Free Grammar (CFG) rules[1] that constrain how syntactic structures may be built in the target language. For example, a (very small) language may have the lexicon

(1) *Lexicon*

Part of Speech	**Entry**
PN	I
N	can play guitar
V	play
AUX	can
DT	a the
ADJ	five-string

Where PN is defined as a pronoun, N is a noun, DT is a determiner (article), ADJ is an adjective, V is a verb and AUX is an auxiliary verb, which are also referred to as terminals. Thus words can have multiple POSs and POSs can have multiple words associated with them.

A grammar fragment might feature rules like

[1] I refer readers unfamiliar with CFG to Jurafsky (2009), as full discuss of CFG is beyond the scope of this project

(2) *Grammar Rules*

Rule
S → NP VP
S → VP
VP → AUX VP
VP → V NP
NP → PN
NP → DT N
NP → DT ADJ N

where S indicates the initial symbol, NP is a noun phrase and VP is a verb phrase, all of which are called nonterminals. The set of units on the right-hand side (RHS) of the arrow indicate what constituents and POSs may generate the larger constituent on the left-hand side (LHS). This set of rules demonstrates that it is possible to have both a terminal and a nonterminal on the left hand side of the rule. Also, recursion is allowed, which is a feature of all known natural languages.

This grammar will successfully parse sentences like

(3) I can play the guitar.
(4) I can play a five-string guitar.
(5) Play the guitar!

However, it should be noted that this grammar is *not* appropriate for sentence generation, as it could produce such ill-formed output as

(6) * I can can can can can play I.[2]

I must also stress that the present chart parsing implementation does not check for grammaticality, and may find a parse for an ungrammatical structure, or fail to parse a perfectly grammatical one. It is up to the user to maintain consistent grammar rules and provide input that follows whatever standard is in place.

It is an advantage of the present program, however, that the codes for terminals and nonterminals are not confined to a specific set. In fact *any* variation of CFG rules and POS tags may be used. This not only provides great flexibility for the user—who may tailor parts of speech and rules to a particular application (such as Named Entity Recognition), or design a grammar according to certain metric (separating *didn't* into *did* and *n't* or keeping it as unit, for example)—this also means that *any language* whose structure can be described in CFG could be parsed by this program.

I would also like to note that I will be using the term *constituent* to refer to any unit in the grammar, both atomic and compound.

[2]* indicates a sentence that is ungrammatical to a native speaker.

2.2 Data Structures

The theoretical data structures needed to successfully chart parse a sequence of words can be separated into the types unit and list[3]. The forms below are borrowed from Ghassem-Sani.

2.2.1 Units

The unit category is made up of two different kinds: constituents and rules, which I will hereafter refer to as keys and arcs, respectively. A unit is of the form

(7) $< i > C < j >$

where C is a constituent, POS or word; $< \cdots >$ is the edge of the constituent in the input; i is the index of the edge on the left side; and j is the index on the right.

A given arc may take any one of the following forms:

(8) $< i > A \rightarrow \bullet Q_1 \ldots Q_n < j >$
(9) $< i > A \rightarrow Q_1 \ldots Q_k \bullet Q_{k+1} \ldots Q_n < j >$
(10) $< i > A \rightarrow Q_1 \ldots Q_n \bullet < j >$

where $< e >$ is an edge in the input with a particular index e; A is the LHS of a given grammar rule; $Q_1 \ldots Q_n$ is the set of units on the RHS; and $\bullet$ is the symbol marking progress through the arc.

Not part of the basic structure of a key, but essential to the successful output of a parse tree, is a record of which specific keys generated it. This is what allows us to backtrace from the final constituent, one of the core concepts of dynamic programming. There are several ways to store these values, but for this program I have incorporated a history structure into each key. This is further discussed in Algorithm Implementation.

2.2.2 Lists

Chart parsing uses four lists: the words in the sentence in key form, operational constituents and POSs in key form, operational arcs and completed constituents. Throughout the rest of this paper, I will refer to these as the Input List, Key List, Arc List and Chart, respectively.

While the Arc List and Chart may be in any order, the ordering of the Input List and Key List is essential. The Input List corresponds to the sequence of words in the input string, so in order to achieve the correct parse their original order must be maintained. Words are processed in a left-to-right fashion.

The Key List should be implemented as a queue. In the process of chart parsing, new keys representing larger constituents may be generated from arcs, and these must be processed *after* the smaller keys in order for every tree structure in the sentence to be explored.

