Ambiguity Detection Using Approximation Techniques

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Software Engineering

Submitted By
Saurabh Kumar Jain
(801131023)

Under the supervision of:
Mr. Ajay Kumar
(Assistant Professor)

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

July 2013

Certificate

I hereby certify that the work which is being presented in the thesis entitled, "*Ambiguity Detection Using Approximation Techniques*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Software Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Mr. Ajay Kumar* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

(Saurabh Kumar Jain)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Mr. Ajay Kumar)
Assistant Professor
Computer Science and
Engineering Department
Thapar University
Patiala

Countersigned by:

(Dr. Maninder Singh)
Associate Professor and Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

// Acknowledgement

I would like to express my deep sense of gratitude to my supervisor, **Mr. Ajay Kumar**, Assistant Professor, Computer Science and Engineering Department, Thapar University, Patiala, for his invaluable help and guidance during the course of thesis. I am highly indebt to him for constantly encouraging me by giving his critics on my work. I am grateful to him for giving me the support and confidence that helped me a lot in carrying out the research work in the present form. And for me, it's an honor to work under him.

I also take the opportunity to thank **Dr. Maninder Singh**, Associate Professor and Head, Computer Science and Engineering Department, Thapar University, Patiala, for providing us with the adequate infrastructure in carrying out the research work.

I would also like to thank my parents and friends for their inspiration and ever encouraging moral support, which went a long way in successful completion of my thesis.

Above all, I would like to thank the almighty God for His blessings and for driving me with faith, hope and courage in the thinnest of the times.

Saurabh Kumar Jain

Abstract

One way to verifying a grammar is the detection of ambiguity. Unfortunately, ambiguity problem for context-free grammars is un decidable. Ambiguity in context-free grammars is a recurring problem in language design and parser generation, as well as in applications where grammars are used as models of real-world physical structures. Context-free grammars are widely used but still hindered by ambiguity. It was observe that there is simple linguistic characterization of the grammar ambiguity problem .This problem divided into form of horizontal and vertical ambiguity. We show the conservative approximation for ambiguity problem. 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.

In this thesis the available techniques have been compared. A new approach has been proposed. This approach work on Chomsky Normal form(CNF) of the Grammar because Chomsky Normal Form of the Context free Grammar construct a polynomial-time algorithm to decide whether or not a given string is in the language generated by that grammar. A Grammar is a simple structure, and that makes it easy to parse. In an arbitrary CFG, there is no a priori bound on the length of a derivation of an input word. The experimental results demonstrate that proposed approach can effectively detect the ambiguity in Context free Grammar.

Table of Contents

Certificate i

Acknowledgement ii

Abstract iii

Table of Contents 3

List of Figures and Table...vi

Chapter 1 Introduction 1

1.1 Language 1

1.2 Formal Grammar 1

1.3 Chomsky Hierarchy 1

1.4 Context free Grammar 2

1.5 Normal Form of Context free Grammar 4

1.6 Advantages of Context free Grammar 5

1.7 Ambiguity in Context free Grammar 5

1.8 Inherent Ambiguity..6

1.9 Ambiguity in different class of Grammar 6

1.10 Organization of Thesis 9

Chapter 2 Ambiguity detection methods for Context free Grammar..........................10

2.1 Parsing Techniques 10

2.2 Ambiguity Approximation 11

2.3 Comparison Parameters 12

2.4 Comparison Framework for Existing Methods 14

Chapter 3 Ambiguity detection with Language Approximation 25

3.1 Horizontal and Vertical ambiguity 25

Chapter 4 Problem Analysis 30
4.1 Problems in ambiguity detection 30
4.2 objective 31
Chapter 5 Proposed Techniques for ambiguity detection 32
5.1 Conversion of Grammar into Chomsky Normal Form 32
5.2 Computation of First and Last Function 36
5.3 Detection of Vertical ambiguity Production 38
5.4 Detection of Horizontal ambiguity Production 38
Chapter 6 Experimental Results 41
6.1 Structure of Input Grammar 41
6.2 Implementation of Proposed Technique 42
Chapter 7 Conclusion and Future Scope 45
7.1 Conclusion...45
7.2 Future Scope..45
References 46
List of Publications 49

