Exploring Effective Parsing of Context-Free Grammars

Tony Liu

May 19, 2014

1 The Problem

1.1 Introduction

Parsing is the analysis of a string of symbols within the context of some predefined grammar, and has many essential applications in the fields of computer science and linguistics. In programming language compilers, parsing is required for syntactic analysis of the input source program, where the syntactic tokens of a given source file can be broken down and arranged in a meaningful structure for the compiler to recognize. Similarly, parsers are needed in computational linguistics to build parse trees where the input string can be organized into meaningful parts of speech. In both of these situations, an efficient and robust parsing algorithm is needed because the source file or program could contain a large amount of tokens to parse and because the defined grammar may be complex. In this paper, we will examine top-down parsing techniques performed on context-free grammars, focusing on the Earley parser, which leverages dynamic programming to produce a successful parse in time polynomial to the size of the input.

1.2 Definitions

First, we will supply some defintions. A grammar G is defined as $G = (\Sigma, N, P, S)$, where Σ is a finite set of terminal symbols, N is a finite set of nonterminal symbols, P is a finite set of production rules, and S is a start (nonterminal) symbol [1]. A symbol s is a stand-alone unit within a grammar; symbols may be terminals or nonterminals. A production rule p has the form $A \to \alpha$, where $A \subseteq N$ is a sequence of nonterminals and α is a string of terminals and nonterminals. A may have different production rules associated with it, written as $A \to \alpha \mid \beta$, where α and β are strings of symbols. A grammar is *context-free* if all production rules have only a single nonterminal on the left hand side, $|A| = 1$. That is, a production rule can be applied whenever a nonterminal exists, regardless of the symbols that surround it. We'll define the size of a grammar as the sum of the lengths of the total possible right hand sides for its productions:

$$|G| = \sum_{A \in N} \sum_{(A \to \alpha) \in P} |\alpha|$$

A *language* is the set of all the possible valid strings (sentences) that can be generated by a grammar G, denoted as $L(G)$. We'll define the input string s as $s = a_1 a_2 a_3 ... a_n$ where a_i is a terminal symbol. The size of the input string s is defined as $|s| = n$. Two strings are equal if they both contain the same sequencing of the same terminals.

1.3 Formal Problem Statement

Given a context-free grammar $G = (\Sigma, N, P, S)$ and an input string s, does there exist a sequence of terminals w that can be generated by applying rules found in P such that $w = s$? That is, can

every symbol in s be matched to a terminal produced by the grammar G that is consistent with a valid sentence in the language of G? Subsequently, if there is such a string w, the sequence of production rules applied to symbols can be traced such that a parse tree illustrating the steps taken to get to w can be generated.

2 Parsing Basics

2.1 Top-Down vs. Bottom-Up

There are two main approaches that can be taken when attempting to parse an input string in accordance to some CFG. One approach would be to begin with the input string s, matching rules that produce the terminals in s and building a parse tree upwards; a successful parse would produce a tree rooted at the start symbol S while having leaves (all the terminal symbols) match the entire input s. This technique is called *bottom-up* parsing as it begins at the input and works upwards when applying rules. The other other approach would be to begin with the start symbol S, expanding possible solutions downwards and seeing if the terminals generated from applying the production rules correspond to the input; this is known as *top-down* parsing.

Both strategies have their advantages and disadvantages. Bottom-up parsing will spend time building partial solutions for portions of the input that will never fit with the surrounding symbols in the input in a valid structure, leading to many discarded partial solutions that have no hope of reaching the start symbol. Similarly, though top-down parsing will always be building a valid sentence within the grammar because it begins at the start symbol, it will spend time building partial solutions that will not be consistent with the input [2]. Though efficient algorithms exist utilizing either strategy, top-down parsing is a more structured and organized approach to parsing as we will always be building one centralized parse as opposed to having multiple partial solutions in consideration. We now will examine a basic parsing technique utilizing top-down parsing.