[3]Actual data structure implementation does not follow these formats.

The Arc List likewise does not depend on order, but is instantiated as a queue for the purposes of this project.

The Chart stores all processed keys, and therefore completed constituents. It allows the parser to track viable syntactic structures for the input sentence as they appear. A parsable sentence will thus result in an S constituent (for the grammar in [2]) spanning the length of the sentence, and it is with this key that backtracing begins.

2.3 Algorithm

Chart parsing is a form of dynamic programming, and therefore has two basic steps: generating the "program", or Chart in this case, and backtracing to yield the result. The algorithm described below discusses arcs and keys in reference to their forms in (7)-(10). Their actual implementation in the program is slightly different, which is discussed further in the section Algorithm Implementation.

2.3.1 Generating the Chart

Chart parsing starts by generating keys out of the input string (recording the indices of the left and right edges of the word) and adding them to the Input List. Once the parsing algorithm has been initiated, it will remove the first item from the Input List and continue until all input keys have been removed, at which point it terminates.

For a given word, the program refers to the lexicon to find its POSs. Each POS is made into a key, with its edge indices matching those of the input word. These are added to the Key List.

From here, the algorithm removes a key and finds all arcs in the grammar whose right-hand sides begin with the POS of the key (match Q_1). These rules are made into arcs by assigning left and right indices corresponding to the *left* edge of the key and a $\bullet$ at the front of the RHS constituents. This form corresponds to the arc in (8). These arcs are added to the Arc List.

Next, each arc gets "extended" with the current key. Only arcs whose right edge index matches the left edge index of the current key, and whose $\bullet$ is to the immediate left of a constituent code matching that of the key are eligible for extension. Extension is carried out by creating a copy, moving the $\bullet$ of the copy one constituent to the left (indicating the constituent has been found in the sentence at a location identical to that specified by the arc), and updating the right edge of the arc to match the right edge of the current key. The copy is necessary because several keys in the course of the algorithm could be used to extend the same arc, and it is vital that these others not get blocked as this may result in a failure to parse.

Any arc whose $\bullet$ is at the end of the RHS sequence of constituents is made into a key itself. This new key will be the nonterminal on the LHS of the arc and its left and right edges will correspond to the arc's left and right edges. It gets added to the end of the Key List, and the loop continues.

Once the Key List is empty, the algorithm removes the next word on the Input List for processing.

The following gives a pseudocode version of the algorithm, taken in part from Jurafsky (2009 p. 449):

```
function CHART-PARSE(words, grammar, agenda-strategy) returns chart
    INITIALIZE(chart, agenda, words)
    while agenda
        current-edge ← POP(agenda)
        PROCESS-EDGE(current-edge)
    return(chart)

procedure PROCESS-EDGE(edge)
    ADD-TO-CHART(edge)
    if INCOMPLETE?(edge)
        FORWARD-FUNDAMENTAL-RULE(edge)
    else
        BACKWARD-FUNDAMENTAL-RULE(edge)

procedure FORWARD-FUNDAMENTAL-RULE(($A \to \alpha \bullet B\beta, [i, j]$))
    for each($B \to \gamma\bullet, [j, k]$) in chart
        ADD-TO-AGENDA($A \to \alpha B \bullet \beta, [i, k]$)

procedure BACKWARD-FUNDAMENTAL-RULE(($B \to \gamma\bullet, [j, k]$))
    for each($A \to \alpha \bullet B\beta, [i, j]$) in chart
        ADD-TO-AGENDA($A \to \alpha B \bullet \beta, [i, k]$)

procedure ADD-TO-CHART(edge)
    if edge is not already in chart then
        Add edge to chart

procedure ADD-TO-AGENDA(edge)
    if edge is not already in agenda then
        APPLY(agenda-strategy, edge, agenda)
```

2.3.2 Backtracing

Once the generation portion of the algorithm terminates, the next step is to backtrace and output the tree structure. A sentence has been successfully parsed if the Chart contains a key representing an entire sentence (the initial symbol: S in the grammar presented above) whose left edge is 0 and right edge corresponds to the right edge of the last word on the Input List. Intuitively, this means we have built a syntactic structure that spans the entire sentence. If there is no such key on the Chart, the sentence has failed to parse and there is no viable output.