List of Figures and Table

Figure 1.1: Chomsky Hierarchy..1

Figure 2.1: Representation of parse tree for an English language 3

Figure 2.1: Example to clarify ADM 18

Figure 2.2: Position Graph Example 22

Figure 3.1: Two parse tree for horizontal Ambiguity 26

Figure 3.2: Two parse tree for vertical Ambiguity 26

Figure 3.3: Induction basis 27

Figure 3.4: Inductive step of theorem 3.3 27

Figure 3.5: Case of Vertical ambiguity 28

Figure 3.6: Case of Horizontal ambiguity 29

Figure 3.7: Case of Vertical or Horizontal ambiguity..29

Figure 6.1: Demonstration of input grammar into CNF...42

Figure 6.2: First function of the Input grammar 42

Figure 6.3: Vertical or Horizontal Ambiguity type detection 43

Figure 6.4: Another Scenario the Ambiguity test (Vertical or Horizontal).................43

Table

Table 2.1: Summary of existing methods of ambiguity Detection 24

Chapter 1

Introduction

Formal grammars provide a syntactic generative way of defining languages. In this chapter the basic introduction about the grammar and their ambiguity problems are discussed.

1.1 Language

A language consists of three different entities letters, words, and sentences. There is a certain relation between the fact that groups of letters make up words and the fact that groups of words make up sentences. Not all collections of letters form a valid word, and not all collections of words form a valid sentence [1].

1.2 Formal Grammar

It defines the syntax of languages using simple and precise mathematical structure. When characterization of the structure of natural languages is needed, formal grammar is used [2].

1.3 Chomsky Hierarchy

Chomsky hierarchy [2] is a containment hierarchy of classes of formal grammars which is shown from the diagram below Figure 1.1 According to Chomsky hierarchy formal languages can be classified as regular languages, context-free, the context-sensitive and the recursively enumerable languages.

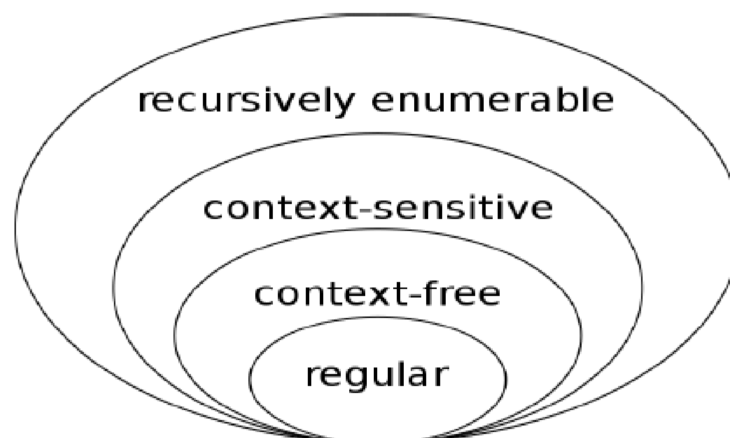


Figure: 1.1 Chomsky Hierarchy [2]

- Type-0 grammar (unrestricted grammars): They generate all languages that can be recognized by a Turing machine [2] .These languages are also known as the recursively enumerable languages.
- Type-1 grammar: They generate the context-sensitive languages. The languages described by these grammars are exactly all languages that can be recognized by a linear bounded automation [2].
- Type-2 grammar: They generate context-free languages. These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton. Context free languages are the basis for the syntax of most programming languages [2].
- Type-3 grammar: They generate regular languages. Such a grammar restricts its rules to a single non-terminal on the left-hand side of the terminal or a non terminal on the right-hand side consisting of a number of terminals. Regular languages are commonly used to define search patterns and the lexical structure of programming languages [2].

1.4 Context-Free Grammar

A grammar for a language is a set of rules used for the generation of sentences in that language. Every language has a grammar, whether it is a natural language (English, Spanish etc) or a programming language (C++, Java, etc.) [1].

Context-free grammars, one of the four classes of grammars as defined by Noam Chomsky [1], 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. Applications that use context-free grammars typically require a unique structure for each sentence the grammars generate. But 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.