2.2 DFS: A Basic Parser

A Depth-First Search can be used to implement a basic top down parser. Beginning at the start symbol, the algorithm will expand the leftmost nonterminal, adding all of the possible states as generated by the production rule to the set of states to be considered. Then, the algorithm will explore one of the recently added states, adding more states to the set if the exploration hits a nonterminal. When terminal symbols are found in the current state being explored, the algorithm will check whether the terminals are consistent with the input; if not, the state is discarded and the next state in the set is considered. Depth-first search lends itself naturally to a stack implementation, with states being pushed onto the stack while the current state is being explored, and exploring a state to either its completion or failure before considering other states. A simple implementation of this algorithm can be found in DParser.java, and pseudocode is presented on the following page.

This algorithm is simple, but not very efficient; if we consider the worst case, the algorithm may have to expand a nonterminal on each pass through the loop, and an expansion of a nonterminal could lead to a large number of new states pushed on the stack (bounded above by $|G|$). Coupled with the fact that many partial solutions could be built and discarded multiple times, the runtime of the algorithm could easily become exponential. Additionally, the DFS Parser does not cope well with some fundamental problems encountered while parsing: recursive production rules and ambiguity.

```
input  : An input string
output: whether input ∈ L(G)
Given Grammar G = (Σ, N, P, S);
Initialize Stack with S;
while Stack is not empty do
    current = Stack.pop();
    if current matches input then
        Return yes;
    end
    if current is a partial match then
        next = A ∈ N, A ∈ current;
        foreach Rule ∈ P, Rule = next → α do
            Stack.push(apply(Rule, current));
        end
    else
        Discard current;
    end
end
Return no;
```

Algorithm 1: Pseudocode for DFS Parser

2.3 Common Parsing Problems

Recursive production rules and ambiguity are two prevalent problems that must be addressed while parsing. A *recursive* production rule is any rule that contains the left hand side of the rule within the right hand side of the rule, in the form $A \rightarrow A\alpha$, where A is a nonterminal and α is a sequence of symbols. We can see that potential problems arise if we attempted to use a DFS parser on a grammar with recursive rules; the algorithm could repeatedly expand A when it is encountered on the right hand side of its own production rule, continuously pushing new states onto the stack and never terminating. This problem can be mitigated by altering the grammar such that recursive rules do not occur or by setting a maximum depth for an exploration; however both solutions are not ideal as they make the algorithm less robust in terms of correctness [2].

Ambiguity is another common problem that arises; given an input string s, a grammar G is *ambiguous* if there exists two or more valid parses that correspond to s within the language. A solution to ambiguous grammars would be simply to track every valid parse within the search space and return all of them once the algorithm is complete. Since the DFS Parser only returns a single valid parse, it must be modified for ambiguous grammars. However, since the runtime for just finding a single parse tree could become exponential, attempting to search the entire problem space using DFS for all possible parses could quickly become impratical for even moderately sized grammars [2].

3 The Earley Recognizer: A Dynamic Solution

3.1 Overview

The Earley Recognizer, developed by Jay Earley in 1970, utilizes a top down approach as well as memoization of partial solutions to provide a robust and efficient parsing algorithm; through the use of dynamic programming, the algorithm more adequately addresses the problems of recursive rules and amibguity while also significantly improving on runtime in comparison to the DFS Parser. The Earley Recognizer builds partial solutions beginning at the start symbols,

and stores relevant states within a *parse chart*. It generates partial solutions systematically using three operations (*scan*, *predict*, and *complete*) and never generates duplicate partial solutions. The later entries of the parse chart are built by looking backwards into the previous entries of the parse chart.

First we will develop some simple data structures used by the algorithm, and then discuss implementation in more detail.

3.2 Data Structures

The Earley Recognizer requires an augmented version of the production rule, called a *dotted rule*, to keep track of how far parsing has gotten within the right hand side of a rule. A dotted rule D is a rule of the form $A \to \alpha \bullet \beta$, where A is a nonterminal, α is the sequence of symbols that have already been analyzed, and β is the sequence of symbols within the rule that has yet to be analyzed. Every time a symbol in the right hand side is analyzed, the $\bullet$ is moved forward by one.

Additionally, we'll need to keep track of where in the parse a dotted rule was started in relation to the input. Thus we also maintain a structure called a *state*, which is esentially a pair (D, i), where D is a dotted rule and i is the position in the input string where D began parsing its right hand side. This allows us to keep track of where dotted rules originated, even if they occur in different state sets.