Backtracing is a process that explores the constituents that created a given key recursively until a terminal is reached. This history represents the syntax tree of the input.

Of course, some applications of chart parsing are only interested in a binary output indicating a successful or failed parse, in which case this step is unnecessary.

3 The Program

In the last section I gave the theoretical construction of a chart parsing algorithm. This section discusses its actual implementation in Interactive Parser, a program written by the author to not only parse sentences, but provide an educational step-by-step breakdown of the process for the user. In addition, Interactive Parser will dynamically build a grammar and lexicon from user input, either from imported files or on the fly.

The sections below go over basics for loading and running the program, special features, implementation of crucial data structures and algorithms, and finally ways to improve the program in the future.

3.1 Features

This section gives an overview of the basic features of the Interactive Parser.

3.1.1 Loading the Program

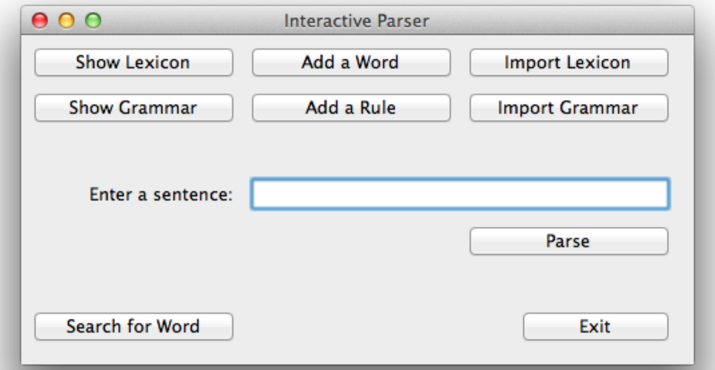
Interactive Parser runs with Python 3 and Tkinter. It can be loaded with the following command:

(11)

```
$ python3 interactive_parser.py
```

This will present the user with the following Graphical User Interface:

(12)



3.1.2 Loading Files

The Interactive Parser allows the user to import a previously written grammar with the button Import Grammar and/or a previously written lexicon with the button Import Lexicon. The user may also start with an empty grammar and/or lexicon and write rules on the fly, or any combination of the above. Of course, the soundness of the grammar and lexicon as a whole are up to the user. A bad set of rules or inconsistent POS code could result in a failure to parse. However, this also gives the user a great deal of freedom to develop a grammar suited to a particular language or application.

While the user may determine the structure and content of the language, the structure and content of the files themselves are quite strict. The lexicon must follow the format

(13) TERMINAL : word_1, word_2, ... , word_n

where the terminal is a POS followed by a colon and a series of one or more comma-separated words that have that part of speech, all on one line. Different POSs should be on separate lines. The program will convert words to lower case (with the exception of *I*) and terminals to upper case automatically, so either is fine as input, however it will conflate codes distinguished by case.

The grammar file must be in a format consistent with the following:

(14) NONTERMINAL $--\!>$ [$\text{TERMINAL}_1|\text{NONTERMINAL}_1$], ... , [$\text{TERMINAL}_n|\text{NONTERMINAL}_n$]

The nonterminal is followed by two hyphens and a greater-than sign, then a sequence of one or more terminals or nonterminals, all on one line. Multiple rules should be on separate lines. As with the terminal in (13), these are converted to upper case by the program, so lower case is permissible as input but may be conflated with other codes.

It is also possible to import the lexicon and grammar from the command line as arguments. If this option is taken, both most be listed, the lexicon file first and the grammar file second. Thus an alternative command to (11) for starting up the Interactive Parser is

(15)

```
$ python3 interactive_parser.py lexicon_file grammar_file
```

Included with this paper are the files sample_grammar.txt and sample_lexicon.txt.

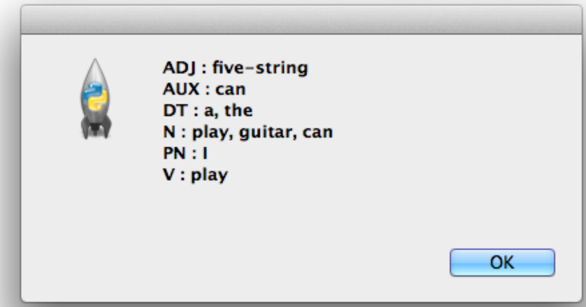
3.1.3 Adding Rules and Words

Whether starting from the ground up or importing complete grammars and lexica, the user may add a new rule or definition at any time. These should be entered individually (including POS-word pairs) and follow the formats shown in (13) and (14).The buttons Add Rule and Add Word provide this function.