A grammar G has a 4-tuple, (V, T, P, S), where V is the set of variables, also called non terminals, 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 [1]. 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 non terminal 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 non terminals [1]. 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.

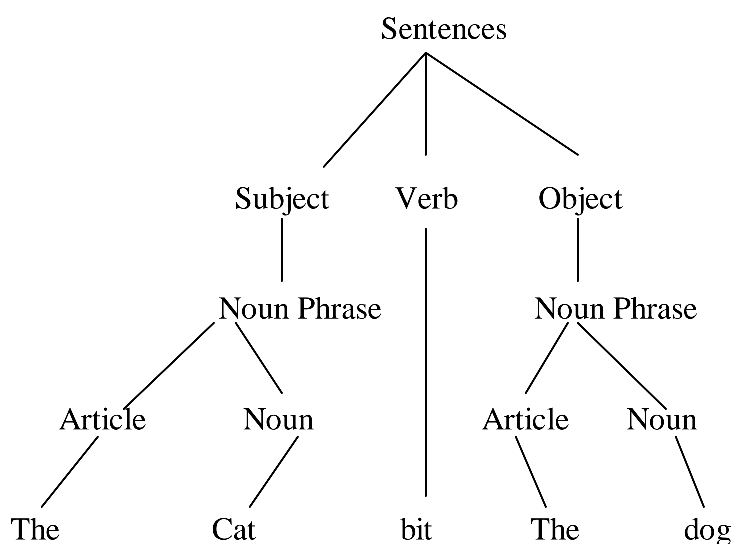


Figure: 1.2 Representation of parse tree for an English language

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 [3]. 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 reduce-reduce conflict. The following example illustrates these conflicts.

Example: 1.1 Given the following grammar and the string w = expr – expr,

expr → expr – expr | expr + expr

diff → expr – 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.

1.5 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 [1]. 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.

1.5.1 Chomsky Normal Form

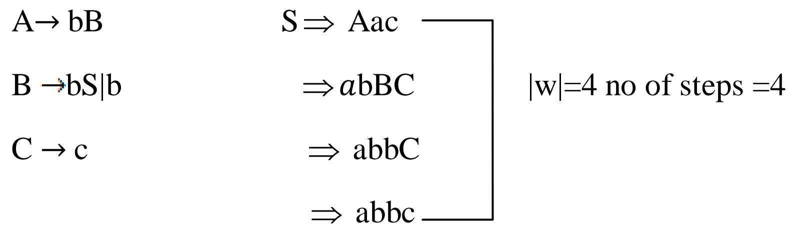
A grammar G = (V, T, P, S) is in Chomsky normal form if its productions are restricted to the forms:

- A → BC where A, B and C are variables in V and exactly two variables are on the right
- A → a where *A* is a variable in V and a is exactly one terminal symbol in T.

1.5.2 Greibach Normal Form

Every CFL grammar that does not produce the empty string can be converted into a grammar in Greibach Normal Form (GNF) [1]. A grammar is in Greibach Normal Form if all of its productions are of the form: A → aα, where A is a non terminal, a is a terminal, and α is a string of zero or more non terminals. 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.

Example: 1.2 Grammar S→ aAC Derivation sequence for w=abbc



1.6 Advantages of Chomsky Normal Form

For 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. For a grammar in Chomsky Normal Form, the parse tree for a given string is always a binary tree.

A grammar in CNF has a simple structure, and that makes it easy to parse. In an arbitrary CFG, there is no a priori bound on the length of a derivation of an input word (there could be many useless rules that later expand into ε), so it is not possible to test membership by generating all possible derivations. In contrast, in grammars in CNF, there is a bound on the length of any derivation of a word of length n, it's exactly 2^n–1. A grammar in CNF is also useful when it is interpreted as a parse tree. This is because the parse tree of a CNF grammar forms a binary tree [4].

The normal forms (Chomsky, Greibach etc.) were invented to solve the elementary problems involving CFL's, such as deciding membership and testing emptiness, more easily. The Chomsky normal form of a grammar yields efficient algorithms.

1.7 Ambiguity in Context Free Grammar

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[1]. 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 [5]. 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 [5].