Finally, a parse chart S is a list of lists, with each entry $S[i]$ corresponding to a list of *states* (or *state set*) that are generated after analyzing the ith symbol in the input string. The parse chart has $n+1$ entries, corresponding to the n symbols in the input and one "dummy" start rule that acts as a seed for the beginning of the algorithm. The dummy rule has the form $\$ \to S$, where $\$$ is a fabricated start nonterminal and S is the actual start symbol for the grammar. The dummy rule is needed for cases where the start rule S of the grammar has multiple possible derivations.

3.3 Implementation

Here we will describe the basic implementation of the Earley Recognizer algorithm.

First, we can formally describe the recurrence. A state $s = (D, i)$ exists within the state set $S[k]$ if and only if there exists some state $s' = (D', i)$ in a previous state set $S[i]$ where $0 \leq i \leq k$ and D' is some dotted rule, and there exists either a valid rule application or equivalent kth symbol in the input for the next symbol in D' (leading to D, which is D' with the dot incremented by one).

$$\boxed{(A \to \alpha x \bullet \beta, i) \in S[k] = x_{valid} \wedge (A \to \alpha \bullet x\beta, i) \in S[i]}$$

A successful parse was completed if there exists a state s of the form $s = (\$ \to S\bullet, 0)$ in the $S[n+1]$ entry of the parse chart.

We will now describe the methods needed to build the occurrence. All notation that follows assumes we are at the kth string in the input, so we are currently building the entry $S[k]$. Capital letters represent nonterminals, lowercase letters represent terminals, and Greek letters represent a sequence of symbols (terminals/nonterminals).

3.3.1 Scanning

If c is the kth symbol in the input string, for every state s in $S(k)$ of the form $(X \to \alpha \bullet c\beta, j)$, we add $(X \to \alpha c \bullet \beta, j)$ to $S(k+1)$ [4]. That is, if we see a terminal symbol as the next symbol to be processed in any of the dotted rules in $S(k)$, we check if the terminal matches the kth symbol in the input string, and if so, increment the dot and place the state in $S(k+1)$.

3.3.2 Predicting

For every state in $S(k)$ of the form $(X \to \alpha \bullet Y\beta, j)$, we will add $(Y \to \bullet\gamma, j)$ to $S(k)$ for all production rules in P that have Y as the left hand side [4]. If we see a nonterminal as the next symbol in a dotted rule to be processed, we add new dotted rules to the current state set $S[k]$ corresponding to the nonterminal.

3.3.3 Completion

For every state in $S(k)$ of the form $(X \to \gamma\bullet, j)$, find states in $S(j)$ of the form $(Y \to \alpha \bullet X\beta, i)$ and add $(Y \to \alpha X \bullet \beta, i)$ to $S(k)$ [4]. Once a dotted rule D has been completed (fully processed), look back to the state set where it orginated from and increment the dot of any other dotted rules that contain the left hand side of D; those incremented states will then be added to the state set $S(k)$.

3.3.4 Pseudocode

The Earley Recognizer flow of execution is shown in the following pseudocode:

```
input  : An input string
output: whether input ∈ L(G)
Given Grammar G = (Σ, N, P, S);
Initialize Chart with state containing S;
for i from 0 to length(input) do
    foreach state ∈ Chart[i] do
        if state is incomplete then
            if state.next() is nonTerminal then
                predict(state);
            else
                scan(state);
            end
        else
            complete(state);
        end
    end
end
if Chart[i+1] contains completed start rule then
    Return yes;
else
    Return no;
end
```

Algorithm 2: Pseudocode for Earley Recognizer

Essentially, the algorithm scans through every state set from left to right, examining the each dotted rule and performing appropriate actions depending on the next symbol (if any) to be analyzed within the dotted rule: if the next symbol is a terminal, we run scan, if the next symbol is a nonterminal, we run predict, and if the rule has been completely analyzed we run complete. A state is only added to a state set in the chart if it does not already exist within the set (unless it is the start rule), solving the problem of recursive and repeated rules. A full implementation of the Earley Recognizer can be found in Earley.java, and an example parse chart based off the simple.txt grammar can be found in Table 1.