3.1.4 Viewing Lexicon and Grammar

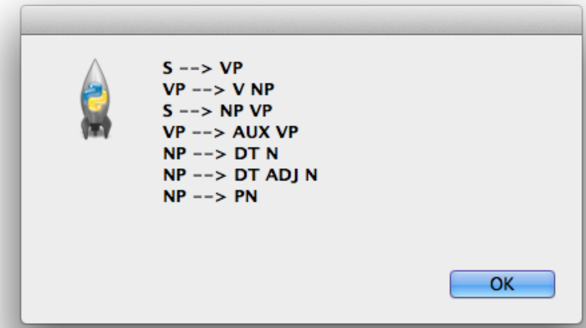
The buttons Show Lexicon and Show Grammar create a visual representation of these items in a separate window and in the format specified by (13) and (14). The grammar fragment featured in this paper would output windows like those below, the lexicon in (16) and the grammar in (17):

(16)



```
ADJ : five–string
AUX : can
DT : a, the
N : play, guitar, can
PN : I
V : play
```

(17)



```
S --> VP
VP --> V NP
S --> NP VP
VP --> AUX VP
NP --> DT N
NP --> DT ADJ N
NP --> PN
```

3.1.5 Searching the Lexicon

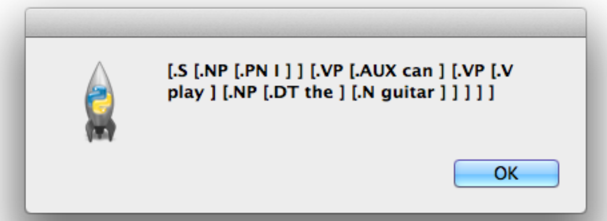
The button Search for Word allows the user to search for the part(s) of speech of a given word, if it exists. The absence of that word from the lexicon prompts the user to add it.

3.1.6 Parsing

The user enters the sentence to be parsed into the dialog box and clicks PARSE. Although most punctuation will automatically be removed, it is advised that the user avoid including it, if possible. Unusual symbols may not be stripped, resulting in the presence of these in the lexicon as part of the word itself. Single quotes are particularly problematic, as these are identical in form to apostrophes, which are specifically not removed to allow for contracted and possessive words such as *didn't* and *guitar's*. In addition, everything but the word *I* is converted to lower case[4]. Thus proper nouns will not retain capitalization (a shortcoming discussed in Future Improvements). Lastly, the program currently does not support parsing across periods, so sentences should be entered one at a time.

The output of parsing is a bracket-delimited tree structure. Internal nodes are marked with a period, followed by their children. Leaf nodes are the words themselves. The following is output for *I can play the guitar* under our grammar fragment.

(18)



```
[.S [.NP [.PN I ] ] [.VP [.AUX can ] [.VP [.V
play ] [.NP [.DT the ] [.N guitar ] ] ] ] ]
```

Due to time limitations I was unable to develop my own graphical output, however this tree representation is directly translatable to the qtree environment of LaTeX (available at http://www.latex-project.org/). An example preamble and command to produce the tree in (18) is

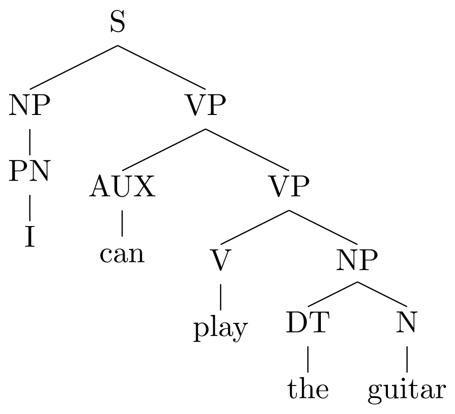
(19)

```
\documentclass{article}
\usepackage{qtree}
\begin{document}

\Tree
[.S [.NP [.PN I ] ] [.VP [.AUX can ] [.VP [.V play ] [.NP [.DT the ] [.N guitar ] ] ] ] ]

\end{document}
```

[4]This is because *I* is the only word in the English language that is not a proper noun, yet is capitalized. This change is more of an aesthetic decision than a linguistic one, as I felt the lower case *i* was an eyesore in the output that was simple to correct, unlike proper nouns.

