EXPLORING THE PROBLEM OF AMBIGUITY IN CONTEXT-FREE GRAMMARS

By

SAICHAITANYA JAMPANA

Bachelor of Technology in Computer Science
Jawaharlal Nehru Technological University
Andhra Pradesh, India
2001

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
In partial fulfillment of
The requirements for
The Degree of
MASTER OF SCIENCE
July 2005

EXPLORING THE PROBLEM OF AMBIGUITY IN

CONTEXT-FREE GRAMMARS

Thesis Approved:

M. H. Samadzadeh
Adviser

J. P. Chandler

N. Park

A. Gordan Emslie
Dean of the Graduate College

PREFACE

The general problem of ambiguity detection is unsolvable. Some context-free languages are inherently ambiguous. There is no algorithm that can detect the ambiguity of context-free grammars (CFG) in general since it is a provably unsolvable problem. The objective of this thesis is to study the notion of ambiguity in context-free grammars and grammars in general. The areas of this study include ambiguity in CFGs, ambiguity in other classes of grammars, and a number of algorithms for finding ambiguous strings and grammars. Also, an algorithm that finds ambiguous strings in a subset of context-free grammars is presented. This algorithm is based on the observation that ambiguity in strings cannot be induced by mere repetitions of applications of productions unless ambiguous strings can be generated without exhibiting the self-embedding property.

Ambiguity in different classes of formal languages and in some programming languages was studied. The problem of ambiguity detection in context-free grammars was studied in depth. An algorithm was designed and implemented to detect ambiguous strings generated by a subset of context-free grammars. The algorithm works successfully for all grammars in the test suite of grammars which are known to be either ambiguous or unambiguous. It was observed that checking the sentential forms for ambiguity instead of checking only the sentences, and applying the productions of the proper grammar instead of the CNF grammar to derive the sentential forms are promising techniques to keep the execution time low. It was also observed that the time in which ambiguity was detected in

the grammars in the test suite was largely independent of factors such as the productivity of a grammar (the number of strings that can be generated without exhibiting the self-embedding property), the degree of ambiguity of a string, and the lengths of strings generated.

ACKNOWLEDGEMENTS

I express my sincere gratitude to my advisor, Dr. Mansur H. Samadzadeh, for his guidance, assistance, and inspiration throughout the study. I am very grateful to him for his patience and support in completing my thesis work.

My sincere appreciation extends to my committee members, Dr. John P. Chandler and Dr. Nophill Park for their suggestions and supervision.

I thank my parents, Mr. Suryanarayana Raju and Mrs. Annapurna Devi and my brother Sundeep for their love and encouragement throughout the duration of my studies in the United States. My special thanks extend to my uncle and aunt, Mr. Ramamurthy Raju and Mrs. Satya Bharati, for their love, care, and support.

My hearty thanks go to my friends Pavan, Mahesh, Dheraj, Sudheer, Vishal, and Bharat for their moral support and help during my down times.

Finally, I thank the Computer Science Department for the quality education and the International Students and Scholars office of Oklahoma State University for their timely assistance. I also would like to thank other people I may have missed who deserve a mention.

TABLE OF CONTENTS

Chapter Page

I INTRODUCTION……... 1

1.1 Problem Statement……1
1.2 Thesis Outline……2

II PRELIMINARIES……...……3

2.1 Context-Free Grammars…..…….....3
2.2 Normal Forms for Context-Free Grammars……...…..... 5
2.2.1 Chomsky Normal Form……..... 6
2.2.2 Greibach Normal Form…….. 8

III AMBIGUITY……..…….10

3.1 Degree of Ambiguity…….…… 10
3.2 Ambiguity Detection and Removal……...…… 11
3.3 Inherent Ambiguity…….……...……11

IV AMBIGUITY IN OTHER CLASSES OF GRAMMARS..……..…….. 13

4.1 Ambiguity Inherent in Languages……...….. 13
4.2 Ambiguity in Programming Languages……..……..13
4.3 Ambiguity in Two Level Grammars…..……...…..14
4.4 Ambiguity in Omega Context-Free Grammars..……....…..15
4.5 Ambiguity in Context-Sensitive and Unrestricted Grammars..……...….. 16

V ALGORITHMS FOR FINDING AMBIGUOUS STRINGS……..……..... 18

5.1 Eickel and Paul's Algorithm ……..……...……18
5.2 Algorithm used by YACC (Yet Another Compiler Compiler)……...….. 18
5.3 Cheung's Algorithm…….…..19
5.4 Scope of the Algorithms …….....….. 19

VI PROPOSED APPROACH FOR FINDING AMBIGUOUS STRINGS…….20

6.1 Description……..……..…20
6.2 Three Variations of an Algorithm That Finds Ambiguous Strings…..……...21
6.3 Observations and Results …….....25

VII SUMMARY AND FUTURE WORK…… 29

7.1 Summary……29
7.2 Observations and Conclusions……...30
7.3 Future Work…….……...31

REFERENCES……..…...……. 32

APPENDICES……... 34

APPENDIX A - GLOSSARY……..……. 35

APPENDIX B – RESULTS OF THE PROGRAMS…….…..……..37

APPENDIX C – SOURCE CODE LISTING…….……..41

APPENDIX D – INPUT SUITE OF GRAMMARS……...…….. 77

LIST OF FIGURES

Figure Page

1 Parse Tree for an English Language Sentence……..……...……..4

2 An Example of Self-Embedding…….……...……7

3 An Example of Greibach Normal Form…….……...…….9

4 The Lengths of Sentential Forms at which Ambiguity was Detected in the 28 Ambiguous Grammars in the Test Suite Divided into Detection Using Sentences Only vs. Detection Using All Sentential Forms……..……...……...…...26

LIST OF TABLES

Table Page

1 Time Taken to Determine the Ambiguity of a Grammar……..…25

2 Execution Times (in seconds) of the three variations of the presented algorithm on a number of few ambiguous grammars…....…….……..…….. 37

3 Execution Times (in seconds) of the three variations of the presented algorithm on a number of unambiguous grammars……..…...…….……..…...38

4 Execution Times (in seconds) of the presented algorithm for ambiguous grammars with varied degree of ambiguity (DOA), productivity, and length ……..39

CHAPTER I

INTRODUCTION

Formal grammars provide a syntactical generative way of defining languages. Context-free grammars, one of the four classes of grammars as defined by Noam Chomsky [Hopcroft et al. 01], have a wide variety of applications. Context-free grammars are primarily used to build compilers to verify the syntax of computer programs. They can also be used to generate complex graphic designs from a set of basic constructs [Prusinkiewicz et al. 88].

Applications that use context-free grammars typically require a unique structure for each sentence the grammars generate. But unfortunately the definition of context-free grammars does allow for the possibility of having more than one structure for a given sentence. This problem, known as ambiguity, can cause serious problems and may make the meaning of a sentence unclear.

1.1 Problem Statement

Because of the marring nature of ambiguity in the realm of language translation and interpretation, unambiguous grammars are obviously preferred. To verify that a grammar is unambiguous, there must be a known procedure to disambiguate the grammar if it is demonstrated to be ambiguous. But there is no general algorithm that can disambiguate a given ambiguous grammar. Moreover, there are languages that are

inherently ambiguous, which means that the set of sentences representing the language cannot be generated by an unambiguous grammar. The algorithms for ambiguity detection that are available in literature can only determine if a given sentence is ambiguous with respect to a given grammar, or determine if a finite set of strings are ambiguous. However, the applications that are based on context-free languages involve potentially infinite sets of words or sentences.

1.2 Thesis Outline

The rest of this thesis report is organized as follows. In Chapter II, an introduction to context-free grammars is given and the concepts of parse trees and normal forms of context-free grammars are discussed in some detail. In Chapter III, the general concept of ambiguity, ambiguity removal, and inherent ambiguity are discussed. In Chapter IV, ambiguity in other classes of grammars and in some programming languages is discussed. In Chapter V, three ambiguous string detection algorithms and their respective scopes are discussed. A new algorithm for finding ambiguous strings in a subset of context-free grammars is presented in Chapter VI and the implementation details of the algorithm are also discussed. The summary and future work are discussed in Chapter VII.

CHAPTER II

PRELIMINARIES

In this chapter the basics of context-free grammars, parse trees, and normal forms of context-free grammars are discussed.

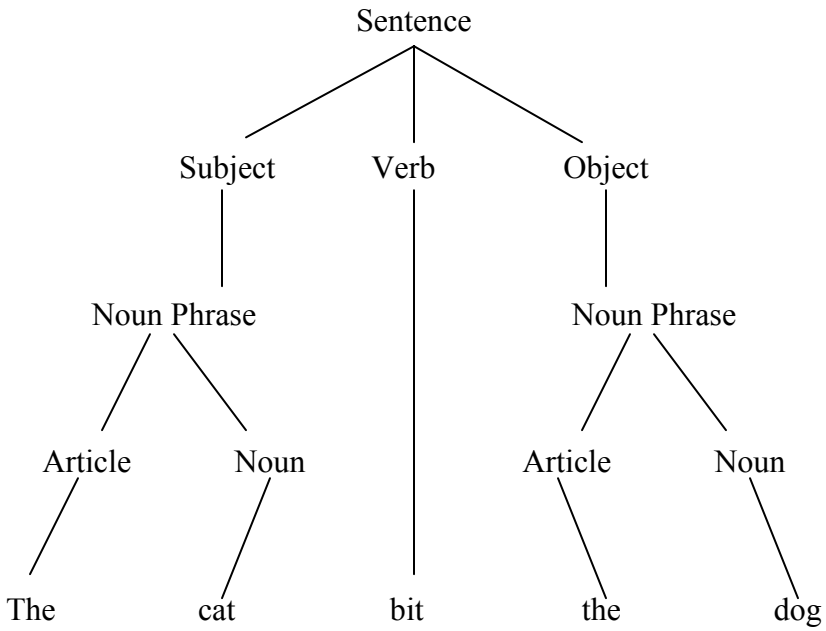
2.1 Context-Free Grammars

A grammar for a language is a set of rules that govern the generation of sentences in that language. Every language, whether it is a natural language (English, Spanish, etc.) or a programming language (C++, Java, etc.), has a grammar. Formally, a grammar G can be viewed as a 4-tuple, (V, T, P, S), where V is the set of variables, also called nonterminals or syntactic categories, each of which represents a set of strings, T is a finite set of symbols, also called terminals, that form the strings of the language being defined, S is the start symbol that represents the language being defined, and P is the finite set of productions or rules that represent the recursive definition of the language [Hopcroft et al. 01].

A production is basically a re-writing rule that consists of a head or left-hand side which is a string of at least one nonterminal and zero or more terminals that is being defined, the production symbol '→', and a body or right-hand side which is a string of zero or more terminals and nonterminals [Hopcroft et al. 01]. The derivation of any string in the language starts from the start symbol. All intermediate stages of the strings resulting from the start symbol in the derivation process are called sentential forms. The

derivation of a string can also be represented in the form of a tree called a parse tree or a derivation tree.

EXAMPLE: What follows is a parse tree for an English language sentence that appears at the leaf nodes.



Noam Chomsky classified grammars into four categories: regular grammars, context-free grammars, context-sensitive grammars, and unrestricted grammars [Hopcroft et al. 01]. If all productions of a grammar have a head of length one, that grammar is called a context-free grammar (CFG). The class of languages generated by context-free grammars is the class of context-free languages. Context-free languages are powerful enough to describe the syntax of programming languages. The LALR(1) grammars (Look Ahead Left to Right) are a special class of context-free grammars that have one token of look ahead. That means that with a single token of look ahead it should be possible to know how to parse from any position in a sentential form back to the start symbol.

During the process of parsing a string, the symbols constituting the string are read left to right using the production rules of the underlying grammar in order to reduce the given string down to the start symbol. While parsing a string, a "shift" action is the process of reading the next symbol and a "reduce" action means that a group of symbols is replaced by another symbol or group of symbols as a result of a match with one of the grammar rules [Johnson et al. 78]. When a parser has a choice to perform either of these two actions, it is called a shift-reduce conflict. On the other hand, if there is a choice of performing two different reductions, it is called a reduce-reduce conflict. The following example illustrates these conflicts.

EXAMPLE: Given the following grammar and the string w = expr – expr,

$$\text{expr} \rightarrow \text{expr} - \text{expr} \mid \text{expr} + \text{expr}$$
$$\text{diff} \rightarrow \text{expr} - \text{expr}$$

there is a choice of performing two reductions resulting in a reduce-reduce conflict.

For the same grammar and the string w = expr – expr – expr, there is a choice of performing a shift or a reduce after reading the second "expr" resulting in a shift-reduce conflict.

2.2 Normal Forms for Context-Free Grammars

If a language is a CFL, it has grammars in some special forms called the normal forms for context-free grammars [Hopcroft et al. 01]. The two normal forms for context-free grammars are Chomsky Normal Form (CNF) and Greibach Normal Form (GNF). These two normal forms are explained in the following two subsections.

2.2.1 Chomsky Normal Form

Every context-free grammar that does not generate the empty string can be transformed into an equivalent one in Chomsky Normal Form. "Equivalent" here means that the two grammars generate the same language. A context-free grammar is in Chomsky Normal Form iff all its production rules are of the form: $A \rightarrow BC$ or $A \rightarrow a$, where A, B, and C are nonterminal symbols and a is a terminal symbol.

We can call the productions whose body comprises of two nonterminals live productions and the ones whose body consists of just one terminal dead productions [Parks 03].

A grammar is self-embedding if for some nonterminal A it is possible that a sentential form containing A can be derived from A, i.e., $A \overset{*}{\Longrightarrow} \alpha A \beta$, where α and β are both non-empty strings [Nederhof 2000]. This property is illustrated in the example discussed below.

Because of the especially simple form of the production rules in Chomsky Normal Form grammars, this normal form has both theoretical and practical implications [Hopcroft et al. 01]. For instance, given a context-free grammar, one can use the Chomsky Normal Form to construct a polynomial-time algorithm to decide whether or not a given string is in the language generated by that grammar. An example of such an algorithm is the Cocke-Younger-Kasami (CYK) algorithm [Hopcroft et al. 01].

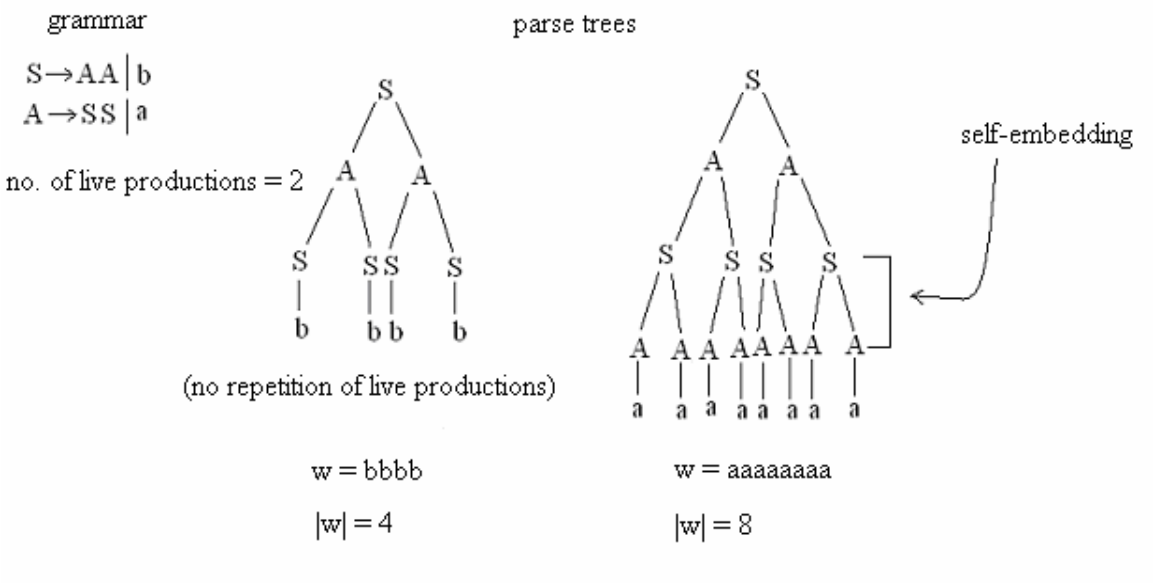
For a grammar in Chomsky Normal Form, the parse tree for a given string is always a binary tree. Let the level of the root be zero and the level at any node be equal to the number of edges present in the path between the root and that node. The following

theorem [Parks 03] states the relation between the number of live productions and maximum possible length of a word that can be derived without any self-embedding.

THEOREM: If G is a context-free grammar in Chomsky Normal Form with p live productions, and if w is a string in L(G) such that the length of w, represented as |w|, is greater than 2^p, then at least one live production must have been used more than once in all possible derivations of w.

PROOF: In a parse tree drawn based on a grammar in Chomsky Normal Form, all internal nodes (except the parents of the leaves) have two children. So, at every level in the parse tree, starting from the root down to the parents of the leaves, there are half as many nodes as in the next level. Hence if $|w| > 2^p$, there must be at least p+1 levels in any parse tree for w, assuming that the level of the root is zero and the leaves have the highest levels. For constructing a parse tree with at least p+1 levels, at least p+1 live productions are necessary. But since there are only p live productions, at least one of them must have been used more than once in deriving w. In other words, a nonterminal on the left-hand-side of one of the live productions will appear at least twice on the path from the root to a leaf.

EXAMPLE: The above theorem states that the longest possible word that can be derived without exhibiting the self-embedding property is 2^p provided that there are p live productions in the grammar. In this example, the self-embedding property is demonstrated for a string length of 8.

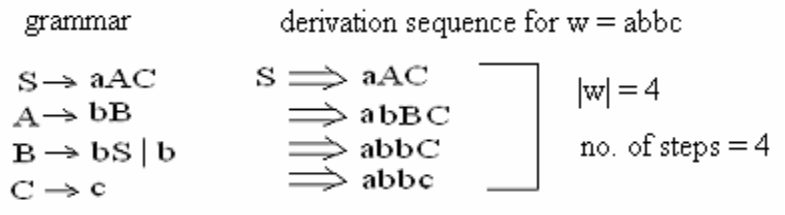


The *productivity* of a context-free grammar can be defined as the number of sentences generated by that grammar without exhibiting the self-embedding property. For example, the productivity of the grammar shown in the above example is five since b, aa, bba, abb, and bbbb are the only sentences that can be derived without exhibiting the self embedding property.

2.2.2 Greibach Normal Form

Every CFL grammar that does not produce the empty string can be converted into a grammar in Greibach Normal Form (GNF) [Hopcroft et al. 01]. A grammar is in Greibach Normal Form iff all of its productions are of the form: $A \rightarrow a\alpha$, where A is a nonterminal, a is a terminal, and α is a string of zero or more nonterminals. Since each application of a production introduces exactly one terminal into a sentential form, a string of length n has a derivation of exactly n steps. The following example illustrates this property.

EXAMPLE:


grammar
derivation sequence for w = abbc
S → aAC
A → bB
B → bS | b
C → c
S ⇒ aAC
⇒ abBC
⇒ abbC
⇒ abbc
|w| = 4
no. of steps = 4

CHAPTER III

AMBIGUITY

In this chapter, the notions of ambiguity, degree of ambiguity, and inherent ambiguity are discussed in some detail.