State 0	State 1	State 2	State 3
$(\$ \rightarrow \bullet S, 0)$	$(S \rightarrow \bullet The\ N\ V, 0)$	$(N \rightarrow dog\bullet, 1)$	$(V \rightarrow runs\bullet, 2)$
$(S \rightarrow \bullet The\ N\ V, 0)$	$(S \rightarrow The\bullet\ N\ V\ A, 0)$	$(S \rightarrow The\ N\bullet\ V, 0)$	$(S \rightarrow The\ N\ V\bullet, 0)$
$(S \rightarrow \bullet The\ N\ V\ A, 0)$	$(N \rightarrow \bullet dog, 1)$	$(S \rightarrow The\ N\bullet\ V\ A, 0)$	$(S \rightarrow The\ N\ V \bullet A, 0)$
	$(N \rightarrow \bullet cat, 1)$	$(V \rightarrow \bullet runs, 2)$	$(\$ \rightarrow S\bullet, 0)$
	$(N \rightarrow \bullet bird, 1)$	$(V \rightarrow \bullet flies, 2)$	$(A \rightarrow \bullet quickly, 3)$
		$(V \rightarrow \bullet soars, 2)$	$(A \rightarrow \bullet slowly, 3)$

Table 1: Parse Chart for "The dog runs."

3.4 Runtime Analysis

3.4.1 Time

By Earley's analysis of his algorithm, the Earley recognizer operates in the worst case $O(n^3)$ time, where n is the size of the input string, and in many cases does better than that runtime [3]. The maximum size of each state set stored within the parse chart after analyzing the nth symbol in the input is proportional to n; since each state is examined only once and only up to Cn new states can be generated as a result (where C is bounded above by the size of the grammar $|G|$), the size of each state set is bounded above by Cn.

Calling scan on a single state s within a state set $S(k)$ takes constant time, since we simply check whether the next symbol to be processed in s matches the kth input terminal and if so, add it to the state set $S(k+1)$. The predict method also spends constant time (proportional to C) on a single state set s, since it searches the grammar for production rules that match the next symbol in s and adds all rules that match it. Calling complete on a state s however, may take up to linear time (Cn) because an entire previous state set may be searched for any states that contain the completed production rule. Note that only one of scan, predict or complete is called on a single state s, so the runtime bound will be determined by the complete method.

Thus, analyzing a single state set (where the size of the set is bounded above by Cn) may take C^2n^2 time because of the complete method. And since there are $n+1$ states within the parse chart, we obtain a run time of $C^2n^2(n+1)$, which gives the asymptotic bound of $O(n^3)$. This runtime assumes that the grammar has already been given; if we were to require a grammar G as a part of the input, then after replacing all the constants C with $|G|$ we see a runtime of $O(|G|^2n^3)$.

This $O(n^3)$ truly is a worse case upper bound; the Earley recognizer is shown to perform as well as $O(n^2)$ or $O(n)$ depending on the grammar [3]. In the case of unambiguous grammars, the algorithm performs in $O(n^2)$ time. Going back to our analysis, the completer method was the step in the algorithm that spent up to $O(n^2)$ time for an entire state set ($O(n)$ time to process each state, up to $O(n)$ states within each state set). But we have also established that a state set can contain at most $O(n)$ states, so the only reason why a call to complete would spend $O(n^2)$ time building a particular state set was if there were multiple ways to add a given state. This can only be the case if the grammar was ambiguous; if the grammar was not ambiguous, each of the $O(n)$ items within a state set could have only originated from one state in some previous state set, thus allowing the complete method to run in $O(n)$ time processing an entire state, bringing the overall runtime down to $O(n^2)$ [3].

3.4.2 Space

As we have established previously, the number of states within a given state set $S(k)$ is proportional to Ck. Since we have $n+1$ state sets within our parse chart, and the number of states within the state set is bounded above by $C(n+1)$, we spend at worst $O(n^2)$ space while building the parse chart [5].