This produces the tree

(20)



3.2 Special Features

The are two special features to this program that distinguish it from other chart parsers. As discussed above, the user is able to add to the existing grammar and lexicon while the program is running, which in itself is a special feature, but this function would be wasted if not for the option, at exit, to save the lexicon and grammar back to a file. Thus clicking Exit brings up two windows asking the user if he or she wants to save either or both the working grammar and lexicon to two separate files. This enables the continual improvement, extension and long-term development of a grammar and lexicon.

Secondly, this program also unprecedentedly offers insight into the process of chart parsing. After entering a sentence and clicking Parse, the user has the option of viewing a "verbose" parse, which means that each step of the algorithm is presented in sequential windows with an explanation and often a visual representation. It is possible to cancel out of the verbose parse at almost any time, yielding the final tree structure. As this option can become very verbose indeed, I refer the interested reader to the Interactive Parser itself instead of displaying examples below. In addition, the verbose option has not been adjusted for very long sentences, which may result in an unmanageably large Chart and Arc List. It is advised that those who wish to view the chart parsing process experiment on short sentences.

3.3 Implementation Techniques

In this section I will discuss certain portions of my implementation of the chart processing. The focus will be the instantiation of the units discussed in Data Structures, the process of extending the arcs and the recursive backtrace.

3.3.1 Data Structure Implementation

The two data structures I wish to discuss are the keys in (7) and the arcs in (8)-(10). The information to store in the keys consists of their POS, left edge, right edge, unique name and the keys that created them. I chose to implement this information as a list, in that order. Thus a key for a terminal follows the format

(21)

```
[part_of_speech, left_edge, right_edge, name, -1]
```

where `part_of_speech` is a string, `left_edge` and `right_edge` are integers, `name` is a unique integer (starting at 0 and increasing by one for each generated key), and -1 is the identifier of a key for a word, as these are not created by subconstituents. Thus, after obtaining the parts of speech for a word in list form (`pos_list`), keys are produced with the short for loop

(22)

```
for pos in pos_list:
    key = [pos, word[1], word[2], name, -1]
    name += 1
    key_list.append(key)
```

The program will also create keys for full constituents. These differ from the key presented above in that their edges correspond to arc edges, and furthermore the history of their creation is represented by a list of the `name`s of their subconstituents. Their format is then

(23)

```
[part_of_speech, left_edge, right_edge, name, [sub_1, ... , sub_n]
```

Here, `left_edge` and `right_edge` are the left and right edges of the original arc, and the final item is a list of the `name`s of the keys that built this constituent, one entry for each constituent on the RHS of the original rule.

The arcs are also represented by a list. Their format is

(24)

```
arc = [rule, [key[1], key[1]], progress, tracer]
```

where `rule` is a list of strings representing all constituents, with the LHS of the rule in position 0 and the RHS in subsequent positions, in order; `[key[1], key[1]]` is a list of the edges of the arc, which are initially the index of left edge of the key that caused it to be added to the Arc List; `progress` is an integer tracking progress through the arc during chart parsing, which is incremented by one each time the arc is extended (this can be thought of as the index of $\bullet$ on the RHS); and `tracer` is a list of the `name`s of the constituents that extended the arc.

3.3.2 Extension Implementation

Extension is the simple process of updating the right edge of the arc to match the left edge of the extending key, incrementing the progress marker and updating the `tracer` list to include the key. It undergoes the following lines of code:

(25)

```
if arc[0][arc[2] + 1] == key[0] and arc[1][1] == key[1]:
    arc[1][1] = key[2]
    arc[3][arc[2]] = key[3]
    arc[2] += 1
```

The preliminary `if` statement checks for the criteria for extension. The extended arc is then returned.

3.3.3 Recursive Backtrace Implementation

The backtrace is the recursive function `find_parse` that returns the string representation of the tree. The base case (where the recursion terminates) is the point at which the program has reached a node that contains no history; that is, when the history portion of the key contains a -1:

(26)

```
if node[4] == -1:
    word = get_node(node, word_list)
    if word[0] == 'i':
          word[0] = 'I'
    return('[.' + node[0] + ' ' + word[0] + ' ] ')
```

It also corrects for the unique orthography of *I*. The recursive portion of this method is as follows:

(27)

```
else:
    parse = '[.' + node[0] + ' '
    children = find_children(chart, node)
    for child in children:
        parse += find_parse(chart, child, word_list)

    parse += '] '

    return(parse)
```

The function `find_children` returns the list of keys in the chart that built the current key.