Example: 1.3 The following ambiguous grammar

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow a$

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

$E \Rightarrow E+E \Rightarrow E*E+E \Rightarrow a*E+E \Rightarrow a*a+E \Rightarrow a*a+a$

$E \Rightarrow E*E \Rightarrow a*E \Rightarrow a*E+E \Rightarrow a*a+E \Rightarrow a*a+a$

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

1.8 Inherent Ambiguity

If all grammars that generate a language are ambiguous, that language is said to be inherently ambiguous [1]. 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. In a language L that can be defined as the union of two other languages L1 and L2, all sentences in the intersection of the sets L1 and L2 have two different interpretations because they belong to both L1 and L2. 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 $L1 \cap L2$, is inherently ambiguous [6].

1.9 Ambiguity in Different Classes of Grammar

Ambiguity in different classes of formal grammars and ambiguity in programming languages are discussed.

1.9.1 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 [7].

1.9.2 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 [8]. 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 [8]. The Pascal language has ambiguities in types, sets, and scope rules 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 [9]. 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 [10].

Example: 1.4 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.//
```

1.9.3 Ambiguity in Two-Level Grammars

Two-level grammars are a class of grammars that have their productions defined in two levels. W-grammars are those that have a context-free grammar as their first level grammar which defines the non terminals of the context-free grammar in their second level grammar [11].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 [11]. 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

1.9.4 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 non terminals, x has at least one non terminal 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 non terminals, and A is a non terminal [1]. 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 [5]. Because of the context-sensitiveness, productions of a CSG are generally stricter and also more powerful than those of a context-free grammar. [12].

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. It is un-decidable 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 grammar. 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 non terminals and x has at least one non terminal [1]. 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 [12].

1.10 Organization of Thesis

The rest of the thesis is organized as follows:

Chapter 2 - This chapter describes in detail the problem of ambiguity and Their existing techniques to solve the problem and comparison between these techniques based on different factors.

Chapter 3 - This chapter describe the approximation method for solving this problem, That gives the classification of ambiguity problem.

Chapter 4 - This chapter describes the problem Analysis of the thesis work. It gives the problem analysis, problem statement and aim of the thesis.

Chapter 5 - This chapter describes the proposed method for the solution of the ambiguity problem.

Chapter 6 - This chapter focus on the implementation details and experimental results description of proposed technique and snapshot of results.

Chapter 7 – This chapter describes the conclusion and future research work possible.

Chapter 2

Ambiguity Detection Methods for Context Free Grammar

The problem of deciding whether a given (context-free) grammar for a language is ambiguous is unsolvable. 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 un-decidable, is reducible to the problem of detecting ambiguity in a context-free grammar. Hence, the problem of context-free grammar ambiguity detection is also un-decidable [2].

There is no algorithm which takes ambiguous CFG as input, can always produce an unambiguous context-free grammar as output that generates the same language [6].
These are following techniques to analyze this problem-

- Parsing Techniques
- Ambiguity Approximation

2.1 Parsing Techniques

Parsing is the retrieval of the tree structure of a one-dimensional sequence of symbols. The tree structure is described by a context-free grammar. The recovered structure is called a parse tree.

2.1.1 Eickel and Paul's Algorithm

The algorithm proposed by Eickel et al. [27] 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, if identical sentential forms are encountered in derivation process, the input string is consider 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.

2.1.2 Algorithm Used by YACC (Yet another Compiler Compiler)

YACC is an LALR (1) parser generator developed at the Bell laboratories. YACC detects ambiguity in LALR (1) grammars [3]. YACC indicates ambiguity when it encounters reduce-shift conflicts or reduce- reduce conflicts. 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 [4].

2.1.3 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 [13]. 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 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.

2.2 Ambiguity Approximation