3.1 Degree of Ambiguity

A CFG G = (V, T, P, S) is ambiguous if there is at least one string w in L(G) for which there are at least two different parse trees, each with its root labeled S and yielding w [Hopcroft et al. 01]. Each parse tree corresponds to a left-most or a right-most derivation. The number of different parse trees of a string w is called the degree of ambiguity of w [Kuich and Salomaa 85]. If no string produced by a grammar G has a degree of ambiguity more than x, the degree of ambiguity of G is x. It is possible to classify ambiguous grammars based on their degree of ambiguity. If the number of distinct parse trees for each string increases with the length of strings generated by a grammar, it is possible that the degree of ambiguity of that grammar is infinite.

EXAMPLE: Given the following ambiguous grammar,

E→E+E
E→E*E
E→ (E)
E→a

there are two left-most derivations for a*a+a

E==>E+E==>E*E+E==>a*E+E==>a*a+E==>a*a+a

E==>E*E==>a*E==>a*E+E==>a*a+E==>a*a+a
Hence, the degree of ambiguity of a*a+a is two.

3.2 Ambiguity Detection and Removal

The problem of deciding whether a given (context-free) grammar for a language is ambiguous is unsolvable [Hopcroft et al. 01]. In other words, there is no general algorithm that can tell us whether a CFG is ambiguous or not. The problem of finding a solution to Post's Correspondence Problem (PCP), which is known to be undecidable, is reducible to the problem of detecting ambiguity in a context-free grammar. Hence, the problem of context-free grammar ambiguity detection is also undecidable.

To handle the ambiguity arising from reduce-shift or reduce-reduce conflicts (see Section 2.1), disambiguating rules can be written [Johnson et al. 78]. Disambiguating rules attempt to remove specific known ambiguities. Disambiguating rules can assign priorities to rules (i.e., which rule to choose when a reduce-reduce conflict occurs) and to operations (i.e., whether to perform a shift or a reduce when a shift-reduce conflict occurs). However, there is no algorithm which, given an ambiguous CFG as input, can always produce an unambiguous context-free grammar as output that generates the same language [Ullian 69].

3.3 Inherent Ambiguity

If all grammars that generate a language are ambiguous, that language is said to be inherently ambiguous [Hopcroft et al. 01]. An ambiguous grammar does not necessarily generate an ambiguous language. In other words, for a language L to be unambiguous, at least one of the grammars that can generate it should be unambiguous.

The problem of determining whether an arbitrary language is inherently ambiguous is recursively unsolvable [Ginsburg and Ullian 66].

In a language L that can be defined as the union of two other languages L_1 and L_2, all sentences in the intersection of the sets L_1 and L_2 have two different interpretations because they belong to both L_1 and L_2. This means that ambiguity is inherent in L, and it is not possible to disambiguate languages such as L. For example, the language $L = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$, with the language $\{a^n b^n c^n d^n \mid n \geq 1\}$ as $L_1 \cap L_2$, is inherently ambiguous.

CHAPTER IV

AMBIGUITY IN OTHER CLASSES OF GRAMMARS

In this chapter, ambiguity in different classes of formal grammars and ambiguity in programming languages are discussed.

4.1 Ambiguity Inherent in Languages

Ambiguity of a grammar cannot always be attributed to the shortcomings of a certain class of grammars or to a badly written grammar. Many languages, including natural languages such as English, have ambiguity inherently. For example, the sentence "House flies like garbage." in English can be interpreted in two different ways: noun-verb-noun or noun-verb-adj-noun. Both of the interpretations are valid but only one of them makes sense [Frank et al. 98].

4.2 Ambiguity in Programming Languages

Since the problems of ambiguity detection and removal are in general unsolvable, one can argue that all programming languages allow statements that are not free from ambiguity. Ambiguity arises due to the power incorporated into the language through the flexibility of the generation of the statements and constructs. Algol60, which can be considered the "mother" of many modern day programming languages, is ambiguous in the way it handles goto statements, for statements, parameters, and several other

constructs [Paulson 81]. The Pascal language has ambiguities in types, sets, and scope rules [Paulson 81].

Extensively used languages like C++ and Java are not free from ambiguity either. C++ supports multiple-inheritance that can potentially cause ambiguity if proper scope resolution is not used [Microsoft 03]. Java has problems with resolving an ambiguous situation arising from multiple definitions of a function in both parent and child classes using the specific syntax shown in the example below [Gacek 05].

EXAMPLE: Given two Java classes PAR and CHD, both defining a function cpy with arguments cpy(class PAR) and cpy(class CHD), the following statements result in ambiguity.

```
CHD child = new CHD()
cpy(child,child); // This statement is ambiguous but its syntax is legal.//
```

4.3 Ambiguity in Two-Level Grammars

Two-level grammars are a class of grammars that have their productions defined in two levels [Cleaveland and Uzgalis 77]. W-grammars (a subclass of two-level grammars as defined by van Wijngaarden [Cleaveland and Uzgalis 77]) are those that have a context-free grammar as their first level grammar which defines the nonterminals of the context-free grammar in their second level grammar [Cleaveland and Uzgalis 77]. This two-level generation capability creates the possibility of infinite number of rules in the resulting context-free grammar. In W-grammars, the rules in the first level are called “proto” productions and the rules at the second level are called the “hyper” productions.

In two-level grammars, in addition to the possibility of existence of ambiguity at each level independently, ambiguity can arise due to the definition of a pattern at both levels [Cleaveland and Uzgalis 77]. In most cases, this type of ambiguity can be avoided using disambiguating rules by setting priorities, resolving conflicts depending on the context, or by predefining the meaning of ambiguous sentences [Cleaveland and Uzgalis 77].

4.4 Ambiguity in Omega Context-Free Grammars

Given an alphabet Σ, an ω-word (omega-word) can be defined as a sequence $a_1a_2\ldots a_n\ldots$, where a_i is a symbol in Σ for all $i \geq 1$. All omega words are of infinite length. The set of all ω-words over an alphabet Σ is denoted by Σ^{ω}. An ω-language over Σ is a subset of Σ^{ω}. If a language L belongs to the ω-Kleene closure of a context-free language, then L is an ω-context-free language [Finkel 03].

The ω-context-free languages can be defined by their acceptance mechanism (i.e., pushdown automata) rather than by their generation mechanism (i.e., grammars). The ω-context-free languages are accepted by Büchi pushdown automata (BPDA) and Muller pushdown automata (MPDA) [Finkel 03]. Acceptance by BPDA or MPDA is based on entering a final state (or states) infinitely many times during a run on an ω-word. An ω-context-free language L is non-ambiguous if there is a unique accepting run on a BPDA or MPDA for all ω-words in L. In other words, an ω-context-free language L is ambiguous if a prefix ($\sigma = a_1a_2\ldots a_n$) of an ω-word ($a_1a_2...a_n\ldots$) in L has more than one accepting run on a BPDA or MPDA. Any BPDA that accepts an ambiguous ω-context-

free language is an ambiguous BPDA. The following are a number of proven facts about ambiguity in ω -context-free languages [Finkel 03].

The class of non-ambiguous ω-context-free languages (NA-CFL_ω) is closed under intersection with ω-regular languages. The class NA-CFL_ω is not closed under finite union, but it is closed under disjoint finite union. The class NA-CFL_ω is strictly included in the ω-Kleene closure of the class of non-ambiguous context-free languages. The class of inherently ambiguous ω-context-free languages is not included in the ω-Kleene closure of the inherently ambiguous context-free languages. It is undecidable whether or not an arbitrary ω -context-free language is ambiguous.

4.5 Ambiguity in Context-Sensitive and Unrestricted Grammars

A grammar that has productions of the form $x \rightarrow y$, where $|x| \leq |y|$, is called a context-sensitive grammar (x, y are strings of terminals and nonterminals, x has at least one nonterminal and |x| and |y| represent the lengths of strings x and y, respectively). Context-sensitive grammars (CSGs) can also be defined as grammars whose productions are of the form $xAy \rightarrow xwy$, where x, y, and w are strings of terminals and nonterminals, and A is a nonterminal [Hopcroft et al. 01]. So, obviously, a context-sensitive grammar whose productions are of the form $xAy \rightarrow xwy$, where x, y are empty strings, is a context-free grammar. Context-sensitive grammars can be used to define natural languages to an extent. They can also be used to develop applications that convert active voice to passive voice [Wilson 02].

Because of the context-sensitiveness, productions of a CSG are generally stricter and also more powerful than those of a context-free grammar. The productions of a CSG

allow for multiple definitions of a sentence and it can be argued that the potential for ambiguity is more in context-sensitive grammars compared to context-free grammars. The class of context-free languages is a subset of the class of context-sensitive languages. So, it is undecidable in general whether or not a CSG is ambiguous. Also, inherently ambiguous context-sensitive languages exist for which it is not possible to write unambiguous grammars. An example for an inherently ambiguous context-sensitive language is $L = \{a^i b^i c^i d^j e^j f^j \cup a^i b^j c^i d^j e^i f^j\}$ because all sentences of the form $a^n b^n c^n d^n e^n f^n$ are ambiguous regardless of how the language L is defined by a context-sensitive grammar.

Unrestricted grammars are defined as grammars whose productions are of the form $x \rightarrow y$, where x and y are strings of terminals and nonterminals and x has at least one nonterminal [Hopcroft et al. 01]. There are no restrictions on the lengths of x and y, and hence, the length of a derivation sequence need not be proportional to the length of a sentence. Unrestricted grammars are not widely used because of their extreme power which makes writing efficient parsers for unrestricted grammars difficult [Wilson 02].

CHAPTER V

ALGORITHMS FOR FINDING AMBIGUOUS STRINGS

In this chapter three algorithms for finding ambiguous strings available in the literature and their scopes are briefly discussed.

5.1 Eickel and Paul's Algorithm

The algorithm proposed by Eickel and Paul [Eickel and Paul 66] is one of the first algorithms to detect the ambiguity of a string. This algorithm first converts a given context-free grammar into Chomsky Normal Form. Then, starting from a given input string, all possible sentential forms that directly derive the given string are considered one at a time. After a number of iterations of the above process of considering the sentential forms that can potentially yield the given string in the derivation process, if identical sentential forms are encountered, the input string is decided to be ambiguous. The limitation of this algorithm is that it only determines whether a given string is ambiguous with respect to a grammar, but it does not determine whether the given grammar itself is ambiguous.

5.2 Algorithm used by YACC (Yet Another Compiler Compiler)

YACC is an LALR (1) parser generator developed at the Bell laboratories [Johnson et al. 78]. YACC detects ambiguity in LALR (1) grammars [Johnson et al. 78]. YACC indicates ambiguity when it encounters reduce-shift conflicts or reduce-reduce

conflicts (see Section 2.1 for a brief explanation of the two types of conflicts) [Johnson et al. 78]. The weakness of this algorithm is that it accepts BNF grammars which are equivalent to context-free grammars, but it only detects ambiguity in LALR(1) grammars, which are a proper subset of context-free grammars.

5.3 Cheung's Algorithm

This algorithm attempts to search a CFG systematically for ambiguity. First a given CFG is converted to GNF, a process that is ambiguity preserving [Cheung 95]. Then it generates strings by applying each grammar rule once. Then it iteratively checks the strings so formed for ambiguity, and longer strings are generated by applying GNF rules again. The significant weakness of this algorithm is that it fails when the string length is not bounded. It means that the algorithm solves only cases with known maximum string lengths.

5.4 Scope of the Algorithms

The algorithms available in the literature that find ambiguous strings are limited in their scope in that they basically check whether or not a given string of bounded length is ambiguous with respect to a given grammar, or check if an LALR(1) grammar is ambiguous. As expected, none of the algorithms stated in this chapter are powerful enough to determine if a given context-free grammar is ambiguous.

CHAPTER VI

PROPOSED APPROACH FOR FINDING AMBIGUOUS STRINGS

In this chapter three variations of a new algorithm to find ambiguous strings in a subset of context-free grammars are presented and the details of their implementation are discussed.

6.1 Description

Non-trivial Grammars define non-finite languages in a recursive manner. That means the sentences in such a language are generated by repetitions of one or more defined patterns. Based on this observation, any context-free language L (or a language in general) can be divided into two groups of strings: those with one or more repetitions of one or more patterns (set A) and the rest of the strings (set B).

For the strings in set A, the repetition of application of a live production in the process of derivation causes the repetition of a pattern in the resulting string. According to the theorem in Section 1.2.2, the length of a string, which does not have any repeated patterns, is at most 2^p, where p is the number of live productions of a given CNF context-free grammar. Hence, by generating the strings of L whose length is less than or equal to 2^p, the set B mentioned above can be created.

The possibility of existence of different parse trees or interpretations for a string is the result of having an ambiguous grammar. Ambiguity is due to the possibility of

deriving a sentence in more than one way, using different sets of productions of the same grammar or the same set of productions of the same grammar in a different order.

If all the strings in set B mentioned above are unambiguous, there is no possibility that the grammar for language L is ambiguous. That means that the problem of determining whether or not a context-free grammar (in CNF) for a language L is ambiguous comes down to the problem of determining whether or not the strings in the corresponding set B are ambiguous.

6.2 Three Variations of an Algorithm That Finds Ambiguous Strings

The proposed algorithm to find ambiguous strings generated by a context-free grammar in CNF can functionally be viewed as having two parts. The first part generates the set B (mentioned in Section 6.1) for a given grammar, and the second part determines if at least one of the strings in set B can be parsed in more than one way. In order to increase the efficiency of this process of finding ambiguous strings, instead of generating all strings of set B and then checking for ambiguity, each string is checked for ambiguity as soon as it is generated. This way, some of the time for generating strings in set B can be saved because as soon as an ambiguous string is found, the remaining strings in set B need not be generated. The process described above is shown as an algorithm below.

```
// A function to convert a CFG into CNF.
Function ToCNF(CFG G)
Begin

        Remove nongenerating symbols from G
        Remove unreachable symbols from G
        Remove epsilon productions from G
        Remove unit productions from G
```

```
        Re-write all rules so that the RHS of all productions is always a terminal
        or two nonterminals
End

// A function to calculate the maximum length of a string that is to be checked
// for ambiguity.
Function CalcSize(CFG in CNF G')
Begin
        Integer n = 0
        For all productions of G' Do
                If the length of RHS of the production is two Then
                        Increment n by 1
                End If
        End For
        Integer size = 2 power n
        Return n
End

// A function to generate strings and check for ambiguity.
Function GenerateStrings(G')
Begin
        Array SententialFormsPending
        Insert Start Symbol into SententialFormsPending
        While there are more sentential forms in the Array
                Apply productions considering the next sentential form in the
                Array SententialFormsPending
                If there is a chance of applying other productions then
                        insert the sentential form in to the Array and also remember
                        the productions that were applied last time so that there are
                        no repetitions. Also, before inserting the sentential form
                        into the array, make sure its length is less than or equal to
                        CalcSize(G')
                End If
                If a sentence is derived then
                        If the length is less than or equal to CalcSize(G') then
                                Check if the same string has been generated before
                                using another derivation sequence.
                                If so then
                                        Check if both derivations are left most or if
                                        both of them are rightmost
                                        If so then
                                                Print "Ambiguous string found"
                                                Return
                                        End If
                                End If
                                Remember the string and its derivation sequence
```

```
                End If
            End If
            Store the sentential form in the array after making sure its length is
            less than or equal to CalcSize(G')
        End While
        Print "Not Ambiguous"
End

// The main function that reads a grammar and checks for ambiguity.
Main()
Begin
        Read CFG G
        ToCNF(G)
        GenerateStrings(G')
End
```

In order to improve on the time to detect ambiguous strings, sentential forms are checked for ambiguity instead of waiting until a sentence is generated. Only the sentential forms that are capable of generating sentences whose lengths are less than or equal to the required size are considered. This is achieved by a small change in the function GenerateStrings as shown below.

```
// A function to check for ambiguity using sentential forms.
    Function GenerateStrings(G')
    Begin
        Array SententialFormsPending
        Insert Start Symbol into SententialFormsPending
        While there are more sentential forms in the Array
            Apply productions considering the next sentential form in the
            Array SententialFormsPending
            If there is a chance of applying other productions then
                insert the sentential Form in to the Array and also
                remember the productions that were applied last time so
                that there are no repetitions. Also before inserting the
                sentential form into the array, make sure its length is less
                than or equal to CalcSize(G')
            End If
            If the length is less than or equal to CalcSize(G') then
                Check if the same string has been generated before using
                another derivation sequence.
```

```
                    If so then
                        Check if both derivations are left most or if both of
                        them are rightmost
                        If so then
                            Print "Ambiguous"
                            Return
                        End If
                    Else
                        Remember the sentential form and its derivation
                        sequence
                        Store the sentential form in SententialFormsPending
                    End If
                End If
            End While
            Print "Not Ambiguous"
End
```

This modified approach is based on the fact that if a sentential form is ambiguous then so are the sentential forms and the sentences that are derived from it. Since the grammars are first converted to CNF, there are no useless symbols, which means that every sentential form leads to a sentence. Also, if a sentence is ambiguous, there is a sentential form in the derivation sequence that is ambiguous unless the ambiguity arises in the last step of derivation sequence. Since the variation shown above checks the sentences in addition to the sentential forms for ambiguity, ambiguity arising in the last step of a derivation sequence can also be captured.

Another way to reduce the time taken for ambiguous string detection by the algorithm presented above would be to use the productions from the proper grammar corresponding to the given context-free grammar to generate the strings instead of using the productions from the corresponding CNF grammar. This is based on an observation that the length of a derivation sequence for a sentence using productions of a proper grammar is shorter than the derivation length of the same sentence using the

corresponding CNF productions. An algorithm for this process is the same as the algorithm on pages 21 through 23 except that proper grammar productions are used instead of the corresponding CNF grammar productions to derive strings that are checked for ambiguity.

Although proper grammar productions are used to generate strings, a CFG needs to be converted to CNF to calculate the maximum length of a string to be considered for ambiguity checking. So the time taken to convert a grammar to CNF is ignored.

6.3 Observations and Results

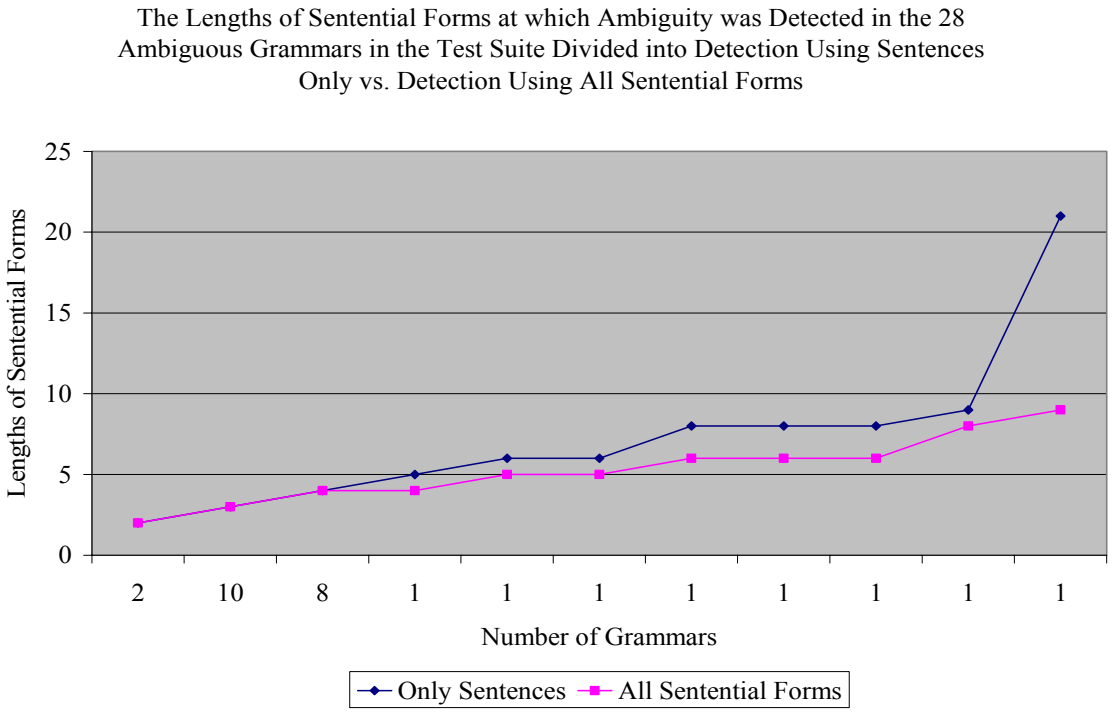
Various context-free grammars were used to test the proposed algorithm. The test suite included grammars of different sizes (varying number of productions), ambiguous grammars, unambiguous grammars, and grammars that generate known inherently ambiguous languages. The test suite consisted of 50 grammars, of which 28 were known to be ambiguous and 22 were unambiguous. All three variations of the presented algorithm yielded correct results. Table 1 below shows the execution times of the implementations of the above three variations of the presented algorithm.