3.5 Traceback: Generating Parse Trees

The Earley algorithm as described can determine whether an input string exists within the language described by a grammar, but it does not return a parse of the input string. This can be remedied by a slight modification to the complete method: whenever a state s_i containing the rule $A \rightarrow \alpha$ is completed and a state s_j containing the rule $R \rightarrow \beta A \bullet \gamma$ is added to the parse chart, we construct a pointer from the instance of A in R's right hand side to the state s_i, which indicates that the A in the rule of state s_j was parsed as α, from our state s_i. Once the parse is completed, traceback would occur by following the pointer from the completed start rule in the final state set, tracing all the subsequent pointers and storing the states they lead to until all the pointers have led to terminals. In the absolute worst case, we trace pointers back to every single entry in the parse chart, which has $O(n^2)$ size. Thus, constructing the parse tree from a completed execution of the algorithm will spend time no greater than it took to construct the parse chart itself [3].

The algorithm checks if there exists a completed start rule within the final state set of the parse chart to determine whether a successful parse was made. In the case of ambiguous grammars, multiple completed start rules may exist within the final state set. Returning the different possible parses of the input string simply becomes running traceback on the instances of the completed start rule, as they will lead to different production rules within the chart.

4 Another Method of Parsing: CYK

There are a number of other parsers that are comparable to Earley in asymptotic performance. In particular, the CYK Algorithm (from Cocke, Younger, and Kasami, all of whom developed implementations of the algorithm independently), leverages dynamic programming in bottom-up parsing. Like Earley, it builds and stores partial solutions in a parse table. We will explore its implementation and its advantages/disadvantages when compared against the Earley algorithm.

4.1 Implementation

The CYK algorithm requires grammars to be in *Chomsky Normal Form*. A context-free grammar in Chomsky Normal Form contains production rules of the following form:

- $A \rightarrow BC$, where A, B, and C are nonterminals.
- $A \rightarrow a$, where A is a nonterminal and a is a single terminal.
- $S \rightarrow \epsilon$, where S is the start rule and ϵ is the null symbol. [1]

We notice that grammars in Chomsky Normal Form have production rules that have right hand sides of size no greater than two, and the algorithm leverages this fact while constructing the parse table by partitioning the span of terminal symbols it is currently analyzing into two areas.

The CYK algorithm fills an $n+1$ by $n+1$ parse table (where n is the size of the input string), with each entry in the table holding some subset $V \subseteq N$ of nonterminals within the grammar. We can define the recurrence relation of the algorithm as follows: A nonterminal A with corresponding production rule $A \rightarrow BC$ appears in table entry $T[i,j]$ if and only if there exists entries $B \in T[i,k]$ and $C \in T[k,j]$ for some k $(i < k < j)$.

$$\boxed{A \in T[i,j] = B \in T[i,k] \wedge C \in T[k,j]}$$

Thus, the set $T[i,j]$ corresponds to valid nonterminals that "span" the input string from i to j. We know we have a successful parse if the start symbol appears in the entry $T[0,n]$ of the table [5].

The algorithm begins execution by seeding the superdiagonal of the table with the nonterminals that correspond to the terminals within the input string. For example, if the ith symbol in the input is a, and the production rule $A \rightarrow a$ exists within the grammar, then $A \in T[i, i+1]$. The algorithm then fills the table from left to right, beginning at the main diagonal entries and filling upwards (thus only the upper triangular portion of the table is filled). A simple implementation is given in CYK.java, and pseudocode is presented here:

```
input  : An input string
output: whether input ∈ L(G)
Given Grammar G = (Σ, N, P, S);
Initialize table T[n + 1][n + 1] for i from 1 to length(input) do
    T[i − 1, i] = A, A → input(i − 1);
end
for j from 1 to length(input) do
    for i from j-1 to 0 do
        for k from i+1 to j-1 do
            foreach Rule A → BC ∈ P do
                if B ∈ T[i, k] and C ∈ T[k, j] then
                    Add A to T[i, j];
                end
            end
        end
    end
end
if T[0,length(input)] contains S then
    Return yes;
else
    Return no;
end
```

Algorithm 3: Pseudocode for CYK

4.2 Runtime

The algorithm builds a chart with $n^2/2$ entries, and scans backwards through a given row and column. We have to check every nonterminal production rule in the grammar for each entry, scanning backwards through a given row and column will spend up to $O(n)$ time. Thus, filling one entry in the table spends $|G|n$ time. Given that there $n^2/2$ entries in the table, the CYK algorithm runs in $O(|G|n^3)$ time, which when assuming the grammar is given is $O(n^3)$ time.