This concludes the overview of the program's functions and features. See Appendix.txt for the full source code. The final sections will discuss its shortcomings as well as potential fixes and improvements. Lastly, I will go over some of the work that went into the creation of this program.

3.4 Future Improvements

As a initial effort, this project succeeds on many levels: not only have I implemented a chart parser, but included interactive and user-friendly functions, given the user a look under the hood with detailed and informative descriptions of almost every step in the algorithm, and added the very useful option of saving the working grammar and lexicon to back-up files. As with any project, however, there is always room for improvement. The next few sections each discuss a particular area where I feel this program could be improved, though they are by no means inclusive to all the shortcomings.

3.4.1 Rule and Word Deletion

As it stands, the only way for a user to delete a word from the lexicon or a rule from the grammar is to manually delete it from the output file and reimport. A full-featured version of this program should include two more buttons DELETE A WORD and DELETE A RULE that would allow the user to perform these tasks during run time.

3.4.2 Ambiguity

A major shortcoming of this parser is that it is unable to yield more than one parse of a sentence, thus it cannot account for ambiguity in language. Ideally this parser would search for every interpretation (that is, every spanning S key on the chart) and return these as alternative interpretations. However, this fell outside of the scope of this project, the overall aim of which was to build a parser that invites user interaction.

3.4.3 Optimality

While the run time of basic chart parsing is vastly improved over brute-force parsing, there are still some ways to further optimize the algorithm, which runs in $O(n^3)$, where n is the length of the Input List. Without going into the details, it is possible to predict which constituents will be part of the final parse, thereby eliminating some of the bulk of the Arc List, which only ever increases or maintains its size in the course of the algorithm.

3.4.4 Multiple Sentences

The labor-intensive process of inputing sentences one at a time would ideally be eliminated with the implementation of multiple-sentence parsing. This would involve the identification of the period (or some other symbol) as the parse delimiter, and would increase backtracing complexity.

3.4.5 Robustness

While this program is appropriate for the parsing of most English sentences, its robustness in punctuation elimination, unusual character handling (such as for borrowed words like *résumé*) and morphological evaluation (for example, splitting *hasn't* into *has* and *not*) has not been fully tested or implemented. It is also a great shortcoming that this version does not handle proper nouns.

Furthermore, numerous characters in foreign languages (ñ in Spanish and ü in German, not to mention the alternate alphabets of Russian, Japanese, etc.) would presently pose a problem. Future developments of this program should enable the user to select a language or input an acceptable alphabet, in addition to specifying characters to be deleted or retained, and capitalized or uncapitalized (proper nouns in English, all nouns in German, etc.). Currently, the treatment of punctuation is not robust in all situations and potentially subject to error.

3.4.6 Graphics

Finally, the representation of the final parse is very unsatisfying. I would have liked to generate a graphical version within the program, but could not dedicate the time or resources to implement this, especially considering that it was not the focus of the project.

3.5 Division of Work

This project was created and developed by me alone. The actual implementation of the data structures and algorithms were my own versions, not borrowed from any other source, although I owe thanks to the authors in the References section and Professor Tanimoto for basic chart parsing theory, pseudocode and advice.

4 Conclusion

This paper has given an overview of chart parsing—its data structures, algorithms and implementation—and outlined the general use and special features of Interactive Parser. While many parsers may be slightly faster, slightly flashier or more robust across environments, Interactive Parser provides functions that the others do not: a breakdown of the chart parsing process that is easy to understand, informative and attractive in presentation; an opportunity to build a grammar and lexicon from nothing[5] and save it for future use.

[5]Useful for CSE 415 homework!

References

Ghassem-Sani, Gholamreza. “Chart Parsing” N.p., n.d. Web. 5 Oct. 2012. <sharif.edu/ sani/courses/nlp/lec3.pp>.

Jurafsky, D. and J. Martin (2009). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition.* 2nd ed. Upper Saddle River, New Jersey: Pearson/Prentice-Hall.

Kay, M. (1982). Algorithm schemata and data structures in syntactic processing. In Allén, S. (Ed.), *Text Processing: Text Analysis and Generation, Text Typology and Attribution*, pp. 327-358. Almqvist and Wiksell, Stockholm.