Type of Grammars / Variation	Average over 28 Ambiguous Grammars	Average over 22 Unambiguous Grammars
CNF Grammar (Sentences)	0.527 Seconds	27.230 Seconds
CNF Grammar (Sentential Forms)	0.038 Seconds	18.944 Seconds
Proper Grammar (Sentences)	0.042 Seconds	1.054 Seconds

Table 1: Time Taken to Determine the Ambiguity of a Grammar

The individual execution times required to determine the ambiguity of each grammar is the difference between two variables that hold the times before and after a grammar is checked for ambiguity by the program. The execution platform was the author's personal computer. To ensure the stability of the outputs, the programs were executed three times for each grammar.

It can be observed from Table 1 that checking the sentential forms for ambiguity yields results faster than checking only the sentences for ambiguity for the grammars (both ambiguous and unambiguous) in the test suite. This means that the execution time taken for the process of checking whether or not sentential forms are sentences and then checking sentences for ambiguity, is more than that of the process of checking all sentential forms for ambiguity. The lengths of sentential forms at which the ambiguity is caught, using these two techniques is shown in the following figure.



As the results from Table 1 show, the time taken by the third algorithm (which uses proper grammar productions to derive sentences) to declare that an input unambiguous grammar is unambiguous is better than the other two approaches. But using proper grammar is not as good a technique (compared to the technique of detecting ambiguity in sentential forms) when it comes to the ambiguous grammars in the test suite. Detecting ambiguity at the strict sentential form stage (i.e., when the derived strings contain terminal as well as nonterminal symbols) appears to be better than waiting for a sentence to be derived. Moreover, the advantage of checking such sentential forms for ambiguity is that it shows exactly where the ambiguity arises.

It is intuitive to expect the ambiguity to show up faster if the degree of ambiguity of the grammar under consideration is higher. However, when the program was tested on a suite of 30 ambiguous grammars (10 grammars each with degrees of ambiguity 2, 3, and 4), the results did not show any relation between the degree of ambiguity and the execution time required to detect ambiguous strings.

An effort was made to find out if the degree of ambiguity together with the productivity of a grammar have anything to do with how fast an ambiguous string could be detected. The productivity of a grammar, as used in this thesis, is the number of sentences a grammar can generate without repetition of application of its productions. Table 4 on page 39 lists the productivity of a grammar, the degree of ambiguity of the first encountered ambiguous sentence, and the execution time required to find ambiguous strings for a set of 30 ambiguous grammars. The values from the table indicate that the productivity of a grammar is generally independent of the execution time in the context

of the grammars in the test suite. Also, it is not always true that ambiguity can be detected faster if the ambiguous strings appear at smaller lengths.

The results from the implementation of the proposed algorithms show that the algorithms yield correct results when tested on grammars which are known to be ambiguous or unambiguous. But for an arbitrary context-free grammar, the problem of detecting ambiguity is undecidable. That means that even though the proposed algorithm halts and gives an output of whether a given arbitrary grammar is ambiguous or not, the result is not provable in general.

Attempts to fail the algorithm on an input grammar, so that the class of grammars for which the algorithm works can be studied, were not fruitful. In other words, the subset of context-free grammars for which the algorithm works is unknown. The time complexity of the presented algorithm is $O(2^n)$, where n is the number of productions of the input context-free grammar.

CHAPTER VII

SUMMARY, CONCLUSION AND FUTURE WORK

In this chapter a brief summary of this thesis is given, a number of conclusions and observations are discussed, and the future work that can be done in this area is outlined.

7.1 Summary

The concepts of grammars and ambiguity are briefly introduced in Chapter I. Chapter II contains a discussion of context-free grammars and parse trees. Chomsky Normal Form and Greibach Normal Form are also introduced. In Chapter III, the concepts of ambiguity and inherent ambiguity are discussed. Ambiguity in other classes of grammars and in some programming languages is discussed in Chapter IV. In Chapter V, three algorithms from literature for finding ambiguous strings are discussed. Three variations of an algorithm to detect ambiguous strings are presented in Chapter VI.

This thesis aimed at studying the ambiguity of context-free grammars and ambiguity in general. Ambiguity in different classes of grammars was studied. Some algorithms present in the literature for finding ambiguity of sentences and bounded grammars were studied. An algorithm was proposed to detect ambiguous strings in a subclass of context-free grammars. Attempts to characterize this subclass of CFGs were unfruitful. The proposed algorithm is based on an observation that the source of ambiguity cannot be mere repetitions of applications of the productions of a context-

free grammar. The implementation of this algorithm paves a way for studying the possibility of a relationship between the degree of ambiguity of ambiguous strings and the execution time required to find ambiguous strings generated by that grammar.

7.2 Observations and Conclusions

There are formal languages that are inherently ambiguous. Inherently ambiguous languages can be found in the class of context-free, context-sensitive, and unrestricted languages. Natural languages such as English have inherent ambiguity stemming from their flexible grammatical constructs. Programming languages from Algol and Pascal to C++ and Java support constructs that can potentially lead to ambiguity. Degree of ambiguity of a grammar is an upper bound on the number of distinct parse trees for any sentence generated by that grammar and grammars with infinite degree of ambiguity exist.

The following observations have been made from the implementation of the proposed algorithm and running it on a test suite of 50 grammars. The time taken to find an ambiguous string generated by a grammar in CNF is independent of the degree of ambiguity of the first encountered ambiguous string. The time taken to find ambiguous strings is also independent of the productivity of the grammar. That means that the process of ambiguity detection, as suggested by this prototype study, does not depend on the rate at which a grammar produces strings. Yet another way to express the observation is that the rate at which a grammar produces ambiguous strings need not be proportional to the rate at which the grammar generates strings. Detecting ambiguity in sentential forms generated using productions in the CNF is the fastest way to find ambiguous

strings generated by an input ambiguous grammar. On the other hand, to show that an input unambiguous grammar is unambiguous, using the productions of a proper grammar to detect ambiguity in the strings yields results faster.

7.3 Future Work

Attempts must be made to formally characterize the subclass of context-free grammars for which the proposed algorithm works. That should pave the way to study the question of ambiguity in CFGs more deeply. A formal proof or a theorem may be developed to prove that the proposed algorithm works correctly. Also, a large test suite of grammars can be gleaned to test the practicality of the proposed algorithm. The implementation of the algorithm might be improved and optimized.

As the proposed algorithm suggests, if new ambiguous strings are not produced by mere repetitions of application of the productions of a CFG, then it can be inferred that the ambiguous strings that a grammar generates are predictable in pattern (and thus in length). An algorithm, which captures the frequency at which ambiguous strings are generated, may be developed in future, and, then the ambiguous strings with a desired length can be extrapolated from the basic ambiguous strings (that are derived without exhibiting the self-embedding property). Such an algorithm could be useful to context-free grammar based application developers who maybe interested in finding out whether or not sentences of a certain length generated by a grammar are ambiguous.

Also, similar techniques might prove fruitful for detecting ambiguous strings generated by grammars that are more powerful than CFGs. Taking another look at degree of ambiguity and its growth in ambiguous grammars would be interesting too.

REFERENCES

[Cheung 95] Bruce S. N. Cheung, "Ambiguity in Context-Free Grammars", *Proceedings of the 1995 ACM Symposium on Applied Computing,* pp. 272-276, Nashville, TN, February 1995.

[Cleaveland and Uzgalis 77] J. Craig Cleaveland and Robert C. Uzgalis, *Grammars for Programming Languages*, Elsevier, New York, NY, 1977.

[Eickel and Paul 66] J. Eickel and M. Paul, "The Parsing and Ambiguity Problem for Chomsky Languages", T. B. Steel, Jr. (Ed.), *Formal Language Description Languages for Computer Programming,* North-Holland Publishing Company, London, UK, 1966.

[Finkel 03] Olivier Finkel, "Ambiguity in Omega Context Free Languages", *Theoretical Computer Science,* Vol. 301, No. 1-3, pp. 217-220, May 2003.

[Frank et al. 98] Anette Frank, Tracy H. King, Jonas Kuhn, and John Maxwell, Stanford University, "Optimality Theory Style Constraint Ranking in Large-Scale LFG Grammars", http://csli-publications.stanford.edu/LFG/3/frank-etal/, creation date = 08/11/1998, access date = 20/01/2005.

[Gacek 05] Andrew Gacek, University of Minnesota, IT Lab Forums, "Ambiguous Method Call in Java", http://www.itlabs.umn.edu/HyperNews/get/gopalan/courses/CSCI5161/Spring05/hw4/1.html, creation date = 24/02/2005, access date = 28/02/2005.

[Ginsburg and Ullian 66] Seymour Ginsburg and Joseph Ullian, "Ambiguity in Context Free Languages", *Journal of the ACM (JACM),* Vol. 13, No. 1, pp. 62-89, January 1966.

[Hopcroft et al. 01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Pearson Education Asia, Inc., Delhi, India, 2001.

[Johnson et al. 78] S. C. Johnson, B. W. Kernighan, and M. D. Mcllroy, "YACC: Yet Another Compiler-Compiler", *UNIX Programmer's Manual,* Bell Laboratories, 7th edition, Murray Hill, NJ, 1978.

[Kuich and Salomaa 85] Werner Kuich and Arto Salomaa, *Semirings, Automata, Languages,* Springer-Verlag, London, UK, 1985.

[Microsoft 03] Microsoft Corporation, "TN016: Using C++ Multiple Inheritance with MFC", http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vclib/html/_mfcnotes_tn016.asp, creation date = 19/05/2003, access date = 10/02/2005.

[Nederhof 2000] Mark Jan Nederhof, "Regular Approximation of CFLs: A Grammatical View", *Advances in Probabilistic and Other Parsing Technologies*, pp. 221-241, Kluwer Academic Publishers, Norwell, MA, 2000.

[Parks 03] Dee Parks, Appalachian State University, Lecture Notes, "Chapter 16: Non-Context-Free Languages", http://www.cs.appstate.edu/~dap/classes/2490/chap16.html, creation date = 11/04/2003, access date = 02/01/2004.

[Paulson 81] Lawrence Paulson, Stanford University, "A Compiler Generator for Semantic Grammars", Ph.D. Dissertation Report, Department of Computer Science, http://www.cl.cam.ac.uk/users/lcp/papers/Reports/thesis.pdf, creation date = 12/??/1981, access date = 02/22/2005.

[Prusinkiewicz et al. 88] Przemyslaw Prusinkiewicz, Aristid Lindenmayer, and James Hanan, "Development Models of Herbaceous Plants for Computer Imagery Purposes", *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 141-150, New York, NY, June 1988.

[Rosenberg 70] Arnold L. Rosenberg, "A Note on Ambiguity of Context Free Languages and Presentations of Semilinear Sets", *Journal of the ACM (JACM),* Vol. 17, No. 1, pp. 44-50, January 1970.

[Ullian 69] Joseph S. Ullian, "The Inherent Ambiguity Partial Algorithm Problem for Context Free Languages", *Annual ACM Symposium on Theory of Computing*, pp. 19-20, Marina del Rey, CA, May 1969.

[Wilson 02] Bill Wilson, The University of New South Wales, "Grammars and Parsing", http://www.cse.unsw.edu.au/~billw/cs9414/notes/nlp/grampars.html, creation date = ??/??/2002, Access date = 12/12/2004.

APPENDICES

APPENDIX A

GLOSSARY

Ambiguous	A word, a phrase, or a sentence is called ambiguous if it can be reasonably interpreted in more than one way.
CFG	Context-Free Grammar, a formal grammar in which every production is of the form A→w, where A is a nonterminal and w is a string of zero or more terminals and nonterminals.
CNF	A CFG is in Chomsky Normal Form if all its productions are of the form A→BC or A→a, where A, B, and C are nonterminal symbols and a is a terminal symbol.
Dead Production	A production that has one terminal on the right hand side.
GNF	A grammar is in Greibach Normal Form if all its productions are of the form A→aα, where A is a nonterminal, a is a terminal, and α is a string of zero or more nonterminals.
Grammar	A formal set of symbols and re-writing rules or productions that define a language.
LALR(1)	Look Ahead Left to Right (LALR(1)) is a parsing technique. LALR(1) is also used to refer to a special class of context-free grammars that can accomodate one token of look ahead.
Language	The set of sentences or strings that can be generated by applying the set of productions of a grammar defined over an alphabet starting from a start symbol.
Live Production	A production that has two nonterminals on the right hand side.
Nonterminal	A variable (also called a syntactic category) that can represent a string of zero or more terminals or nonterminals.
Parse Tree	Hierarchical representation of the derivation of a sentence or string in a language.

Productions	Re-writing rules used for specifying how to derive valid syntactic constructs in formal grammars.
Sentence	The result of application of the set of rules or productions of a grammar starting from the start symbol until only terminal characters are present.
Start Symbol	A special nonterminal symbol from which the applications of the rules or productions starts for generation of the sentences or strings of a language.
Terminal	Symbols whose concatenations yield the strings of a language.

APPENDIX B

RESULTS OF THE PROGRAMS

This appendix contains the results of the implementation of the presented algorithm (Appendix C) on the test suite of grammars (Appendix D). Tables 2 and 3 show the execution times of implementations of the three variations of the presented algorithm for ambiguous grammars and unambiguous grammars, respectively. Table 4 shows how execution times vary with degree of ambiguity, productivity, and length.

Sentences-CNF	Sentential Forms-CNF	Sentences-Proper
0.00	0.00	0.01
0.00	0.01	0.00
0.00	0.01	0.00
0.00	0.00	0.01
0.00	0.01	0.00
0.00	0.00	0.01
0.01	0.00	0.00
0.01	0.00	0.00
0.01	0.01	0.00
0.01	0.00	0.00
0.02	0.01	0.01
0.03	0.02	0.01
0.03	0.02	0.01
0.03	0.01	0.00
0.03	0.01	0.02
0.03	0.01	0.02
0.07	0.01	0.00
1.11	0.27	0.48
1.11	0.26	0.47
12.09	0.07	0.02

Table 2: Execution Times (in seconds) of the three variations of the presented algorithm on a number of ambiguous grammars

Sentences-CNF	Sentential Forms-CNF	Sentences-Proper
0.00	0.01	0.00
0.01	0.01	0.00
0.01	0.02	0.01
0.01	0.01	0.00
0.02	0.01	0.00
0.02	0.05	0.00
0.03	0.03	0.04
0.03	0.03	0.01
0.10	0.13	0.03
0.22	0.18	0.06
1.50	3.02	0.02
3.91	4.14	3.94
37.68	30.93	6.88
47.02	35.29	0.32
96.66	48.30	1.62
411.83	294.60	10.26

Table 3: Execution Times (in seconds) of the three variations of the presented algorithm on a number of unambiguous grammars

The three columns in Tables 2 and 3 represent the three variations of the presented algorithm. The first column “Sentences-CNF” represents the variation in which CNF grammar productions were used to generate sentences and only sentences were checked for ambiguity. The second column “Sentential Forms-CNF” represents the variation in which CNF productions were used to generate sentences and sentential forms were checked for ambiguity. The third column “Sentences-Proper” represents the variation in which proper grammar productions were used to generate sentences and sentences were checked for ambiguity. Results with zeroes in all rows were omitted from the tables. The execution times in Tables 2, 3, and 4 are the difference between two times before and after a grammar is checked for ambiguous strings.

DOA	Productivity	Limit	Length	ETime	(c4/c3)*c2	c2/c1	c6/c1
2	58	28	6	0.063	12.43	29.00	6.21
2	7	10	3	0.000	2.10	3.50	1.05
2	11	3	3	0.000	11.00	5.50	5.50
2	8	8	4	0.000	4.00	4.00	2.00
2	11	3	3	0.000	11.00	5.50	5.50
2	14	22	21	0.047	13.36	7.00	6.68
2	21	12	3	0.016	5.25	10.50	2.62
2	10	7	4	0.000	5.71	5.00	2.86
2	16	4	4	0.016	16.00	8.00	8.00
2	11	8	3	0.000	4.12	5.50	2.06
3	20	36	3	0.015	1.67	6.67	0.56
3	3	4	2	0.000	1.50	1.00	0.50
3	11	11	4	0.000	4.00	3.67	1.33
3	14	8	4	0.000	7.00	4.67	2.33
3	15	16	3	0.000	2.81	5.00	0.94
3	14	8	3	0.016	5.25	4.67	1.75
3	21	12	3	0.000	5.25	7.00	1.75
3	30	9	3	0.016	10.00	10.00	3.33
3	6	19	4	0.000	1.26	2.00	0.42
3	8	11	3	0.000	2.18	2.67	0.73
4	19	40	4	0.047	1.90	4.75	0.48
4	4	5	2	0.000	1.60	1.00	0.40
4	12	15	4	0.000	3.20	3.00	0.80
4	15	12	4	0.015	5.00	3.75	1.25
4	16	18	3	0.016	2.67	4.00	0.67
4	18	16	3	0.000	3.37	4.50	0.84
4	22	16	3	0.000	4.12	5.50	1.03
4	12	11	2	0.000	2.18	3.00	0.55
4	7	21	4	0.000	1.33	1.75	0.33
4	8	12	3	0.016	2.00	2.00	0.50

Table 4: Execution Times (in seconds) of the presented algorithm for ambiguous grammars with varied degrees of ambiguity (DOA), productivity, and length

In Table 4, DOA stands for degree of ambiguity of a sentence, productivity stands for the number of sentences generated without exhibiting self-embedding, limit stands for the maximum length of sentence to be checked for ambiguity, and length stands for the

length of an ambiguous string found. The terms c1, c2, c3, c4, and c6 represent columns 1,2,3,4, and 6 in the table, and ETime represents execution time in seconds.

It is intuitive to expect that the productivity of a grammar is directly proportional to the time taken to check for ambiguous strings and that the degree of ambiguity is indirectly proportional to the time taken for ambiguous string detection. Column 7 contains the values of "productivity/degree of ambiguity" for the grammars under consideration so that these values can be examined together with the execution times to find possible relationships among them. Column 6 contains the values of "(length*productivity)/limit" so that the possibility of a relationship between execution time and the length of ambiguous strings, productivity, and limit (maximum length of strings that are checked for ambiguity) can be explored. Column 8 contains the values of column 6 divided by degree of ambiguity. At the time of this study, no relationship has been found between the execution time and the factors considered.