4.3 Limitations

Despite the competitive runtime the CYK algorithm has in relation to the Earley parser, it has a number of limitations that affect its performance. As noted before, the CYK algorithm requires a grammar in CNF; though every context-free grammar can be converted into CNF, using CYK on general context-free grammars requires some preprocessing time for the conversion. Addtionally, depending on the nature of the source grammar, the converted Chomsky Normal Form may be significantly larger in size. Certain grammars can see an exponential increase in size, making the grammar all but unusable [5]. Variations of the CYK algorithm that do not rely on CNF have been developed; Martin and Leib proposed a variant that only required Binary Normal Form (all right hand sides have size no greater than two), but requires even more preprocessing in the form of developing auxilary data structures [6]. The requirement

of some form of preprocessing in order to use CYK makes it less attractive in comparison to Earley, which works universally on all context-free grammars.

Additionally, despite being a dynamic programming algorithm that builds only possible partial solutions, the CYK algorithm still spends a signficant amount of time performing needless computation. This is because of its nature as a bottom-up processor, where partial solutions that do not correspond to a completely successful parse are being built. The CYK algorithm also loses efficiency because it has to iterate over all of the entries in the parse table; though the size of the Earley table has the same asymptotic bound ($O(n^2)$), it rarely is forced to iterate over that many partial solutions.

The biggest blow to the viability of the CYK algorithm is its need for Chomsky Normal Form; many grammars become far too large after the conversion for the algorithm to be used effectively.

5 Testing and Results

In an illustration of the advantages of the Earley algorithm, a number of tests using the Earley algorithm and the DFS Parser were conducted on input strings of various sizes. The CYK algorithm was omitted from testing because the conversion of grammars to CNF made the grammar sizes unmanageable and testing impratical.

5.1 Setup

We use an unambiguous, nonrecursive grammar of medium size ($|G| = 83$) as the test grammar, as the DFS Parser fails to terminate on even the simplest recursive grammars (it should be noted that after converting the test grammar to Chomsky Normal Form, its size ballooned to $|G'| = 962$). Using differing start rules within the grammar, a random sentence generator creates a number of valid sentences within the language with a wide range of lengths. Due to the randomized nature of the generator, exact string lengths are difficult to achieve, but we test on input strings that are approximately 150, 350, 550, 900, and 1200 symbols in length. Stress tests on Earley of input size of approximately 5000 and 20000 were included as well. The algorithms are timed in milliseconds using the native Java method $System.CurTimeMillis()$, with timing beginning right before the algorithm is called and timing ending once the algorithm has confirmed a successful parse. 100 trials on each approximate input size were conducted.

5.2 Results

Avg. Input Size	DFS Parser Speed (ms)	Earley Speed (ms)
132	1.31	1.62
357	13.51	1.68
545	41.08	1.84
898	195.1	2.17
1256	510.87	2.40
5549	N/A	9.18
22314	N/A	44.87

Table 2: Test Results for DFS and Earley

Though the DFS Parser performed slightly better than Earley on the smallest input size, this is likely due to Earley requiring the initialization of a multitude of data structures and because the input size was small enough such that DFS could traverse through the entire search space in a reasonable amount of time. However, when moving on to larger input sizes, the DFS

scales very poorly due to its exhaustive search, while Earley performs well, handling the larger data sets with no substantial blow up in running time. These results are expected based on our previous analysis of the algorithms. For the CYK algorithm, its runtime appears to be affected by an extremely large constant factor, as the algorithm would not terminate in an appreciable amount of time on anything other than the smallest grammars. This may simply be the result of using this particular grammar; we should not dismiss the CYK algorithm as unviable, as it may very well perform better on other grammars and inputs. The poor/unusable performance of the CYK algorithm may be attributed to the simplistic CNF converter used to transform grammars; however, efficient conversion from CFG to CNF is a research topic in itself, so we'll leave CYK and focus on Earley.