There is no algorithm exists which solves the ambiguity problem for every context free grammar, the existing algorithms solve the problem for some grammar, We call such algorithms approximations [28]. An approximation algorithm can return up to three values for the ambiguity problem: unambiguous, ambiguous and n/a, if it cannot make a decision. Here, we only consider algorithms that finish in finite time and never return a wrong result (i.e. it does never return ambiguous when the grammar is unambiguous nor unambiguous for an ambiguous grammar). Such an algorithm is called a safe approximation. There are following two types of methods have been discussed.

2.2.1 Position Automation Method

A method to determine whether a grammar is unambiguous is to generate an LR(k) parser. If there are no conflicts, then the grammar is unambiguous. This does not necessarily mean that if conflicts were found that the grammar is ambiguous, thus this is a conservative approximation. An extension has been presented by Sylvain Schmitz

in [20]. It has some similarities to the GLR extension for LR parsers: The processing is not aborted when a conflict is encountered, but it continued to parse. Within a GLR parser, this means to run multiple parser instances in parallel. For an ambiguity detector it means to find out whether it is possible for more than one instance to reach an accept action. The approach is not specific to LR(k) parsers. Instead, it works on grammar positions. A grammar position is the abstraction of the configuration of a parser and denotes its current position within a parse tree.

Noncanonical Unambiguity (NU) test [12] is a conservative ambiguity detection method that uses approximation to limit its search space. It constructs a nondeterministic finite automaton (NFA) that describes an approximation of the language of the tested grammar. The approximated language is then checked for ambiguity by searching the automaton for different paths that describe the same string. This can be done in finite time, but at the expense of incorrect results. The test is conservative however, allowing only false positives. If a grammar is Noncanonical Unambiguous then it is not ambiguous.

2.2.2 Ambiguity Checking with Language Approximations

The ambiguity detection algorithm *Ambiguity Checking with Language Approximations* (ACLA) has been proposed by Claus Brabrand et al. [18]. Instead of dealing with the ambiguity problem as a whole, it is separated into two similar problems, namely vertical and horizontal ambiguity. Both problems are un-decidable again.

In this ambiguity problem divided into horizontal and vertical ambiguity, and superset of the original grammar allows an approximation of the ambiguity problem.

2.3 Comparison Parameters

The currently existing *Ambiguity Detection Methods* (ADM) all have different characteristics, because they approach the ambiguity problem from different angles. These characteristics influence the practical usability of an ADM. They will be the criteria we will compare the investigated methods. In order to measure and compare the usability of the investigated methods, we will first need to define it. This section

describes the criteria for practical usability that we distinguish, how we measured them and how we analyzed the results [29].

2.3.1 Termination

The search space of some methods can be infinite, because the ambiguity detection problem is un-decidable. On certain grammars they might run forever, but could also take a very long time to terminate [29]. If the ambiguity of a grammar is not known, it is not possible to predict if some methods will terminate on it. Practically it is not possible to run a method forever, so it will always have to be halted after some time. If at that point the method has not yet given an answer, the ambiguity of the tested grammar remains uncertain. To be practically usable, an *ADM* has to terminate in a reasonable amount of time. Methods that apply exhaustive searching can run into an infinite search space for certain grammars.

2.3.2 Accuracy

The accuracy of a method can be determined by the percentage of correct reports. A method that is guaranteed to correctly identify the ambiguity of an arbitrary grammar, is 100% accurate. Because the ambiguity problem is un-decidable, a method that always terminates and is 100% correct can never exist. There is a tradeoff between termination and accuracy. Some methods are able to correctly report ambiguity, but run forever if a grammar is unambiguous. Other methods are guaranteed to terminate in finite time, but they report potential false positives or false negatives. After a method has terminated on a grammar it needs to correctly identify the ambiguity of the tested grammar. Not all methods are always able to produce the right answer, for instance those that use approximation techniques. Reports of such methods need to be verified by the user, which influences the usability. We define the accuracy of an *ADM* on a set of grammars as its percentage of correct reports. This implies that a method first has to terminate before its report can be tested. Executions that do not terminate within a set time limit are not used in our accuracy calculations [29].

2.3.3 Performance

The worst case complexity of an algorithm tells something about the relation between its input and its number of steps, but this does not tell much about its runtime behavior on a certain grammar [29]. A method of exponential complexity might only