APPENDIX C

SOURCE CODE LISTING

Three programs that check for ambiguous strings generated by CFGs are listed in this appendix. Program1 checks for ambiguity using strings generated by a CNF grammar. Program2 checks for ambiguity using sentential forms without going all the way to sentences. The sentential forms are generated using a CNF grammar. Program3 checks for ambiguity using strings generated by a proper grammar.

Program1

```
/*THIS IS A PROGRAM THAT CONVERTS A GIVEN CONTEXT-FREE GRAMMAR INTO CNF AND CHECKS FOR
ITS AMBIGUITY. THE PROGRAM GENERATES STRING USING THE CNF AND THEN CHECKS IF THE STRING
IS AMBIGUOUS. ALL STRINGS WITH LENGTH LESS THAN OR EQUAL TO 2^p (WHERE p IS THE NUMBER OF
LIVE PRODUCTIONS) ARE CONSIDERED
BEFORE DECIDING THAT A GIVEN GRAMMAR IS AMBIGUOUS.*/

#include<iostream>
#include<fstream>
#include<string>
#include<vector>
#include<time.h>
using namespace std;

/*THIS IS A STRUCTURE THAT DEFINES THE FORMAT OF A GRAMMAR RULE*/
typedef struct gr
{
        string lhs;
        string rhs;
}rule;

/*A STRUCTURE THAT DEFINES THE FORMAT OF A CNF GRAMMAR RULE*/
typedef struct CNFgramRule
{
        string LHS;
        string RHS[2];
}CNFrule;

/*A STRUCTURE THAT CONTAINS A SENTENTIAL FORM AND ITS DERIVATION SEQUENCE*/
typedef struct strHis
{
        string str;
        string history;
}strhis;


vector<rule> grammar;
vector<CNFrule> CNFgrammar;
int limit,unambGram=0,ambGram=0,ambiguityFound,DOA;
double ambTime=0,unambTime=0;
```

```
string startSym,amStr;
clock_t time1;
ofstream out,graph;


/*THIS IS A FUNCTION THAT GENERATES STRINGS TO CHECK FOR AMBIGUITY*/
void genStrings(vector<strhis> *,strhis,int *);

/*THIS FUNCTION CHECKS IF A STRING WITH THE SAME DERIVATION SEQUENCE
HAS ALREADY BEEN CHECKED FOR AMBIGUITY*/
int already(vector<strhis>,strhis,int *);

/*A FUNCTION TO RETRIEVE THE NEXT NONTERMINAL FROM A PRODUCTION BODY*/
string getNextNonter(string,int *,int *);

/*A SIMPLE FUNCTION TO CALCULATE 2^n*/
int twoPower(int);

/*A FUNCTION TO FIND THE NUMBER OF TERMINALS AND NONTERMINALS IN A SENTENTIAL FORM*/
int lengthOf(string,int *);

/*THIS IS A FUNCTION TO REMOVE THE NON GENERATING SYMBOLS FROM A GRAMMAR*/
void removeNonGen(void);

/*A FUNCTION TO CHECK IF A STRING BELONGS TO A VECTOR OF STINGS*/
int belongsTo(string,vector<string>);

/*THIS FUNCTION REMOVES THE UNREACHABLE SYMBOLS FROM A GRAMMAR*/
void removeUnreachable(void);

/*A FUNCTION TO REMOVE EPSILON PRODUCTIONS FROM A GRAMMAR*/
int removeEpsilon(void);

/*THIS FUNCTION CHECKS IF A RULE IS ALREADY PRESENT IN THE SET OF
GRAMMAR RULES BEING CONSIDERED*/
int rulePresent(rule,vector<rule>);

/*FUNCTION TO REMOVE UNIT PRODUCTIONS*/
void removeUnit(void);

/*THIS FUNCTION IS USEDTO CHECK IF A PRODUCTION IS A SINGLE PRODUCTION
OR NOT*/
int singleProd(rule);

/*THIS IS A FUNCTION THAT BREAKS UP A PROPER GRAMMAR INTO A CNF GRAMMAR*/
void toCNF(void);

/*FUNCTION TO CREATE NEW NONTERMINALS DURING THE CONVERSION TO CNF*/
string makeNewNonter(string,vector<rule>,int *);

/*THIS IS A FUNCTION TO CHECK IF A CNF RULE HAS ALREADY BEEN ADDED*/
int cnfRuleAlready(CNFrule);

/*THIS FUNCTION IS USED FOR APPLYING THE RULES TO GENERATE SENTENTIAL
FORMS*/
int presRemStr(string,vector<rule>,string *);

/*TO CHECK IF A STRING IS A SENTENCE OR A SENTENTIAL FORM*/
int isTerStr(string);

/*PROGRAM EXECUTION STARTS HERE*/
int main()
{
        ifstream inp;
        inp.open("grammar.txt");
        out.open("PROFILE.txt");
        if(!inp)
        {
                out<<"INPUT FILE MISSING"<<endl;
                exit(1);
        }
```

```
graph.open("GRAPH.txt");
graph<<"TIME   LENGTH  DOA"<<endl;
while(!inp.eof())
{
        string inpStr;
        getline(inp,inpStr);
        out<<"GRAMMAR NAME: "<<inpStr<<endl;
        getline(inp,inpStr);
        ambiguityFound=0;
        DOA=1;
        while(inpStr!="**END")
        {
                int pos=(int)inpStr.find("->",0);
                rule gramProd;
                gramProd.lhs=inpStr.substr(0,pos);
                gramProd.rhs=inpStr.substr(pos+2,inpStr.length()-pos-1);
                grammar.push_back(gramProd);
                getline(inp,inpStr);
        }
        startSym=grammar[0].lhs;
        if(!removeEpsilon())
        {
                out<<"\t"<<"GRAMMAR DERIVES EMPTY STRING. SO NO CNF"<<endl;

out<<"************************************************"<<endl<<endl;
        }
        else
        {
                removeUnit();
                removeNonGen();
                if(grammar[0].lhs!=startSym)
                {
                        out<<"\t"<<"GRAMMAR DOESN'T PRODUCE ANYTHING"<<endl;

out<<"************************************************"<<endl<<endl;
                }
                else
                {
                        removeUnreachable();
                        if((int)grammar.size()==0)
                        {
                                out<<"\t"<<"GRAMMAR DOESN'T PRODUCE ANYTHING"<<endl;

out<<"************************************************"<<endl<<endl;
                        }
                        else
                        {
                                toCNF();
                                vector<string> LV;
                                int indx;
                                for(indx=0;indx<(int)CNFgrammar.size();indx++)
                                {
                                        if(CNFgrammar[indx].RHS[1]!="")
                                                LV.push_back(CNFgrammar[indx].LHS);
                                }
                                int LR=0,LDL=0;
                                for(indx=0;indx<(int)CNFgrammar.size();indx++)
                                {
                                        if(CNFgrammar[indx].RHS[1]!="")
                                        {

if(CNFgrammar[indx].RHS[0]==CNFgrammar[indx].RHS[1])
                                                {

if(belongsTo(CNFgrammar[indx].RHS[1],LV))
                                                                LDL++;
                                                        else
                                                                LR++;
                                                }
                                                else
                                                        LR++;
```

```
                                        }
                                }
                                limit=twoPower(LDL)*(LR+1);
                                vector<strhis> senForms;
                                strhis tem;
                                tem.str=CNFgrammar[0].LHS;
                                tem.history="";
                                senForms.push_back(tem);
                                int numNew=0,i=1;
                                out<<"\t"<<"MAXIMUM LENGTH OF AN UNAMBIGUOUS STRING:
"<<limit<<endl;
                                out<<"\t"<<"CHECKING FOR AMBIGUITY....."<<endl;
                                time1=clock();
                                int j;
                                while(i<=limit)
                                {
                                        for(j=0;j<(int)senForms.size();j++)
                                                genStrings(&senForms,senForms[j],&i);
                                        if(DOA>1)
                                        {
                                                i=limit+1;
                                                clock_t time2=clock();
                                                out<<"\t"<<"DEGREE OF AMBIGUITY:
"<<DOA<<endl;
                                                out<<"\t"<<"DIFFERENT LEFT MOST
DERIVATIONS ARE :"<<endl;
                                                for(int
N=0;N<(int)senForms.size();N++)
                                                {
                                                        if(senForms[N].str==amStr)

        out<<endl<<"\t"<<senForms[N].history<<senForms[N].str<<endl;
                                                }
                                                out<<endl<<"\t"<<"PROCESSING TIME FOR
GRAMMAR: "<<(time2-time1)/(double)CLOCKS_PER_SEC<<" SECONDS  (THIS TIME INCLUDES THE TIME
TO CALCULATE DEGREE OF AMBIGUITY)"<<endl;
                                                ambTime+=(time2-
time1)/(double)CLOCKS_PER_SEC;

        graph<<(int)amStr.length()<<"\t"<<DOA<<endl;
                                                ambGram++;

        out<<"*****************************************************"<<endl<<endl;
                                                grammar.clear();
                                                CNFgrammar.clear();
                                        }
                                        i++;
                                }
                                if(i!=limit+2)
                                {
                                        out<<"\t"<<"NO AMBIGUOUS STRINGS FOUND AND
HENCE UNAMBIGUOUS GRAMMAR"<<endl;
                                        clock_t time2=clock();
                                        out<<"\t"<<"PROCESSING TIME FOR GRAMMAR:
"<<(time2-time1)/(double)CLOCKS_PER_SEC<<" SECONDS"<<endl;
                                        unambTime+=(time2-
time1)/(double)CLOCKS_PER_SEC;
                                        unambGram++;

        out<<"*************************************************"<<endl<<endl;
                                        grammar.clear();
                                        CNFgrammar.clear();
                                }
                        }
                }
        }
        out<<"TOTAL NUMBER OF AMBIGUOUS GRAMMARS: "<<ambGram<<endl;
        out<<"AVERAGE TIME TAKEN FOR AMBIGUOUS GRAMMARS: "<<ambTime/(double)ambGram<<"
SECONDS"<<endl;
        out<<"TOTAL NUMBER OF UNAMBIGUOUS GRAMMARS: "<<unambGram<<endl;
```

```
	out<<"AVERAGE TIME TAKEN FOR UNAMBIGUOUS GRAMMARS:
"<<unambTime/(double)unambGram<<" SECONDS"<<endl;
}

/*THIS IS A FUNCTION TO REMOVE THE NON GENERATING SYMBOLS FROM A GRAMMAR*/
void removeNonGen(void)
{
	vector<string> Ni,Ne;
	int i,j,k=0;
	do
	{
		Ni=Ne;
		for(i=0;i<(int)grammar.size();i++)
		{
			int good=1;
			for(j=0;j<(int)grammar[i].rhs.length();)
			{
				string t=getNextNonter(grammar[i].rhs,&j,&k);
				if((t!="")&&(!belongsTo(t,Ni)))
					good=0;
			}
			if((good==1)&&(!belongsTo(grammar[i].lhs,Ne)))
					Ne.push_back(grammar[i].lhs);
		}
	}
	while(Ni!=Ne);
	vector<rule> tempGram;
	for(i=0;i<(int)grammar.size();i++)
	{
		if(belongsTo(grammar[i].lhs,Ne))
		{
			int good=1;
			for(j=0;j<(int)grammar[i].rhs.length();)
			{
				string t=getNextNonter(grammar[i].rhs,&j,&k);
				if((t!="")&&(!belongsTo(t,Ne)))
					good=0;
			}
			if(good)
				tempGram.push_back(grammar[i]);
		}
	}
	grammar.clear();
	grammar=tempGram;
}

/*A FUNCTION TO CHECK IF A STRING BELONGS TO A VECTOR OF STINGS*/
int belongsTo(string s,vector<string> v)
{
	int n;
	for(n=0;n<(int)v.size();n++)
	{
		if(s==v[n])
			return 1;
	}
	return 0;
}

/*THIS FUNCTION REMOVES THE UNREACHABLE SYMBOLS FROM A GRAMMAR*/
void removeUnreachable(void)
{
	vector<string> Ni,Ne;
	int i,j,k=0;
	Ne.push_back(grammar[0].lhs);
	do
	{
		Ni=Ne;
		for(i=0;i<(int)grammar.size();i++)
		{
			if(belongsTo(grammar[i].lhs,Ni))
			{
```

```
                                for(j=0;j<(int)grammar[i].rhs.length();)
                                {
                                        string t=getNextNonter(grammar[i].rhs,&j,&k);
                                        if(!belongsTo(t,Ne))
                                                Ne.push_back(t);
                                }
                        }
                }
        }
        while(Ni!=Ne);
        vector<rule> tempGram;
        for(i=0;i<(int)grammar.size();i++)
        {
                if(belongsTo(grammar[i].lhs,Ne))
                {
                        int good=1;
                        for(j=0;j<(int)grammar[i].rhs.length();)
                        {
                                string t=getNextNonter(grammar[i].rhs,&j,&k);
                                if((t!="")&&(!belongsTo(t,Ne)))
                                        good=0;
                        }
                        if(good)
                                tempGram.push_back(grammar[i]);
                }
        }
        grammar.clear();
        grammar=tempGram;
}

/*A FUNCTION TO REMOVE EPSILON PRODUCTIONS FROM A GRAMMAR*/
int removeEpsilon(void)
{
        vector<string> Ni,Ne;
        int i,j,k=0;
        for(i=0;i<(int)grammar.size();i++)
        {
                if(grammar[i].rhs=="")
                        Ne.push_back(grammar[i].lhs);
        }
        while(Ni!=Ne)
        {
                Ni=Ne;
                int ll=(int)Ne.size();
                for(i=0;i<(int)grammar.size();i++)
                {
                        int good=1,len=0,changed=0;
                        for(j=0;j<(int)grammar[i].rhs.length();)
                        {
                                string t=getNextNonter(grammar[i].rhs,&j,&k);
                                if(t!="")
                                {
                                        len++;
                                        if((belongsTo(t,Ni))&&good)
                                        {
                                                good=1;
                                                changed=1;
                                        }
                                        else
                                                good=0;
                                }
                        }
                        int dummy=0;
                        if((len==lengthOf(grammar[i].rhs,&dummy)) && good && changed &&
(!belongsTo(grammar[i].lhs,Ne)))
                                Ne.push_back(grammar[i].lhs);
                }
        }
        for(i=0;i<(int)grammar.size();i++)
        {
                for(j=0;j<(int)grammar[i].rhs.length();)
```

```
		{
			string t=getNextNonter(grammar[i].rhs,&j,&k);
			if((t!="")&&(belongsTo(t,Ne)))
			{
				string
newString=grammar[i].rhs.substr(0,k)+grammar[i].rhs.substr(j,(int)grammar[i].rhs.length()
-k-(int)t.length());
				rule newRule;
				newRule.lhs=grammar[i].lhs;
				newRule.rhs=newString;
				if(!rulePresent(newRule,grammar))
					grammar.push_back(newRule);
			}
		}
	}
	vector<rule> tempGram;
	for(i=0;i<(int)grammar.size();i++)
	{
		if((grammar[i].lhs==startSym)&&(grammar[i].rhs==""))
			return 0;
		if((grammar[i].rhs!="")&&(!rulePresent(grammar[i],tempGram)))
			tempGram.push_back(grammar[i]);
	}
	grammar.clear();
	grammar=tempGram;
	return 1;
}

/*THIS FUNCTION CHECKS IF A RULE IS ALREADY PRESENT IN THE SET OF
GRAMMAR RULES BEING CONSIDERED*/
int rulePresent(rule r,vector<rule> vec)
{
	int z;
	for(z=0;z<(int)vec.size();z++)
	{
		if((r.lhs==vec[z].lhs)&&(r.rhs==vec[z].rhs))
			return 1;
	}
	return 0;
}

/*FUNCTION TO REMOVE UNIT PRODUCTIONS*/
void removeUnit(void)
{
	vector<string> nonTers;
	int i,j,k=0,n;
	for(i=0;i<(int)grammar.size();i++)
	{
		if(!belongsTo(grammar[i].lhs,nonTers))
			nonTers.push_back(grammar[i].lhs);
		for(j=0;j<(int)grammar[i].rhs.length();)
		{
			string t=getNextNonter(grammar[i].rhs,&j,&k);
			if((t!="")&&(!belongsTo(t,nonTers)))
				nonTers.push_back(t);
		}
	}
	typedef vector<string> pr;
	vector<pr> singleProds;
	for(n=0;n<(int)nonTers.size();n++)
	{
		vector<string> Ni,Ne;
		Ne.push_back(nonTers[n]);
		while(Ni!=Ne)
		{
			Ni=Ne;
			for(i=0;i<(int)grammar.size();i++)
			{
				if(singleProd(grammar[i]))
				{
					if(belongsTo(grammar[i].lhs,Ni))
```

```
                                        {
                                                if(!belongsTo(grammar[i].rhs,Ne))
                                                {
                                                        Ne.push_back(grammar[i].rhs);
                                                }
                                        }
                                }
                        }
                        singleProds.push_back(Ne);
                }
                vector<rule> tempGram;
                for(n=0;n<(int)nonTers.size();n++)
                {
                        for(i=0;i<(int)grammar.size();i++)
                        {

                if((belongsTo(grammar[i].lhs,singleProds[n]))&&(!singleProd(grammar[i])))
                                {
                                        rule nr;
                                        nr.lhs=nonTers[n];
                                        nr.rhs=grammar[i].rhs;
                                        if(!rulePresent(nr,tempGram))
                                                tempGram.push_back(nr);
                                }
                        }
                }
                grammar.clear();
                grammar=tempGram;
}

/*THIS FUNCTION IS USEDTO CHECK IF A PRODUCTION IS A SINGLE PRODUCTION OR NOT*/
int singleProd(rule r)
{
        int dummy=0;
        if((lengthOf(r.rhs,&dummy)==1)&&((int)r.rhs[0]>64)&&((int)r.rhs[0]<91))
                return 1;
        else
                return 0;
}

/*THIS IS A FUNCTION THAT BREAKS UP A PROPER GRAMMAR INTO A CNF GRAMMAR*/
void toCNF(void)
{
        int i,count=1;
        vector<rule> remGram,remString;
        for(i=0;i<(int)grammar.size();i++)
        {
                if((int)grammar[i].rhs.length()==1)
                {
                        CNFrule newCNFrule;
                        newCNFrule.LHS=grammar[i].lhs;
                        newCNFrule.RHS[0]=grammar[i].rhs;
                        newCNFrule.RHS[1]="";
                        CNFgrammar.push_back(newCNFrule);
                }
                else
                {
                        int numNonters=0,len;
                        len=lengthOf(grammar[i].rhs,&numNonters);
                        if(len==2)
                        {
                                if(numNonters==2)
                                {
                                        CNFrule newCNFrule;
                                        newCNFrule.LHS=grammar[i].lhs;
                                        int j=0,k=0;

        newCNFrule.RHS[0]=getNextNonter(grammar[i].rhs,&j,&k);

        newCNFrule.RHS[1]=getNextNonter(grammar[i].rhs,&j,&k);
```

```
                                CNFgrammar.push_back(newCNFrule);
                        }
                        else
                        {
                                if(numNonters==1)
                                {
                                        CNFrule newCNFrule;
                                        newCNFrule.LHS=grammar[i].lhs;
                                        int j=0,k=0;
                                        string ter,nonter,newNT;
                                        nonter=getNextNonter(grammar[i].rhs,&j,&k);
                                        if(k==0)
                                        {
                                                newCNFrule.RHS[0]=nonter;

ter=grammar[i].rhs.substr((int)nonter.length(),1);

newNT=makeNewNonter(ter,remGram,&count);
                                                newCNFrule.RHS[1]=newNT;
                                                CNFgrammar.push_back(newCNFrule);

                                        }
                                        else
                                        {
                                                newCNFrule.RHS[1]=nonter;
                                                ter=grammar[i].rhs.substr(0,1);

newNT=makeNewNonter(ter,remGram,&count);
                                                newCNFrule.RHS[0]=newNT;
                                                CNFgrammar.push_back(newCNFrule);
                                        }
                                        newCNFrule.LHS=newNT;
                                        newCNFrule.RHS[0]=ter;
                                        newCNFrule.RHS[1]="";
                                        if(!cnfRuleAlready(newCNFrule))
                                                CNFgrammar.push_back(newCNFrule);
                                        rule nr;
                                        nr.lhs=newNT;
                                        nr.rhs=ter;
                                        if(!rulePresent(nr,remGram))
                                                remGram.push_back(nr);
                                }
                                else
                                {
                                        CNFrule newCNFrule;
                                        string newNT1,newNT2,ter1,ter2;
                                        newCNFrule.LHS=grammar[i].lhs;
                                        ter1=grammar[i].rhs.substr(0,1);
                                        ter2=grammar[i].rhs.substr(1,1);
                                        newNT1=makeNewNonter(ter1,remGram,&count);
                                        rule nr;
                                        nr.lhs=newNT1;
                                        nr.rhs=ter1;
                                        if(!rulePresent(nr,remGram))
                                                remGram.push_back(nr);
                                        newNT2=makeNewNonter(ter2,remGram,&count);
                                        newCNFrule.RHS[0]=newNT1;
                                        newCNFrule.RHS[1]=newNT2;
                                        CNFgrammar.push_back(newCNFrule);
                                        newCNFrule.LHS=newNT1;
                                        newCNFrule.RHS[0]=ter1;
                                        newCNFrule.RHS[1]="";
                                        if(!cnfRuleAlready(newCNFrule))
                                                CNFgrammar.push_back(newCNFrule);
                                        newCNFrule.LHS=newNT2;
                                        newCNFrule.RHS[0]=ter2;
                                        newCNFrule.RHS[1]="";
                                        if(!cnfRuleAlready(newCNFrule))
                                                CNFgrammar.push_back(newCNFrule);
                                        nr.lhs=newNT2;
                                        nr.rhs=ter2;
```

```
                                        if(!rulePresent(nr,remGram))
                                                remGram.push_back(nr);
                                }
                        }
                }
                else
                {
                        rule curRule;
                        curRule=grammar[i];
                        int dummy=0;
                        while(lengthOf(curRule.rhs,&dummy)!=2)
                        {
                                string rep="";
                                if(presRemStr(curRule.rhs,remString,&rep))
                                        curRule.rhs=rep;
                                else
                                {
                                        rule newRule,dumRule;
                                        dumRule.lhs=curRule.rhs;
                                        newRule.lhs=curRule.lhs;
                                        string nonter,symbol,newNT;
                                        int j=0,k=0;
                                        newNT=makeNewNonter("dummy",remGram,&count);
                                        curRule.lhs=newNT;
                                        nonter=getNextNonter(curRule.rhs,&j,&k);
                                        if((k==0)&&(nonter!=""))
                                        {
                                                newRule.rhs=nonter;
                                                newRule.rhs+=newNT;

        curRule.rhs=curRule.rhs.substr((int)nonter.length(),(int)curRule.rhs.length()-
(int)nonter.length());
                                        }
                                        else
                                        {
                                                newRule.rhs=curRule.rhs.substr(0,1);
                                                newRule.rhs+=newNT;

        curRule.rhs=curRule.rhs.substr(1,(int)curRule.rhs.length()-1);
                                        }
                                        dumRule.rhs=newRule.rhs;
                                        remString.push_back(dumRule);
                                        grammar.push_back(newRule);
                                }
                                grammar.push_back(curRule);
                        }
                }
        }
        }
}

/*FUNCTION TO CREATE NEW NONTERMINALS DURING THE CONVERSION TO CNF*/
string makeNewNonter(string str,vector<rule> vec,int *cnt)
{
        int indx;
        for(indx=0;indx<(int)vec.size();indx++)
        {
                if(str==vec[indx].rhs)
                        return vec[indx].lhs;
        }
        char temp[10];
        sprintf(temp,"Z%d",(*cnt));
        (*cnt)++;
        return temp;
}

/*THIS IS A FUNCTION TO CHECK IF A CNF RULE HAS ALREADY BEEN ADDED*/
int cnfRuleAlready(CNFrule cr)
{
        int indx;
        for(indx=0;indx<(int)CNFgrammar.size();indx++)
```

```
        {

        if((cr.LHS==CNFgrammar[indx].LHS)&&(cr.RHS[0]==CNFgrammar[indx].RHS[0])&&(cr.RHS[1
]==CNFgrammar[indx].RHS[1]))
                        return 1;
        }
        return 0;
}

/*THIS FUNCTION IS USED FOR APPLYING THE RULES TO GENERATE SENTENTIAL FORMS*/
int presRemStr(string st,vector<rule> v,string *ret)
{
        int indx;
        for(indx=0;indx<(int)v.size();indx++)
        {
                if(st==v[indx].lhs)
                {
                        (*ret)=v[indx].rhs;
                        return 1;
                }
        }
        return 0;
}

/*THIS IS A FUNCTION THAT GENERATES STRINGS TO CHECK FOR AMBIGUITY*/
void genStrings(vector<strhis> *vec,strhis x,int *y)
{
        int prods,ps=0,p=0,amb;
        string newString,NT;
        NT=getNextNonter(x.str,&p,&ps);
        for(prods=0;prods<(int)CNFgrammar.size();prods++)
        {
                if((CNFgrammar[prods].LHS==NT)&&(NT!=""))
                {

        newString=x.str.substr(0,ps)+CNFgrammar[prods].RHS[0]+CNFgrammar[prods].RHS[1]+x.s
tr.substr(p,(int)x.str.length()-ps-(int)NT.length());
                        int dumy;
                        if(lengthOf(newString,&dumy)<=(*y))
                        {
                                strhis tem;
                                tem.str=newString;
                                tem.history=x.history+x.str+"==>";
                                amb=1;
                 if(!already((*vec),tem,&amb))
                                  vec->push_back(tem);
                                if(amb>1)
                                {
                                        if((ambiguityFound==0)&&(isTerStr(newString)))
                                        {
                                 ambiguityFound=1;
                                                amStr=newString;
                                                clock_t time2=clock();
                                                out<<"\t"<<"AMBIGUOUS SENTENTIAL FORM
"<<newString<<" FOUND AT LENGTH: "<<(int)newString.length()<<endl;
                                                out<<"\t"<<"TIME TAKEN TO DETECT AMBIGUITY:
"<<(time2-time1)/(double)CLOCKS_PER_SEC<<" SECONDS"<<endl;
                                                graph<<(time2-
time1)/(double)CLOCKS_PER_SEC<<"\t";
                                        }
                                        if((ambiguityFound==1)&&(newString==amStr)&&(amb>1))
                                                DOA++;
                                }
                        }
                }
        }
}

/*THIS FUNCTION CHECKS IF A STRING WITH THE SAME DERIVATION SEQUENCE HAS ALREADY BEEN
CHECKED FOR AMBIGUITY*/
int already(vector<strhis> st,strhis s,int *am)
```

```
{
        int x,retval=0;
        for(x=0;x<(int)st.size();x++)
        {
                if(s.str==st[x].str)
                {
                        if(s.history==st[x].history)
                 retval=1;
                        else
                                (*am)++;
                }
        }
        return retval;
}

/*A FUNCTION TO RETRIEVE THE NEXT NONTERMINAL FROM A PRODUCTION BODY*/
string getNextNonter(string s,int *z,int *y)
{
        while(((*z)<(int)s.length())&&((int)s[(*z)]<65)||((int)s[(*z)]>90))
                (*z)++;
        if((*z)==(int)s.length())
                return "";
        else
        {
                (*y)=(*z);
                (*z)++;
                while(((*z)<(int)s.length())&&((int)s[(*z)]<=57)&&((int)s[(*z)]>=48))
                        (*z)++;
                return(s.substr((*y),(*z)-(*y)));
        }
}

/*A SIMPLE FUNCTION TO CALCULATE 2^n*/
int twoPower(int n)
{
        int indx,val=1;
        for(indx=1;indx<=n;indx++)
                val*=2;
        return val;
}

/*A FUNCTION TO FIND THE NUMBER OF TERMINALS AND NONTERMINALS IN A SENTENTIAL FORM*/
int lengthOf(string str,int *nN)
{
        int len=0,indx;
        for(indx=0;indx<(int)str.length();indx++)
        {
                if((str[indx]<48)||(str[indx]>57))
                        len++;
                if((str[indx]>64)&&(str[indx]<91))
                        (*nN)++;
        }
        return len;
}

/*TO CHECK IF A STRING IS A SENTENCE OR A SENTENTIAL FORM*/
int isTerStr(string str)
{
        int x;
        for(x=0;x<(int)str.length();x++)
        {

        if((((int)str[x]<91)&&((int)str[x]>64))||(((int)str[x]<58)&&((int)str[x]>47)))
                        return 0;
        }
        return 1;
}
```