6 Limitations of Earley and Extensions

Though the strength of the Earley parser is its flexibility, as it can be used for any context-free grammar, it loses in terms of performance to other parsers that specialize in a specific set of grammars. In particular, the grammars that describe the syntax of programming languages are limited and often unambiguous; a class of bottom-up parsers called LR parsers are deterministic and can parse an input string in $O(n)$ time [5]. The Earley parser is best suited for use in environments where grammars become complex, as it can handle many classes of grammars or can be easily extended to do so. Thus, the algorithm sees most of its use in the field of computational linguistics, especially in natural language processing, where problems such as ambiguous grammars run abound and performance is not as much of a priority as expressiveness.

Numerous extensions have been made to the algorithm over the years, which have improved the time and space efficiency of the algorithm and expanded its usability to different classes of grammars. The Earley parser can be modified to handle stochastic context free grammars, where each production rule and nonterminal is assigned a probability of occurring [7], and other extensions allow Earley to run in linear time on LR grammars [5]. Schabes shows that the "predict" states can be precomputed, leading to an improvement upon the practical efficiency of the algorithm [8].

Another simple modification we will explore here allows us to save space in the parse chart while Earley is executing. Referring back to Table 1, we notice that many of the completed states such as $(N \rightarrow dog\bullet, 1)$ are not used any further by the algorithm once it has been processed. So, keeping these states stored within the parse chart is unecessary. If a completed state only results in a single new item within the parse chart, we can perform an "eager completion", completing all states that can be completed and storing only the final result. Note that an eager completion halts if two or more states require the completed state to advance, as eagerly completing more than one state could result in a proliferation of new states. In certain cases (especially recursive grammars), eager completion reduces the space requirement of the Earley algorithm to $O(n)$ [5]. The many ways the Earley algorithm can be modified and utilized illustrates its strength as a general parsing algorithm.

7 Conclusion

In this paper, we established the parsing problem of context-free grammars, and explored a number of ways to solve the problem. Top-down methods of parsing are preferred over bottom-up methods as the partial solutions generated via top-down parsing are more centralized. We examined three algorithms that are designed to solve this problem: the DFS Parser, the Earley algorithm, and the CYK Algorithm. The Earley algorithm is a particularly viable solution because it leverages the advantages of top-down parsing and dynamic programming to solve the problem effectively. It performs markedly better than other basic techniques used to solve this problem, and though other algorithms may perform better on certain grammar classes, the

strength of the Earley algorithm lies in its expressiveness; it handles ambiguous and recursive grammars well, and can easily be extended for use in specific instances.

8 Acknowledgements

I'd like to thank Bill Lenhart for his helpful guidance on this project, as well as Steve Freund who provided the grammars I used when testing the algorithms. Thanks for a great semester!

References

[1] J. Power. *Notes on Formal Language Theory and Parsing.* Department of Computer Science, National University of Ireland. 2002.

[2] D. Jurafsky and J. H. Martin. *Speech and Language Processing.* Prentice Hall, Inc. 2008.

[3] J. Earley. An Efficient Context-Free Parsing Algorithm. *Communications of the ACM*, Volume 13 Issue 2 (94-102), 1970.

[4] J. Aycock and R. N. Horspool. Practical Earley Parsing. *The Computer Journal*, Volume 45 Issue 6 (620-630), 2002.

[5] D. Grune and C. J.H. Jacobs. *Parsing Techniques: A Practical Guide.* Springer Science + Business Media. 2008.

[6] M. Lange and H. Leib. To CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm. *Informatica Didactica*, Volume 8. 2009.

[7] A Stolcke. An Efficient Probabilistic Context-Free Parsing Algorithm that Computes Prefix Probabilities. *Association for Computational Linguistics*, 1995.

[8] Y. Schabes. Polynomial Time and Space Shift-Reduce Parsing of Arbitrary Context-free Grammars. *Assocation for Computational Linguistics*, 1991.