Program2

```
/*THIS IS A PROGRAM THAT CONVERTS A GIVEN CONTEXT-FREE GRAMMAR INTO CNF AND CHECKS FOR
ITS AMBIGUITY. THE PROGRAM GENERATES SENTENTIAL FORMS USING THE CNF AND THEN CHECKS IF
THEY ARE AMBIGUOUS. ALL SENTENTIAL FORMS WHICH CAN PRODUCE
A SENTENCE LESS THAN OR EQUAL TO 2^p (WHERE p IS THE NUMBER OF LIVE PRODUCTIONS) ARE
CONSIDERED BEFORE DECIDING THAT A GIVEN GRAMMAR IS AMBIGUOUS.*/
#include<iostream>
#include<fstream>
#include<string>
#include<vector>
#include<time.h>
using namespace std;

/*THIS IS A STRUCTURE THAT DEFINES THE FORMAT OF A GRAMMAR RULE*/
typedef struct gr
{
        string lhs;
        string rhs;
}rule;

/*A STRUCTURE THAT DEFINES THE FORMAT OF A CNF GRAMMAR RULE*/
typedef struct CNFgramRule
{
        string LHS;
        string RHS[2];
}CNFrule;

/*A STRUCTURE THAT CONTAINS A SENTENTIAL FORM AND ITS DERIVATION SEQUENCE*/
typedef struct strHis
{
        string str;
        string history;
}strhis;


vector<rule> grammar;
vector<CNFrule> CNFgrammar;
int limit,unambGram=0,ambGram=0,ambiguityFound,DOA;
double ambTime=0,unambTime=0;
string startSym,amStr;
clock_t time1;
ofstream out,graph;


/*THIS IS A FUNCTION THAT GENERATES STRINGS TO CHECK FOR AMBIGUITY*/
void genStrings(vector<strhis> *,strhis,int *);

/*THIS FUNCTION CHECKS IF A STRING WITH THE SAME DERIVATION SEQUENCE HAS ALREADY BEEN
CHECKED FOR AMBIGUITY*/
int already(vector<strhis>,strhis,int *);

/*A FUNCTION TO RETRIEVE THE NEXT NONTERMINAL FROM A PRODUCTION BODY*/
string getNextNonter(string,int *,int *);

/*A SIMPLE FUNCTION TO CALCULATE 2^n*/
int twoPower(int);

/*A FUNCTION TO FIND THE NUMBER OF TERMINALS AND NONTERMINALS IN A SENTENTIAL FORM*/
int lengthOf(string,int *);

/*THIS IS A FUNCTION TO REMOVE THE NON GENERATING SYMBOLS FROM A GRAMMAR*/
void removeNonGen(void);

/*A FUNCTION TO CHECK IF A STRING BELONGS TO A VECTOR OF STINGS*/
int belongsTo(string,vector<string>);

/*THIS FUNCTION REMOVES THE UNREACHABLE SYMBOLS FROM A GRAMMAR*/
void removeUnreachable(void);
```

```
/*A FUNCTION TO REMOVE EPSILON PRODUCTIONS FROM A GRAMMAR*/
int removeEpsilon(void);

/*THIS FUNCTION CHECKS IF A RULE IS ALREADY PRESENT IN THE SET OF
GRAMMAR RULES BEING CONSIDERED*/
int rulePresent(rule,vector<rule>);

/*FUNCTION TO REMOVE UNIT PRODUCTIONS*/
void removeUnit(void);

/*THIS FUNCTION IS USEDTO CHECK IF A PRODUCTION IS A SINGLE PRODUCTION
OR NOT*/
int singleProd(rule);

/*THIS IS A FUNCTION THAT BREAKS UP A PROPER GRAMMAR INTO A CNF GRAMMAR*/
void toCNF(void);

/*FUNCTION TO CREATE NEW NONTERMINALS DURING THE CONVERSION TO CNF*/
string makeNewNonter(string,vector<rule>,int *);

/*THIS IS A FUNCTION TO CHECK IF A CNF RULE HAS ALREADY BEEN ADDED*/
int cnfRuleAlready(CNFrule);

/*THIS FUNCTION IS USED FOR APPLYING THE RULES TO GENERATE SENTENTIAL
FORMS*/
int presRemStr(string,vector<rule>,string *);

/*THIS FUNCTION CHECKS IF A SENTENTIAL FORM GENERATES A STRING WITHIN THE LIMIT OF 2^p OR
NOT*/
int strInLimit(string,int);

/*PROGRAM EXECUTION STARTS HERE*/
int main()
{
        ifstream inp;
        inp.open("grammar.txt");
        out.open("PROFILE.txt");
        if(!inp)
        {
                out<<"INPUT FILE MISSING"<<endl;
                exit(1);
        }
        graph.open("GRAPH.txt");
        graph<<"TIME   LENGTH  DOA"<<endl;
        while(!inp.eof())
        {
                string inpStr;
                getline(inp,inpStr);
                out<<"GRAMMAR NAME: "<<inpStr<<endl;
                getline(inp,inpStr);
                ambiguityFound=0;
                DOA=1;
                while(inpStr!="**END")
                {
                        int pos=(int)inpStr.find("->",0);
                        rule gramProd;
                        gramProd.lhs=inpStr.substr(0,pos);
                        gramProd.rhs=inpStr.substr(pos+2,inpStr.length()-pos-1);
                        grammar.push_back(gramProd);
                        getline(inp,inpStr);
                }
                startSym=grammar[0].lhs;
                if(!removeEpsilon())
                {
                        out<<"\t"<<"GRAMMAR DERIVES EMPTY STRING. SO NO CNF"<<endl;

        out<<"**********************************************"<<endl<<endl;
                }
                else
                {
                        removeUnit();
```

```
                        removeNonGen();
                        if(grammar[0].lhs!=startSym)
                        {
                                out<<"\t"<<"GRAMMAR DOESN'T PRODUCE ANYTHING"<<endl;

        out<<"************************************************"<<endl<<endl;
                        }
                        else
                        {
                                removeUnreachable();
                                if((int)grammar.size()==0)
                                {
                                        out<<"\t"<<"GRAMMAR DOESN'T PRODUCE ANYTHING"<<endl;

        out<<"************************************************"<<endl<<endl;
                                }
                                else
                                {
                                        toCNF();
                                        vector<string> LV;
                                        int indx;
                                        for(indx=0;indx<(int)CNFgrammar.size();indx++)
                                        {
                                                if(CNFgrammar[indx].RHS[1]!="")
                                                        LV.push_back(CNFgrammar[indx].LHS);
                                        }
                                        int LR=0,LDL=0;
                                        for(indx=0;indx<(int)CNFgrammar.size();indx++)
                                        {
                                                if(CNFgrammar[indx].RHS[1]!="")
                                                {

        if(CNFgrammar[indx].RHS[0]==CNFgrammar[indx].RHS[1])
                                                        {

        if(belongsTo(CNFgrammar[indx].RHS[1],LV))
                                                                        LDL++;
                                                                else
                                                                        LR++;
                                                        }
                                                        else
                                                                LR++;
                                                }
                                        }
                                        limit=twoPower(LDL)*(LR+1);
                                        vector<strhis> senForms;
                                        strhis tem;
                                        tem.str=CNFgrammar[0].LHS;
                                        tem.history="";
                                        senForms.push_back(tem);
                                        int numNew=0,i=1;
                                        out<<"\t"<<"MAXIMUM LENGTH OF AN UNAMBIGUOUS STRING:
"<<limit<<endl;
                                        out<<"\t"<<"CHECKING FOR AMBIGUITY....."<<endl;
                                        time1=clock();
                                        int j;
                                        while(i<=limit)
                                        {
                                                for(j=0;j<(int)senForms.size();j++)
                                                        genStrings(&senForms,senForms[j],&i);
                                                if(DOA>1)
                                                {
                                                        i=limit+1;
                                                        clock_t time2=clock();
                                                        out<<"\t"<<"DEGREE OF AMBIGUITY:
"<<DOA<<endl;
                                                        out<<"\t"<<"DIFFERENT LEFT MOST
DERIVATIONS ARE :"<<endl;
                                                        for(int
N=0;N<(int)senForms.size();N++)
                                                        {
```

```
                                                        if(senForms[N].str==amStr)

        out<<endl<<"\t"<<senForms[N].history<<senForms[N].str<<endl;
                                                        }
                                                        out<<endl<<"\t"<<"PROCESSING TIME FOR
GRAMMAR: "<<(time2-time1)/(double)CLOCKS_PER_SEC<<" SECONDS  (THIS TIME INCLUDES THE TIME
TO CALCULATE DEGREE OF AMBIGUITY)"<<endl;
                                                        ambTime+=(time2-
time1)/(double)CLOCKS_PER_SEC;

        int zz=0;
        graph<<lengthOf(amStr,&zz)<<"\t"<<DOA<<endl;
                                                        ambGram++;

        out<<"*****************************************************"<<endl<<endl;
                                                        grammar.clear();
                                                        CNFgrammar.clear();
                                                }
                                                i++;
                                        }
                                        if(i!=limit+2)
                                        {
                                                out<<"\t"<<"NO AMBIGUOUS STRINGS FOUND AND
HENCE UNAMBIGUOUS GRAMMAR"<<endl;
                                                clock_t time2=clock();
                                                out<<"\t"<<"PROCESSING TIME FOR GRAMMAR:
"<<(time2-time1)/(double)CLOCKS_PER_SEC<<" SECONDS"<<endl;
                                                unambTime+=(time2-
time1)/(double)CLOCKS_PER_SEC;
                                                unambGram++;

        out<<"***********************************************"<<endl<<endl;
                                                grammar.clear();
                                                CNFgrammar.clear();
                                        }
                                }
                        }
                }
        }
        out<<"TOTAL NUMBER OF AMBIGUOUS GRAMMARS: "<<ambGram<<endl;
        out<<"AVERAGE TIME TAKEN FOR AMBIGUOUS GRAMMARS: "<<ambTime/(double)ambGram<<"
SECONDS"<<endl;
        out<<"TOTAL NUMBER OF UNAMBIGUOUS GRAMMARS: "<<unambGram<<endl;
        out<<"AVERAGE TIME TAKEN FOR UNAMBIGUOUS GRAMMARS:
"<<unambTime/(double)unambGram<<" SECONDS"<<endl;
}


/*THIS IS A FUNCTION TO REMOVE THE NON GENERATING SYMBOLS FROM A GRAMMAR*/
void removeNonGen(void)
{
        vector<string> Ni,Ne;
        int i,j,k=0;
        do
        {
                Ni=Ne;
                for(i=0;i<(int)grammar.size();i++)
                {
                        int good=1;
                        for(j=0;j<(int)grammar[i].rhs.length();)
                        {
                                string t=getNextNonter(grammar[i].rhs,&j,&k);
                                if((t!="")&&(!belongsTo(t,Ni)))
                                        good=0;
                        }
                        if((good==1)&&(!belongsTo(grammar[i].lhs,Ne)))
                                        Ne.push_back(grammar[i].lhs);
                }
        }
        while(Ni!=Ne);
        vector<rule> tempGram;
```

```
	for(i=0;i<(int)grammar.size();i++)
	{
		if(belongsTo(grammar[i].lhs,Ne))
		{
			int good=1;
			for(j=0;j<(int)grammar[i].rhs.length();)
			{
				string t=getNextNonter(grammar[i].rhs,&j,&k);
				if((t!="")&&(!belongsTo(t,Ne)))
					good=0;
			}
			if(good)
				tempGram.push_back(grammar[i]);
		}
	}
	grammar.clear();
	grammar=tempGram;
}

/*A FUNCTION TO CHECK IF A STRING BELONGS TO A VECTOR OF STINGS*/
int belongsTo(string s,vector<string> v)
{
	int n;
	for(n=0;n<(int)v.size();n++)
	{
		if(s==v[n])
			return 1;
	}
	return 0;
}

/*THIS FUNCTION REMOVES THE UNREACHABLE SYMBOLS FROM A GRAMMAR*/
void removeUnreachable(void)
{
	vector<string> Ni,Ne;
	int i,j,k=0;
	Ne.push_back(grammar[0].lhs);
	do
	{
		Ni=Ne;
		for(i=0;i<(int)grammar.size();i++)
		{
			if(belongsTo(grammar[i].lhs,Ni))
			{
				for(j=0;j<(int)grammar[i].rhs.length();)
				{
					string t=getNextNonter(grammar[i].rhs,&j,&k);
					if(!belongsTo(t,Ne))
						Ne.push_back(t);
				}
			}
		}
	}
	while(Ni!=Ne);
	vector<rule> tempGram;
	for(i=0;i<(int)grammar.size();i++)
	{
		if(belongsTo(grammar[i].lhs,Ne))
		{
			int good=1;
			for(j=0;j<(int)grammar[i].rhs.length();)
			{
				string t=getNextNonter(grammar[i].rhs,&j,&k);
				if((t!="")&&(!belongsTo(t,Ne)))
					good=0;
			}
			if(good)
				tempGram.push_back(grammar[i]);
		}
	}
	grammar.clear();
```

```
	grammar=tempGram;
}

/*A FUNCTION TO REMOVE EPSILON PRODUCTIONS FROM A GRAMMAR*/
int removeEpsilon(void)
{
	vector<string> Ni,Ne;
	int i,j,k=0;
	for(i=0;i<(int)grammar.size();i++)
	{
		if(grammar[i].rhs=="")
			Ne.push_back(grammar[i].lhs);
	}
	while(Ni!=Ne)
	{
		Ni=Ne;
		int ll=(int)Ne.size();
		for(i=0;i<(int)grammar.size();i++)
		{
			int good=1,len=0,changed=0;
			for(j=0;j<(int)grammar[i].rhs.length();)
			{
				string t=getNextNonter(grammar[i].rhs,&j,&k);
				if(t!="")
				{
					len++;
					if((belongsTo(t,Ni))&&good)
					{
						good=1;
						changed=1;
					}
					else
						good=0;
				}
			}
			int dummy=0;
			if((len==lengthOf(grammar[i].rhs,&dummy)) && good && changed &&
(!belongsTo(grammar[i].lhs,Ne)))
				Ne.push_back(grammar[i].lhs);
		}
	}
	for(i=0;i<(int)grammar.size();i++)
	{
		for(j=0;j<(int)grammar[i].rhs.length();)
		{
			string t=getNextNonter(grammar[i].rhs,&j,&k);
			if((t!="")&&(belongsTo(t,Ne)))
			{
				string
newString=grammar[i].rhs.substr(0,k)+grammar[i].rhs.substr(j,(int)grammar[i].rhs.length()
-k-(int)t.length());
				rule newRule;
				newRule.lhs=grammar[i].lhs;
				newRule.rhs=newString;
				if(!rulePresent(newRule,grammar))
					grammar.push_back(newRule);
			}
		}
	}
	vector<rule> tempGram;
	for(i=0;i<(int)grammar.size();i++)
	{
		if((grammar[i].lhs==startSym)&&(grammar[i].rhs==""))
			return 0;
		if((grammar[i].rhs!="")&&(!rulePresent(grammar[i],tempGram)))
			tempGram.push_back(grammar[i]);
	}
	grammar.clear();
	grammar=tempGram;
	return 1;
}
```

```
/*THIS FUNCTION CHECKS IF A RULE IS ALREADY PRESENT IN THE SET OF
GRAMMAR RULES BEING CONSIDERED*/
int rulePresent(rule r,vector<rule> vec)
{
	int z;
	for(z=0;z<(int)vec.size();z++)
	{
		if((r.lhs==vec[z].lhs)&&(r.rhs==vec[z].rhs))
			return 1;
	}
	return 0;
}

/*FUNCTION TO REMOVE UNIT PRODUCTIONS*/
void removeUnit(void)
{
	vector<string> nonTers;
	int i,j,k=0,n;
	for(i=0;i<(int)grammar.size();i++)
	{
		if(!belongsTo(grammar[i].lhs,nonTers))
			nonTers.push_back(grammar[i].lhs);
		for(j=0;j<(int)grammar[i].rhs.length();)
		{
			string t=getNextNonter(grammar[i].rhs,&j,&k);
			if((t!="")&&(!belongsTo(t,nonTers)))
				nonTers.push_back(t);
		}
	}
	typedef vector<string> pr;
	vector<pr> singleProds;
	for(n=0;n<(int)nonTers.size();n++)
	{
		vector<string> Ni,Ne;
		Ne.push_back(nonTers[n]);
		while(Ni!=Ne)
		{
			Ni=Ne;
			for(i=0;i<(int)grammar.size();i++)
			{
				if(singleProd(grammar[i]))
				{
					if(belongsTo(grammar[i].lhs,Ni))
					{
						if(!belongsTo(grammar[i].rhs,Ne))
						{
							Ne.push_back(grammar[i].rhs);
						}
					}
				}
			}
		}
		singleProds.push_back(Ne);
	}
	vector<rule> tempGram;
	for(n=0;n<(int)nonTers.size();n++)
	{
		for(i=0;i<(int)grammar.size();i++)
		{

	if((belongsTo(grammar[i].lhs,singleProds[n]))&&(!singleProd(grammar[i])))
			{
				rule nr;
				nr.lhs=nonTers[n];
				nr.rhs=grammar[i].rhs;
				if(!rulePresent(nr,tempGram))
					tempGram.push_back(nr);
			}
		}
	}
```

```
	grammar.clear();
	grammar=tempGram;
}

/*THIS FUNCTION IS USEDTO CHECK IF A PRODUCTION IS A SINGLE PRODUCTION OR NOT*/
int singleProd(rule r)
{
	int dummy=0;
	if((lengthOf(r.rhs,&dummy)==1)&&((int)r.rhs[0]>64)&&((int)r.rhs[0]<91))
		return 1;
	else
		return 0;
}

/*THIS IS A FUNCTION THAT BREAKS UP A PROPER GRAMMAR INTO A CNF GRAMMAR*/
void toCNF(void)
{
	int i,count=1;
	vector<rule> remGram,remString;
	for(i=0;i<(int)grammar.size();i++)
	{
		if((int)grammar[i].rhs.length()==1)
		{
			CNFrule newCNFrule;
			newCNFrule.LHS=grammar[i].lhs;
			newCNFrule.RHS[0]=grammar[i].rhs;
			newCNFrule.RHS[1]="";
			CNFgrammar.push_back(newCNFrule);
		}
		else
		{
			int numNonters=0,len;
			len=lengthOf(grammar[i].rhs,&numNonters);
			if(len==2)
			{
				if(numNonters==2)
				{
					CNFrule newCNFrule;
					newCNFrule.LHS=grammar[i].lhs;
					int j=0,k=0;

	newCNFrule.RHS[0]=getNextNonter(grammar[i].rhs,&j,&k);

	newCNFrule.RHS[1]=getNextNonter(grammar[i].rhs,&j,&k);
					CNFgrammar.push_back(newCNFrule);
				}
				else
				{
					if(numNonters==1)
					{
						CNFrule newCNFrule;
						newCNFrule.LHS=grammar[i].lhs;
						int j=0,k=0;
						string ter,nonter,newNT;
						nonter=getNextNonter(grammar[i].rhs,&j,&k);
						if(k==0)
						{
							newCNFrule.RHS[0]=nonter;

	ter=grammar[i].rhs.substr((int)nonter.length(),1);

	newNT=makeNewNonter(ter,remGram,&count);
							newCNFrule.RHS[1]=newNT;
							CNFgrammar.push_back(newCNFrule);

						}
						else
						{
							newCNFrule.RHS[1]=nonter;
							ter=grammar[i].rhs.substr(0,1);
```

```
newNT=makeNewNonter(ter,remGram,&count);
                                                newCNFrule.RHS[0]=newNT;
                                                CNFgrammar.push_back(newCNFrule);
                                        }
                                        newCNFrule.LHS=newNT;
                                        newCNFrule.RHS[0]=ter;
                                        newCNFrule.RHS[1]="";
                                        if(!cnfRuleAlready(newCNFrule))
                                                CNFgrammar.push_back(newCNFrule);
                                        rule nr;
                                        nr.lhs=newNT;
                                        nr.rhs=ter;
                                        if(!rulePresent(nr,remGram))
                                                remGram.push_back(nr);
                                }
                                else
                                {
                                        CNFrule newCNFrule;
                                        string newNT1,newNT2,ter1,ter2;
                                        newCNFrule.LHS=grammar[i].lhs;
                                        ter1=grammar[i].rhs.substr(0,1);
                                        ter2=grammar[i].rhs.substr(1,1);
                                        newNT1=makeNewNonter(ter1,remGram,&count);
                                        rule nr;
                                        nr.lhs=newNT1;
                                        nr.rhs=ter1;
                                        if(!rulePresent(nr,remGram))
                                                remGram.push_back(nr);
                                        newNT2=makeNewNonter(ter2,remGram,&count);
                                        newCNFrule.RHS[0]=newNT1;
                                        newCNFrule.RHS[1]=newNT2;
                                        CNFgrammar.push_back(newCNFrule);
                                        newCNFrule.LHS=newNT1;
                                        newCNFrule.RHS[0]=ter1;
                                        newCNFrule.RHS[1]="";
                                        if(!cnfRuleAlready(newCNFrule))
                                                CNFgrammar.push_back(newCNFrule);
                                        newCNFrule.LHS=newNT2;
                                        newCNFrule.RHS[0]=ter2;
                                        newCNFrule.RHS[1]="";
                                        if(!cnfRuleAlready(newCNFrule))
                                                CNFgrammar.push_back(newCNFrule);
                                        nr.lhs=newNT2;
                                        nr.rhs=ter2;
                                        if(!rulePresent(nr,remGram))
                                                remGram.push_back(nr);
                                }
                        }
                }
                else
                {
                        rule curRule;
                        curRule=grammar[i];
                        int dummy=0;
                        while(lengthOf(curRule.rhs,&dummy)!=2)
                        {
                                string rep="";
                                if(presRemStr(curRule.rhs,remString,&rep))
                                        curRule.rhs=rep;
                                else
                                {
                                        rule newRule,dumRule;
                                        dumRule.lhs=curRule.rhs;
                                        newRule.lhs=curRule.lhs;
                                        string nonter,symbol,newNT;
                                        int j=0,k=0;
                                        newNT=makeNewNonter("dummy",remGram,&count);
                                        curRule.lhs=newNT;
                                        nonter=getNextNonter(curRule.rhs,&j,&k);
                                        if((k==0)&&(nonter!=""))
```

```
                                                {
                                                        newRule.rhs=nonter;
                                                        newRule.rhs+=newNT;

        curRule.rhs=curRule.rhs.substr((int)nonter.length(),(int)curRule.rhs.length()-
(int)nonter.length());
                                                }
                                                else
                                                {
                                                        newRule.rhs=curRule.rhs.substr(0,1);
                                                        newRule.rhs+=newNT;

        curRule.rhs=curRule.rhs.substr(1,(int)curRule.rhs.length()-1);
                                                }
                                                dumRule.rhs=newRule.rhs;
                                                remString.push_back(dumRule);
                                                grammar.push_back(newRule);
                                        }
                                        grammar.push_back(curRule);
                                }
                        }
                }
        }
}

/*FUNCTION TO CREATE NEW NONTERMINALS DURING THE CONVERSION TO CNF*/
string makeNewNonter(string str,vector<rule> vec,int *cnt)
{
        int indx;
        for(indx=0;indx<(int)vec.size();indx++)
        {
                if(str==vec[indx].rhs)
                        return vec[indx].lhs;
        }
        char temp[10];
        sprintf(temp,"Z%d",(*cnt));
        (*cnt)++;
        return temp;
}

/*THIS IS A FUNCTION TO CHECK IF A CNF RULE HAS ALREADY BEEN ADDED*/
int cnfRuleAlready(CNFrule cr)
{
        int indx;
        for(indx=0;indx<(int)CNFgrammar.size();indx++)
        {

        if((cr.LHS==CNFgrammar[indx].LHS)&&(cr.RHS[0]==CNFgrammar[indx].RHS[0])&&(cr.RHS[1
]==CNFgrammar[indx].RHS[1]))
                        return 1;
        }
        return 0;
}

/*THIS FUNCTION IS USED FOR APPLYING THE RULES TO GENERATE SENTENTIAL FORMS*/
int presRemStr(string st,vector<rule> v,string *ret)
{
        int indx;
        for(indx=0;indx<(int)v.size();indx++)
        {
                if(st==v[indx].lhs)
                {
                        (*ret)=v[indx].rhs;
                        return 1;
                }
        }
        return 0;
}

/*THIS IS A FUNCTION THAT GENERATES SENTENTIAL FORMS TO CHECK FOR AMBIGUITY*/
void genStrings(vector<strhis> *vec,strhis x,int *y)
```

```
{
        int prods,ps=0,p=0,amb;
        string newString,NT;
        NT=getNextNonter(x.str,&p,&ps);
        for(prods=0;prods<(int)CNFgrammar.size();prods++)
        {
                if((CNFgrammar[prods].LHS==NT)&&(NT!=""))
                {

        newString=x.str.substr(0,ps)+CNFgrammar[prods].RHS[0]+CNFgrammar[prods].RHS[1]+x.s
tr.substr(p,(int)x.str.length()-ps-(int)NT.length());
                        int dumy;
                        if((lengthOf(newString,&dumy)<=(*y))&&(strInLimit(newString,(*y))))
                        {
                                strhis tem;
                                tem.str=newString;
                                tem.history=x.history+x.str+"==>";
                                amb=1;
                 if(!already((*vec),tem,&amb))
                                 vec->push_back(tem);
                                if(amb>1)
                                {
                                        if(ambiguityFound==0)
                                        {
                                 ambiguityFound=1;
                                                amStr=newString;
                                                DOA=amb;
                                                clock_t time2=clock();
                                                out<<"\t"<<"AMBIGUOUS SENTENTIAL FORM
"<<newString<<" FOUND AT LENGTH: ";
int zz=0;out<<lengthOf(newString,&zz)<<endl;
                                                out<<"\t"<<"TIME TAKEN TO DETECT AMBIGUITY:
"<<(time2-time1)/(double)CLOCKS_PER_SEC<<" SECONDS"<<endl;
                                                graph<<(time2-
time1)/(double)CLOCKS_PER_SEC<<"\t";
                                        }
                                        if((ambiguityFound==1)&&(newString==amStr))
                                                DOA=amb;
                                }
                        }
                }
        }
}


/*THIS FUNCTION CHECKS IF A STRING WITH THE SAME DERIVATION SEQUENCE HAS ALREADY BEEN
CHECKED FOR AMBIGUITY*/
int already(vector<strhis> st,strhis s,int *am)
{
        int x,retval=0;
        for(x=0;x<(int)st.size();x++)
        {
                if(s.str==st[x].str)
                {
                        if(s.history==st[x].history)
                 retval=1;
                        else
                                (*am)++;
                }
        }
        return retval;
}

/*A FUNCTION TO RETRIEVE THE NEXT NONTERMINAL FROM A PRODUCTION BODY*/
string getNextNonter(string s,int *z,int *y)
{
        while(((*z)<(int)s.length())&&((int)s[(*z)]<65)||((int)s[(*z)]>90))
                (*z)++;
        if((*z)==(int)s.length())
                return "";
        else
```

```
        {
                (*y)=(*z);
                (*z)++;
                while(((*z)<(int)s.length())&&((int)s[(*z)]<=57)&&((int)s[(*z)]>=48))
                        (*z)++;
                return(s.substr((*y),(*z)-(*y)));
        }
}

/*A SIMPLE FUNCTION TO CALCULATE 2^n*/
int twoPower(int n)
{
        int indx,val=1;
        for(indx=1;indx<=n;indx++)
                val*=2;
        return val;
}

/*A FUNCTION TO FIND THE NUMBER OF TERMINALS AND NONTERMINALS IN A SENTENTIAL FORM*/
int lengthOf(string str,int *nN)
{
        int len=0,indx;
        for(indx=0;indx<(int)str.length();indx++)
        {
                if((str[indx]<48)||(str[indx]>57))
                        len++;
                if((str[indx]>64)&&(str[indx]<91))
                        (*nN)++;
        }
        return len;
}

/*THIS FUNCTION CHECKS IF A SENTENTIAL FORM GENERATES A STRING WITHIN THE LIMIT OF 2^p OR
NOT*/
int strInLimit(string str,int lt)
{
        int dum=0,num=0;
        for(int p=0;p<(int)str.length();)
        {
                int isLive=0,isDead=0;
                string nonTer=getNextNonter(str,&p,&dum);
                for(int q=0;q<(int)CNFgrammar.size();q++)
                {
                        if((CNFgrammar[q].RHS[1]!="")&&(CNFgrammar[q].LHS==nonTer))
                                isLive=1;
                        if((CNFgrammar[q].RHS[1]=="")&&(CNFgrammar[q].LHS==nonTer))
                                isDead=1;
                }
                if(isLive &&(!isDead))
                        num++;
        }
        if((lengthOf(str,&dum)+num)<=lt)
                return 1;
        else
                return 0;
}
```

Program3

```
/*THIS IS A PROGRAM THAT CONVERTS A GIVEN CONTEXT-FREE GRAMMAR INTO CNF AND CHECKS FOR
ITS AMBIGUITY. THE PROGRAM GENERATES STRING USING THE PROPER GRAMMAR AND THEN CHECKS IF
THE STRING IS AMBIGUOUS. ALL STRINGS WITH LENGTH LESS THAN
OR EQUAL TO 2^p (WHERE p IS THE NUMBER OF LIVE PRODUCTIONS) ARE CONSIDERED BEFORE
DECIDING THAT A GIVEN GRAMMAR IS AMBIGUOUS.*/
#include<iostream>
#include<fstream>
#include<string>
#include<vector>
```

```
#include<time.h>
using namespace std;

/*THIS IS A STRUCTURE THAT DEFINES THE FORMAT OF A GRAMMAR RULE*/
typedef struct gr
{
        string lhs;
        string rhs;
}rule;

/*A STRUCTURE THAT DEFINES THE FORMAT OF A CNF GRAMMAR RULE*/
typedef struct CNFgramRule
{
        string LHS;
        string RHS[2];
}CNFrule;

/*A STRUCTURE THAT CONTAINS A SENTENTIAL FORM AND ITS DERIVATION SEQUENCE*/
typedef struct strHis
{
        string str;
        string history;
}strhis;


vector<rule> grammar;
vector<CNFrule> CNFgrammar;
int limit,unambGram=0,ambGram=0,ambiguityFound,DOA;
double ambTime=0,unambTime=0;
string startSym,amStr;
clock_t time1;
ofstream out,graph;


/*THIS IS A FUNCTION THAT GENERATES STRINGS TO CHECK FOR AMBIGUITY*/
void genStrings(vector<strhis> *,strhis,int *);

/*THIS FUNCTION CHECKS IF A STRING WITH THE SAME DERIVATION SEQUENCE HAS ALREADY BEEN
CHECKED FOR AMBIGUITY*/
int already(vector<strhis>,strhis,int *);

/*A FUNCTION TO RETRIEVE THE NEXT NONTERMINAL FROM A PRODUCTION BODY*/
string getNextNonter(string,int *,int *);

/*A SIMPLE FUNCTION TO CALCULATE 2^n*/
int twoPower(int);

/*A FUNCTION TO FIND THE NUMBER OF TERMINALS AND NONTERMINALS IN A SENTENTIAL FORM*/
int lengthOf(string,int *);

/*THIS IS A FUNCTION TO REMOVE THE NON GENERATING SYMBOLS FROM A GRAMMAR*/
void removeNonGen(void);

/*A FUNCTION TO CHECK IF A STRING BELONGS TO A VECTOR OF STINGS*/
int belongsTo(string,vector<string>);

/*THIS FUNCTION REMOVES THE UNREACHABLE SYMBOLS FROM A GRAMMAR*/
void removeUnreachable(void);

/*A FUNCTION TO REMOVE EPSILON PRODUCTIONS FROM A GRAMMAR*/
int removeEpsilon(void);

/*THIS FUNCTION CHECKS IF A RULE IS ALREADY PRESENT IN THE SET OF
GRAMMAR RULES BEING CONSIDERED*/
int rulePresent(rule,vector<rule>);

/*FUNCTION TO REMOVE UNIT PRODUCTIONS*/
void removeUnit(void);

/*THIS FUNCTION IS USEDTO CHECK IF A PRODUCTION IS A SINGLE PRODUCTION OR NOT*/
int singleProd(rule);
```

```
/*THIS IS A FUNCTION THAT BREAKS UP A PROPER GRAMMAR INTO A CNF GRAMMAR*/
void toCNF(void);

/*FUNCTION TO CREATE NEW NONTERMINALS DURING THE CONVERSION TO CNF*/
string makeNewNonter(string,vector<rule>,int *);

/*THIS IS A FUNCTION TO CHECK IF A CNF RULE HAS ALREADY BEEN ADDED*/
int cnfRuleAlready(CNFrule);

/*THIS FUNCTION IS USED FOR APPLYING THE RULES TO GENERATE SENTENTIAL FORMS*/
int presRemStr(string,vector<rule>,string *);

/*TO CHECK IF A STRING IS A SENTENCE OR A SENTENTIAL FORM*/
int isTerStr(string);

/*PROGRAM EXECUTION STARTS HERE*/
int main()
{
        ifstream inp;
        inp.open("grammar.txt");
        out.open("PROFILE.txt");
        if(!inp)
        {
                out<<"INPUT FILE MISSING"<<endl;
                exit(1);
        }
        graph.open("GRAPH.txt");
        graph<<"TIME   LENGTH  DOA"<<endl;
        while(!inp.eof())
        {
                string inpStr;
                getline(inp,inpStr);
                out<<"GRAMMAR NAME: "<<inpStr<<endl;
                getline(inp,inpStr);
                ambiguityFound=0;
                DOA=1;
                while(inpStr!="**END")
                {
                        int pos=(int)inpStr.find("->",0);
                        rule gramProd;
                        gramProd.lhs=inpStr.substr(0,pos);
                        gramProd.rhs=inpStr.substr(pos+2,inpStr.length()-pos-1);
                        grammar.push_back(gramProd);
                        getline(inp,inpStr);
                }
                startSym=grammar[0].lhs;
                if(!removeEpsilon())
                {
                        out<<"\t"<<"GRAMMAR DERIVES EMPTY STRING. SO NO CNF"<<endl;

        out<<"************************************************"<<endl<<endl;
                }
                else
                {
                        removeUnit();
                        removeNonGen();
                        if(grammar[0].lhs!=startSym)
                        {
                                out<<"\t"<<"GRAMMAR DOESN'T PRODUCE ANYTHING"<<endl;

        out<<"************************************************"<<endl<<endl;
                        }
                        else
                        {
                                removeUnreachable();
                                if((int)grammar.size()==0)
                                {
                                        out<<"\t"<<"GRAMMAR DOESN'T PRODUCE ANYTHING"<<endl;

        out<<"************************************************"<<endl<<endl;
```

```
                                }
                                else
                                {
                                        properGrammar=grammar;
                                        toCNF();
                                        vector<string> LV;
                                        int indx;
                                        for(indx=0;indx<(int)CNFgrammar.size();indx++)
                                        {
                                                if(CNFgrammar[indx].RHS[1]!="")
                                                        LV.push_back(CNFgrammar[indx].LHS);
                                        }
                                        int LR=0,LDL=0;
                                        for(indx=0;indx<(int)CNFgrammar.size();indx++)
                                        {
                                                if(CNFgrammar[indx].RHS[1]!="")
                                                {

        if(CNFgrammar[indx].RHS[0]==CNFgrammar[indx].RHS[1])
                                                        {

        if(belongsTo(CNFgrammar[indx].RHS[1],LV))
                                                                        LDL++;
                                                                else
                                                                        LR++;
                                                        }
                                                        else
                                                                LR++;
                                                }
                                        }
                                        limit=twoPower(LDL)*(LR+1);
                                        vector<strhis> senForms;
                                        strhis tem;
                                        tem.str=CNFgrammar[0].LHS;
                                        tem.history="";
                                        senForms.push_back(tem);
                                        int numNew=0,i=1;
                                        out<<"\t"<<"MAXIMUM LENGTH OF AN UNAMBIGUOUS STRING:
"<<limit<<endl;
                                        out<<"\t"<<"CHECKING FOR AMBIGUITY....."<<endl;
                                        time1=clock();
                                        int j;
                                        while(i<=limit)
                                        {
                                                for(j=0;j<(int)senForms.size();j++)
                                                        genStrings(&senForms,senForms[j],&i);
                                                if(DOA>1)
                                                {
                                                        i=limit+1;
                                                        clock_t time2=clock();
                                                        out<<"\t"<<"DEGREE OF AMBIGUITY:
"<<DOA<<endl;
                                                        out<<"\t"<<"DIFFERENT LEFT MOST
DERIVATIONS ARE :"<<endl;
                                                        for(int
N=0;N<(int)senForms.size();N++)
                                                        {
                                                                if(senForms[N].str==amStr)

        out<<endl<<"\t"<<senForms[N].history<<senForms[N].str<<endl;
                                                        }
                                                        out<<endl<<"\t"<<"PROCESSING TIME FOR
GRAMMAR: "<<(time2-time1)/(double)CLOCKS_PER_SEC<<" SECONDS  (THIS TIME INCLUDES THE TIME
TO CALCULATE DEGREE OF AMBIGUITY)"<<endl;
                                                        ambTime+=(time2-
time1)/(double)CLOCKS_PER_SEC;

        graph<<(int)amStr.length()<<"\t"<<DOA<<endl;
                                                        ambGram++;

        out<<"*****************************************************"<<endl<<endl;
```

```
                                                grammar.clear();
                                                CNFgrammar.clear();
                                        }
                                        i++;
                                }
                                if(i!=limit+2)
                                {
                                        out<<"\t"<<"NO AMBIGUOUS STRINGS FOUND AND
HENCE UNAMBIGUOUS GRAMMAR"<<endl;
                                        clock_t time2=clock();
                                        out<<"\t"<<"PROCESSING TIME FOR GRAMMAR:
"<<(time2-time1)/(double)CLOCKS_PER_SEC<<" SECONDS"<<endl;
                                        unambTime+=(time2-
time1)/(double)CLOCKS_PER_SEC;
                                        unambGram++;

        out<<"*************************************************"<<endl<<endl;
                                        grammar.clear();
                                        CNFgrammar.clear();
                                }
                        }
                }
        }
        out<<"TOTAL NUMBER OF AMBIGUOUS GRAMMARS: "<<ambGram<<endl;
        out<<"AVERAGE TIME TAKEN FOR AMBIGUOUS GRAMMARS: "<<ambTime/(double)ambGram<<"
SECONDS"<<endl;
        out<<"TOTAL NUMBER OF UNAMBIGUOUS GRAMMARS: "<<unambGram<<endl;
        out<<"AVERAGE TIME TAKEN FOR UNAMBIGUOUS GRAMMARS:
"<<unambTime/(double)unambGram<<" SECONDS"<<endl;
}


/*THIS IS A FUNCTION TO REMOVE THE NON GENERATING SYMBOLS FROM A GRAMMAR*/
void removeNonGen(void)
{
        vector<string> Ni,Ne;
        int i,j,k=0;
        do
        {
                Ni=Ne;
                for(i=0;i<(int)grammar.size();i++)
                {
                        int good=1;
                        for(j=0;j<(int)grammar[i].rhs.length();)
                        {
                                string t=getNextNonter(grammar[i].rhs,&j,&k);
                                if((t!="")&&(!belongsTo(t,Ni)))
                                        good=0;
                        }
                        if((good==1)&&(!belongsTo(grammar[i].lhs,Ne)))
                                        Ne.push_back(grammar[i].lhs);
                }
        }
        while(Ni!=Ne);
        vector<rule> tempGram;
        for(i=0;i<(int)grammar.size();i++)
        {
                if(belongsTo(grammar[i].lhs,Ne))
                {
                        int good=1;
                        for(j=0;j<(int)grammar[i].rhs.length();)
                        {
                                string t=getNextNonter(grammar[i].rhs,&j,&k);
                                if((t!="")&&(!belongsTo(t,Ne)))
                                        good=0;
                        }
                        if(good)
                                tempGram.push_back(grammar[i]);
                }
        }
```

```
	grammar.clear();
	grammar=tempGram;
}

/*A FUNCTION TO CHECK IF A STRING BELONGS TO A VECTOR OF STINGS*/
int belongsTo(string s,vector<string> v)
{
	int n;
	for(n=0;n<(int)v.size();n++)
	{
		if(s==v[n])
			return 1;
	}
	return 0;
}

/*THIS FUNCTION REMOVES THE UNREACHABLE SYMBOLS FROM A GRAMMAR*/
void removeUnreachable(void)
{
	vector<string> Ni,Ne;
	int i,j,k=0;
	Ne.push_back(grammar[0].lhs);
	do
	{
		Ni=Ne;
		for(i=0;i<(int)grammar.size();i++)
		{
			if(belongsTo(grammar[i].lhs,Ni))
			{
				for(j=0;j<(int)grammar[i].rhs.length();)
				{
					string t=getNextNonter(grammar[i].rhs,&j,&k);
					if(!belongsTo(t,Ne))
							Ne.push_back(t);
				}
			}
		}
	}
	while(Ni!=Ne);
	vector<rule> tempGram;
	for(i=0;i<(int)grammar.size();i++)
	{
		if(belongsTo(grammar[i].lhs,Ne))
		{
			int good=1;
			for(j=0;j<(int)grammar[i].rhs.length();)
			{
				string t=getNextNonter(grammar[i].rhs,&j,&k);
				if((t!="")&&(!belongsTo(t,Ne)))
					good=0;
			}
			if(good)
				tempGram.push_back(grammar[i]);
		}
	}
	grammar.clear();
	grammar=tempGram;
}

/*A FUNCTION TO REMOVE EPSILON PRODUCTIONS FROM A GRAMMAR*/
int removeEpsilon(void)
{
	vector<string> Ni,Ne;
	int i,j,k=0;
	for(i=0;i<(int)grammar.size();i++)
	{
		if(grammar[i].rhs=="")
			Ne.push_back(grammar[i].lhs);
	}
	while(Ni!=Ne)
	{
```

```
                Ni=Ne;
                int ll=(int)Ne.size();
                for(i=0;i<(int)grammar.size();i++)
                {
                        int good=1,len=0,changed=0;
                        for(j=0;j<(int)grammar[i].rhs.length();)
                        {
                                string t=getNextNonter(grammar[i].rhs,&j,&k);
                                if(t!="")
                                {
                                        len++;
                                        if((belongsTo(t,Ni))&&good)
                                        {
                                                good=1;
                                                changed=1;
                                        }
                                        else
                                                good=0;
                                }
                        }
                        int dummy=0;
                        if((len==lengthOf(grammar[i].rhs,&dummy)) && good && changed &&
(!belongsTo(grammar[i].lhs,Ne)))
                                Ne.push_back(grammar[i].lhs);
                }
        }
        for(i=0;i<(int)grammar.size();i++)
        {
                for(j=0;j<(int)grammar[i].rhs.length();)
                {
                        string t=getNextNonter(grammar[i].rhs,&j,&k);
                        if((t!="")&&(belongsTo(t,Ne)))
                        {
                                string
newString=grammar[i].rhs.substr(0,k)+grammar[i].rhs.substr(j,(int)grammar[i].rhs.length()
-k-(int)t.length());
                                rule newRule;
                                newRule.lhs=grammar[i].lhs;
                                newRule.rhs=newString;
                                if(!rulePresent(newRule,grammar))
                                        grammar.push_back(newRule);
                        }
                }
        }
        vector<rule> tempGram;
        for(i=0;i<(int)grammar.size();i++)
        {
                if((grammar[i].lhs==startSym)&&(grammar[i].rhs==""))
                        return 0;
                if((grammar[i].rhs!="")&&(!rulePresent(grammar[i],tempGram)))
                        tempGram.push_back(grammar[i]);
        }
        grammar.clear();
        grammar=tempGram;
        return 1;
}

/*THIS FUNCTION CHECKS IF A RULE IS ALREADY PRESENT IN THE SET OF
GRAMMAR RULES BEING CONSIDERED*/
int rulePresent(rule r,vector<rule> vec)
{
        int z;
        for(z=0;z<(int)vec.size();z++)
        {
                if((r.lhs==vec[z].lhs)&&(r.rhs==vec[z].rhs))
                        return 1;
        }
        return 0;
}

/*FUNCTION TO REMOVE UNIT PRODUCTIONS*/
```

```
void removeUnit(void)
{
	vector<string> nonTers;
	int i,j,k=0,n;
	for(i=0;i<(int)grammar.size();i++)
	{
		if(!belongsTo(grammar[i].lhs,nonTers))
			nonTers.push_back(grammar[i].lhs);
		for(j=0;j<(int)grammar[i].rhs.length();)
		{
			string t=getNextNonter(grammar[i].rhs,&j,&k);
			if((t!="")&&(!belongsTo(t,nonTers)))
				nonTers.push_back(t);
		}
	}
	typedef vector<string> pr;
	vector<pr> singleProds;
	for(n=0;n<(int)nonTers.size();n++)
	{
		vector<string> Ni,Ne;
		Ne.push_back(nonTers[n]);
		while(Ni!=Ne)
		{
			Ni=Ne;
			for(i=0;i<(int)grammar.size();i++)
			{
				if(singleProd(grammar[i]))
				{
					if(belongsTo(grammar[i].lhs,Ni))
					{
						if(!belongsTo(grammar[i].rhs,Ne))
						{
							Ne.push_back(grammar[i].rhs);
						}
					}
				}
			}
		}
		singleProds.push_back(Ne);
	}
	vector<rule> tempGram;
	for(n=0;n<(int)nonTers.size();n++)
	{
		for(i=0;i<(int)grammar.size();i++)
		{

	if((belongsTo(grammar[i].lhs,singleProds[n]))&&(!singleProd(grammar[i])))
			{
				rule nr;
				nr.lhs=nonTers[n];
				nr.rhs=grammar[i].rhs;
				if(!rulePresent(nr,tempGram))
					tempGram.push_back(nr);
			}
		}
	}
	grammar.clear();
	grammar=tempGram;
}

/*THIS FUNCTION IS USEDTO CHECK IF A PRODUCTION IS A SINGLE PRODUCTION OR NOT*/
int singleProd(rule r)
{
	int dummy=0;
	if((lengthOf(r.rhs,&dummy)==1)&&((int)r.rhs[0]>64)&&((int)r.rhs[0]<91))
		return 1;
	else
		return 0;
}

/*THIS IS A FUNCTION THAT BREAKS UP A PROPER GRAMMAR INTO A CNF GRAMMAR*/
```

```
void toCNF(void)
{
	int i,count=1;
	vector<rule> remGram,remString;
	for(i=0;i<(int)grammar.size();i++)
	{
		if((int)grammar[i].rhs.length()==1)
		{
			CNFrule newCNFrule;
			newCNFrule.LHS=grammar[i].lhs;
			newCNFrule.RHS[0]=grammar[i].rhs;
			newCNFrule.RHS[1]="";
			CNFgrammar.push_back(newCNFrule);
		}
		else
		{
			int numNonters=0,len;
			len=lengthOf(grammar[i].rhs,&numNonters);
			if(len==2)
			{
				if(numNonters==2)
				{
					CNFrule newCNFrule;
					newCNFrule.LHS=grammar[i].lhs;
					int j=0,k=0;

	newCNFrule.RHS[0]=getNextNonter(grammar[i].rhs,&j,&k);

	newCNFrule.RHS[1]=getNextNonter(grammar[i].rhs,&j,&k);
					CNFgrammar.push_back(newCNFrule);
				}
				else
				{
					if(numNonters==1)
					{
						CNFrule newCNFrule;
						newCNFrule.LHS=grammar[i].lhs;
						int j=0,k=0;
						string ter,nonter,newNT;
						nonter=getNextNonter(grammar[i].rhs,&j,&k);
						if(k==0)
						{
							newCNFrule.RHS[0]=nonter;

	ter=grammar[i].rhs.substr((int)nonter.length(),1);

	newNT=makeNewNonter(ter,remGram,&count);
							newCNFrule.RHS[1]=newNT;
							CNFgrammar.push_back(newCNFrule);

						}
						else
						{
							newCNFrule.RHS[1]=nonter;
							ter=grammar[i].rhs.substr(0,1);

	newNT=makeNewNonter(ter,remGram,&count);
							newCNFrule.RHS[0]=newNT;
							CNFgrammar.push_back(newCNFrule);
						}
						newCNFrule.LHS=newNT;
						newCNFrule.RHS[0]=ter;
						newCNFrule.RHS[1]="";
						if(!cnfRuleAlready(newCNFrule))
							CNFgrammar.push_back(newCNFrule);
						rule nr;
						nr.lhs=newNT;
						nr.rhs=ter;
						if(!rulePresent(nr,remGram))
							remGram.push_back(nr);
					}
```

```
                                        else
                                        {
                                                CNFrule newCNFrule;
                                                string newNT1,newNT2,ter1,ter2;
                                                newCNFrule.LHS=grammar[i].lhs;
                                                ter1=grammar[i].rhs.substr(0,1);
                                                ter2=grammar[i].rhs.substr(1,1);
                                                newNT1=makeNewNonter(ter1,remGram,&count);
                                                rule nr;
                                                nr.lhs=newNT1;
                                                nr.rhs=ter1;
                                                if(!rulePresent(nr,remGram))
                                                        remGram.push_back(nr);
                                                newNT2=makeNewNonter(ter2,remGram,&count);
                                                newCNFrule.RHS[0]=newNT1;
                                                newCNFrule.RHS[1]=newNT2;
                                                CNFgrammar.push_back(newCNFrule);
                                                newCNFrule.LHS=newNT1;
                                                newCNFrule.RHS[0]=ter1;
                                                newCNFrule.RHS[1]="";
                                                if(!cnfRuleAlready(newCNFrule))
                                                        CNFgrammar.push_back(newCNFrule);
                                                newCNFrule.LHS=newNT2;
                                                newCNFrule.RHS[0]=ter2;
                                                newCNFrule.RHS[1]="";
                                                if(!cnfRuleAlready(newCNFrule))
                                                        CNFgrammar.push_back(newCNFrule);
                                                nr.lhs=newNT2;
                                                nr.rhs=ter2;
                                                if(!rulePresent(nr,remGram))
                                                        remGram.push_back(nr);
                                        }
                                }
                        }
                        else
                        {
                                rule curRule;
                                curRule=grammar[i];
                                int dummy=0;
                                while(lengthOf(curRule.rhs,&dummy)!=2)
                                {
                                        string rep="";
                                        if(presRemStr(curRule.rhs,remString,&rep))
                                                curRule.rhs=rep;
                                        else
                                        {
                                                rule newRule,dumRule;
                                                dumRule.lhs=curRule.rhs;
                                                newRule.lhs=curRule.lhs;
                                                string nonter,symbol,newNT;
                                                int j=0,k=0;
                                                newNT=makeNewNonter("dummy",remGram,&count);
                                                curRule.lhs=newNT;
                                                nonter=getNextNonter(curRule.rhs,&j,&k);
                                                if((k==0)&&(nonter!=""))
                                                {
                                                        newRule.rhs=nonter;
                                                        newRule.rhs+=newNT;

        curRule.rhs=curRule.rhs.substr((int)nonter.length(),(int)curRule.rhs.length()-
(int)nonter.length());
                                                }
                                                else
                                                {
                                                        newRule.rhs=curRule.rhs.substr(0,1);
                                                        newRule.rhs+=newNT;

        curRule.rhs=curRule.rhs.substr(1,(int)curRule.rhs.length()-1);
                                                }
                                                dumRule.rhs=newRule.rhs;
                                                remString.push_back(dumRule);
```

```
                                        grammar.push_back(newRule);
                                }
                                grammar.push_back(curRule);
                        }
                }
        }
}

/*FUNCTION TO CREATE NEW NONTERMINALS DURING THE CONVERSION TO CNF*/
string makeNewNonter(string str,vector<rule> vec,int *cnt)
{
        int indx;
        for(indx=0;indx<(int)vec.size();indx++)
        {
                if(str==vec[indx].rhs)
                        return vec[indx].lhs;
        }
        char temp[10];
        sprintf(temp,"Z%d",(*cnt));
        (*cnt)++;
        return temp;
}

/*THIS IS A FUNCTION TO CHECK IF A CNF RULE HAS ALREADY BEEN ADDED*/
int cnfRuleAlready(CNFrule cr)
{
        int indx;
        for(indx=0;indx<(int)CNFgrammar.size();indx++)
        {

        if((cr.LHS==CNFgrammar[indx].LHS)&&(cr.RHS[0]==CNFgrammar[indx].RHS[0])&&(cr.RHS[1
]==CNFgrammar[indx].RHS[1]))
                        return 1;
        }
        return 0;
}

/*THIS FUNCTION IS USED FOR APPLYING THE RULES TO GENERATE SENTENTIAL FORMS*/
int presRemStr(string st,vector<rule> v,string *ret)
{
        int indx;
        for(indx=0;indx<(int)v.size();indx++)
        {
                if(st==v[indx].lhs)
                {
                        (*ret)=v[indx].rhs;
                        return 1;
                }
        }
        return 0;
}

/*THIS IS A FUNCTION THAT GENERATES STRINGS TO CHECK FOR AMBIGUITY*/
void genStrings(vector<strhis> *vec,strhis x,int *y)
{
        int prods,ps=0,p=0,amb;
        string newString,NT;
        NT=getNextNonter(x.str,&p,&ps);
        for(prods=0;prods<(int)properGrammar.size();prods++)
        {
                if((properGrammar[prods].lhs==NT)&&(NT!=""))
                {

        newString=x.str.substr(0,ps)+properGrammar[prods].rhs+x.str.substr(p,(int)x.str.le
ngth()-ps-(int)NT.length());
                        int dumy;
                        if(lengthOf(newString,&dumy)<=(*y))
                        {
                                strhis tem;
                                tem.str=newString;
```

```
                                tem.history=x.history+x.str+"==>";
                                amb=1;
                        if(!already((*vec),tem,&amb))
                                vec->push_back(tem);
                                if(amb>1)
                                {
                                        if((ambiguityFound==0)&&(isTerStr(newString)))
                                        {
                                ambiguityFound=1;
                                                amStr=newString;
                                                clock_t time2=clock();
                                                out<<"\t"<<"AMBIGUOUS SENTENTIAL FORM
"<<newString<<" FOUND AT LENGTH: "<<(int)newString.length()<<endl;
                                                out<<"\t"<<"TIME TAKEN TO DETECT AMBIGUITY:
"<<(time2-time1)/(double)CLOCKS_PER_SEC<<" SECONDS"<<endl;
                                                graph<<(time2-
time1)/(double)CLOCKS_PER_SEC<<"\t";
                                        }
                                        if((ambiguityFound==1)&&(newString==amStr)&&(amb>1))
                                                DOA++;
                                }
                        }
                }
        }
}


/*THIS FUNCTION CHECKS IF A STRING WITH THE SAME DERIVATION SEQUENCE HAS ALREADY BEEN
CHECKED FOR AMBIGUITY*/
int already(vector<strhis> st,strhis s,int *am)
{
        int x,retval=0;
        for(x=0;x<(int)st.size();x++)
        {
                if(s.str==st[x].str)
                {
                        if(s.history==st[x].history)
                 retval=1;
                        else
                                (*am)++;
                }
        }
        return retval;
}

/*A FUNCTION TO RETRIEVE THE NEXT NONTERMINAL FROM A PRODUCTION BODY*/
string getNextNonter(string s,int *z,int *y)
{
        while(((*z)<(int)s.length())&&((int)s[(*z)]<65)||((int)s[(*z)]>90))
                (*z)++;
        if((*z)==(int)s.length())
                return "";
        else
        {
                (*y)=(*z);
                (*z)++;
                while(((*z)<(int)s.length())&&((int)s[(*z)]<=57)&&((int)s[(*z)]>=48))
                        (*z)++;
                return(s.substr((*y),(*z)-(*y)));
        }
}

/*A SIMPLE FUNCTION TO CALCULATE 2^n*/
int twoPower(int n)
{
        int indx,val=1;
        for(indx=1;indx<=n;indx++)
                val*=2;
        return val;
}
```

```
/*A FUNCTION TO FIND THE NUMBER OF TERMINALS AND NONTERMINALS IN A SENTENTIAL FORM*/
int lengthOf(string str,int *nN)
{
        int len=0,indx;
        for(indx=0;indx<(int)str.length();indx++)
        {
                if((str[indx]<48)||(str[indx]>57))
                        len++;
                if((str[indx]>64)&&(str[indx]<91))
                        (*nN)++;
        }
        return len;
}

/*TO CHECK IF A STRING IS A SENTENCE OR A SENTENTIAL FORM*/
int isTerStr(string str)
{
        int x;
        for(x=0;x<(int)str.length();x++)
        {

        if((((int)str[x]<91)&&((int)str[x]>64))||(((int)str[x]<58)&&((int)str[x]>47)))
                        return 0;
        }
        return 1;
}
```

APPENDIX D

INPUT SUITE OF GRAMMARS

This appendix contains the test suite of 50 grammars that were used as input to the programs presented in AppendixC. Of the 50 grammars, 28 are ambiguous and the rest are unambiguous.

```
GRAMMAR#1 AMBIGUOUS
S->AB
S->a
A->SB
A->b
B->BA
B->a
**END
GRAMMAR#2 CLASSIC AMBIGUOUS
E->E+E
E->E*E
E->(E)
E->a
**END
GRAMMAR#3
E->E*T
E->T
T->T*F
T->F
F->(E)
F->a
**END
GRAMMAR#4 INHERENTLY AMBIGUOUS
S->AB
S->C
A->aAb
A->ab
B->cBd
B->cd
C->aCd
C->aDd
D->bDc
D->bc
**END
GRAMMAR#5 DEGENERATE
S->AB
S->CD
S->EF
A->a
B->b
C->a
D->b
E->a
F->b
**END
```

```
GRAMMAR#6 DANGLING ELSE
S->ictS
S->ictSeS
S->a
**END
GRAMMAR#7 UNAMBIGUOUS IF-ELSE
S->M
S->U
M->ictMeM
M->a
U->ictS
U->ictMeU
**END
GRAMMAR#8 UNAMBIGUOUS
S->aSa
S->bSb
S->c
**END
GRAMMAR#9 UNAMBIGUOUS
S->AB
S->ASB
A->a
B->b
**END
GRAMMAR#10 AMBIGUOUS
S->AB
S->CA
A->a
B->BC
B->AB
C->aB
C->b
B->b
**END
GRAMMAR#11 AMBIGUOUS
S->AB
S->BC
A->BAD
A->a
B->CC
B->bD
C->AB
C->c
D->d
**END
GRAMMAR#12 AMBIGUOUS
S->bA
S->aB
A->bAA
A->aS
A->a
B->bBB
B->b
B->SB
**END
GRAMMAR#13 UNAMBIGUOUS
S->AA
S->a
A->SS
A->b
**END
GRAMMAR#14 AMBIGUOUS
S->AB
S->BC
A->BA
A->a
B->CC
B->b
C->AB
C->a
**END
```

```
GRAMMAR#15 UNAMBIGUOUS
S->ABCD
A->CS
B->bD
D->SB
A->a
B->b
C->c
D->d
**END
GRAMMAR#16 UNAMBIGUOUS
S->N1V1
S->N2V2
N1->DN3S1
N2->DN4
V1->V3N2
V2->V4S1
V2->stink
S1->CS
D->the
N3->fact
N4->cats
N4->dogs
C->that
V4->think
V3->amazes
V3->bothers
**END
GRAMMAR#17 SAME AS #16 BUT WITH SINGLE LETTER TERMINALS
S->N1V1
S->N2V2
N1->DN3S1
N2->DN4
V1->V3N2
V2->V4S1
V2->s
S1->CS
D->t
N3->f
N4->c
N4->d
C->h
V4->i
V3->a
V3->b
**END
GRAMMAR#18 AMBIGUOUS
S->PQ
P->ROT
P->a
R->MP
O->a
O->ab
T->b
T->bb
M->a
Q->CeD
C->a
C->ab
D->d
D->ed
**END
GRAMMAR#19 UNAMBIGUOUS
S->ABD
S->ABC
A->AE
A->a
B->SE
B->b
D->d
C->c
```

```
E->dc
**END
GRAMMAR#20 UNAMBIGUOUS
S->BC
S->AB
A->CF
C->c
F->ged
B->ab
B->aC
C->c
**END
GRAMMAR#21 AMBIGUOUS
S->U
S->V
U->TaU
U->TaT
V->TbV
V->TbT
T->aTbT
T->bTaT
T->
**END
GRAMMAR#22 AMBIGUOUS
S->AA
A->AAA
A->bA
A->Ab
A->a
**END
GRAMMAR#23 UNAMBIGUOUS
S->ACA
A->aAa
A->B
A->C
B->bB
B->b
C->cC
C->c
**END
GRAMMAR#24 UNAMBIGUOUS
T->ABC
T->ABD
A->AD
A->AC
A->a
B->AB
B->b
C->c
D->d
**END
GRAMMAR#25 AMBIGUOUS
A->AS
S->AB
S->BB
S->b
A->bA
A->AS
A->a
**END
GRAMMAR#26 UNAMBIGUOUS
S->AB
B->bb
B->bB
A->a
A->aAb
**END
GRAMMAR#27 AMBIGUOUS
A->a
A->B
A->CA
```

```
B->bD
B->b
D->d
D->dD
D->ad
C->bc
C->c
C->CC
**END
GRAMMAR#28 UNAMBIGUOUS
Z->aXY
Z->bXZ
Z->z
X->aY
X->az
X->y
Y->y
**END
GRAMMAR#29 AMBIGUOUS
S->NVNDJ
N->a
N->h
N->PJN
N->PN
V->f
V->e
P->f
P->p
D->r
D->v
J->b
J->g
J->PJ
**END
GRAMMAR#30 AMBIGUOUS
S->AB
S->CA
A->a
B->BC
B->AB
C->aB
C->b
B->b
**END
GRAMMAR#31 AMBIGUOUS
S->AB
S->BC
A->BAD
A->a
B->CC
B->bD
C->AB
C->c
D->d
**END
GRAMMAR#32 AMBIGUOUS
S->bA
S->aB
A->bAA
A->aS
A->a
A->bBB
B->b
B->SB
**END
GRAMMAR#33 UNAMBIGUOUS
S->T
S->a
T->A
T->b
T->T
```

```
A->ab
A->aab
**END
GRAMMAR#34 AMBIGUOUS
S->t
S->e
S->he
S->S
S->eH
S->eHS
S->H
H->hH
H->h
H->ht
**END
GRAMMAR#35 UNAMBIGUOUS
S->WCT
W->while
C->bconditionb
T->bRb
R->statement
R->statementR
**END
GRAMMAR#36 UNAMBIGUOUS
S->WCT
W->w
C->bcb
T->bRb
R->s
R->sR
**END
GRAMMAR#37 UNAMBIGUOUS
S->wbcbbRb
R->s
R->sR
**END
GRAMMAR#38 AMBIGUOUS
S->SS
S->AS
S->a
S->b
A->AA
A->AS
A->a
**END
GRAMMAR#39 AMBIGUOUS
S->b
S->Tb
S->TQ
T->baT
T->caT
T->aT
T->ba
T->ca
T->a
Q->bc
Q->bcQ
Q->caQ
Q->ca
Q->a
Q->aQ
**END
GRAMMAR#40 UNAMBIGUOUS
T->rXr
T->rXrT
X->text
X->C
C->dtextd
C->dtextdC
**END
GRAMMAR#41 UNAMBIGUOUS
```

```
S->UVPO
S->UVCPO
U->house
U->houseflies
V->flies
V->like
C->like
P->a
P->the
O->banana
**END
GRAMMAR#42 AMBIGUOUS
S->MN
S->PQ
M->aM
M->Mc
M->b
N->Nc
N->bN
N->c
P->Pd
P->cP
P->d
Q->ad
**END
GRAMMAR#43 UNAMBIGUOUS
A->BDE
B->cA
B->c
B->a
D->cD
D->d
D->aB
E->e
E->de
E->ce
**END
GRAMMAR#44 AMBIGUOUS
S->SAB
S->ASB
S->b
A->ab
A->Ba
B->b
B->bB
**END
GRAMMAR#45 UNAMBIGUOUS
L->AND
L->GA
A->a
A->aA
A->ab
N->ab
D->ba
D->Da
G->bG
G->baG
G->ba
**END
GRAMMAR#46 AMBIGUOUS
S->NVN
S->NVNVingN
N->dogs
N->NVingN
V->eat
**END
GRAMMAR#47 UNAMBIGUOUS
A->SB
A->AS
S->ab
B->b
```

```
B->bB
B->SB
**END
GRAMMAR#48 AMBIGUOUS
P->PRQ
P->p
R->pr
R->r
R->rR
Q->q
Q->qQ
Q->rq
**END
GRAMMAR#49 AMBIGUOUS
S->ABSB
S->ASB
S->a
A->aA
A->a
B->ab
B->AB
B->b
B->bB
**END
GRAMMAR#50 AMBIGUOUS
S->AC
S->BA
A->Ba
A->aB
A->a
C->CB
C->c
B->Bc
B->b
B->bc
**END
```

VITA

Saichaitanya Jampana

Candidate for the Degree of

Master of Science

Thesis: EXPLORING THE PROBLEM OF AMBIGUITY IN CONTEXT-FREE GRAMMARS

Major Field: Computer Science

Biographical:

Personal Data: Born in Kakinada, Andhra Pradesh, India, October 17, 1979, son of Mr. Suryanarayana Raju and Mrs. Annapurna Devi.

Education: Received the degree of Bachelor of Technology in Computer Science from Jawaharlal Nehru Technological University, Andhra Pradesh, India, in May 2001; completed the requirements for Master of Science degree in Computer Science at the Computer Science Department of Oklahoma State University, Stillwater, Oklahoma, in July 2005.

Experience: Web Programmer for Star Schools Project under the Earth Science and Microbiology teams at Oklahoma State University, Stillwater, from October 2001 to June 2002; Graduate Assistant and Programmer for Accounting Department of Oklahoma State University, Stillwater, from January 2003 to May 2003.

Name: Saichaitanya Jampana Date of Degree: July 2005

Insitution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: EXPLORING THE PROBLEM OF AMBIGUITY IN CONTEXT-FREE GRAMMARS

Pages in Study: 84 Candidate for the Degree of Master of Science

Major Field: Computer Science

Scope and Method of Study: The general problem of ambiguity detection is unsolvable. Some context-free languages are inherently ambiguous. There is no algorithm that can detect the ambiguity of context-free grammars (CFG) in general since it is a provably unsolvable problem. The objective of this thesis is to study the notion of ambiguity in context-free grammars and grammars in general. The areas of this study include ambiguity in CFGs, ambiguity in other classes of grammars, and a number of algorithms for finding ambiguous strings and grammars. Also, an algorithm that finds ambiguous strings in a subset of context-free grammars is presented. This algorithm is based on the observation that ambiguity in strings cannot be induced by mere repetitions of applications of productions unless ambiguous strings can be generated without exhibiting the self-embedding property.

Findings and Conclusions: Ambiguity in different classes of formal languages and in some programming languages was studied. The problem of ambiguity detection in context-free grammars was studied in depth. An algorithm was designed and implemented to detect ambiguous strings generated by a subset of context-free grammars. The algorithm works successfully for all grammars in the test suite of grammars which are known to be either ambiguous or unambiguous. It was observed that checking the sentential forms for ambiguity instead of checking only the sentences, and applying the productions of the proper grammar instead of the CNF grammar to derive the sentential forms are promising techniques to keep the execution time low. It was also observed that the time in which ambiguity was detected in the grammars in the test suite was largely independent of factors such as the productivity of a grammar (the number of strings that can be generated without exhibiting the self-embedding property), the degree of ambiguity of a string, and the lengths of strings generated.

ADVISER'S APPROVAL: M. H. Samadzadeh