take a few minutes on a certain input, but it might also take days or weeks. How well a method performs on an average desktop PC also influences its usability. This criterion is closely related to termination, because it affects the amount of time one should wait before halting a method.

2.3.4 Usefulness of reports

The goal of an ambiguity detection method is to report ambiguities in a grammar, so they can be resolved. This resolution is not always an easy task. First the cause of the ambiguity has to be understood. Sometimes it is only the absence of a priority declaration, but it can also be a complex combination of different production rules. To aid the grammar developer in understanding the ambiguity it reports should be as clear and concise as possible. After a method has successfully terminated and correctly identified the ambiguity of a grammar, it becomes even more useful if it indicates the sources of ambiguity in the grammar. That way they can be verified or resolved. An ambiguity report should be grammar oriented, localizing and succinct. A very useful one would be an ambiguous example string, preferably as short as possible, together with its multiple derivations [29].

2.3.5 Scalability

There is usually a trade-off between the performance and accuracy of a method. Accurately inspecting every single possible solution is more time consuming than a more superficial check. The scalability of a method is its possibility to be parametrized to run with higher accuracy in favor of execution time or vice versa. Some *ADM* can be executed with various levels of accuracy. There is usually a trade-off between their accuracy and performance [29]. Accurately inspecting every single possible solution is more time consuming than a more superficial check. The finer the scale with which the accuracy of an *ADM* can be exchanged for performance, the more usable the method becomes.

2.4 Comparison Framework for Existing Methods

There are several methods have been used for solving the ambiguity problem, here compare these methods on different parameters.

2.4.1 Gorn Method

In Gorn Method describes a Turing machine that generates all possible strings of a grammar, a so called derivation generator. Upon generation of a new string it searches if it has been generated before. If this is the case then the string has multiple derivations and it is ambiguous [12]. This algorithm is a simple breadth first search of all possible derivations of the grammar. Starting with the start symbol, it generates new sentential forms by expanding all non-terminals with their productions. Strings containing only terminals (the sentences) are compared to previously generated strings. Strings containing non-terminals are expanded to the next derivation level.

Termination For an unambiguous recursive grammar this method will never terminate. In this case the number of derivation levels is infinite. The method does terminate on non recursive grammars. These grammars are usually not very useful however, because their language is finite [29].

Correctness If the method finds an ambiguous string then this report is 100% correct. The method is thus capable of correctly identifying ambiguities in a grammar. However it is not always capable of reporting non-ambiguity, because of its possible non termination. Running the method forever is impossible so it will always have to be halted at some point [29]. At that moment there might still be an undiscovered ambiguity left in the grammar. If the method has not found another ambiguity yet, it will remain uncertain whether the grammar is ambiguous or not.

2.4.2 Cheung and Uzgalis Method

The method of Cheung and Uzgalis [13] is a breadth-first search with pruning of all possible derivations of a grammar. It could be seen as an optimization of Gorn's method. It also generates all possible sentential forms of a grammar and checks for duplicates. However it also terminates the expansion of sentential forms under certain conditions. This happens when a sentential form cannot result in an ambiguous string or when a similar form is also being searched. To detect this it compares the terminal pre and post fixes of the generated sentential forms. For instance, searching a

sentential form is terminated when all other forms have different terminal pre- and postfixes, and the ’middle’ part (what remains if the terminal pre- and postfixes are removed) has already been expanded on its own before. In this way certain repetitive expansion patterns are detected and are searched only once [13]. The following example is an unambiguous grammar with an infinite number of derivations, for which this method terminates and correctly reports non-ambiguity:

$$S \rightarrow A \mid B, \quad A \rightarrow aA \mid a, \mathrm{B} \rightarrow bB \mid b$$

The first level of expansion will result in strings ‘A’ and ‘B’. The second expansion generates strings ‘aA’, ‘a’, ‘bB’ and ‘b’. At this point the expansion of all derivations can be stopped, because they would only repeat themselves. The sentential forms (‘aA’ and ‘bB’) have different terminal prefixes, and their ’middle’ parts (‘A’ and ‘B’) have already been expanded during the second expansion. The *ADM* thus terminates and reports non-ambiguity.

Termination Because the method stops expanding certain derivations, it is possible that it terminates on recursive grammars. However, this does not always have to be the case.

Correctness Just like Gorn this method is able to correctly detect ambiguity. In addition it can also detect non-ambiguity for some grammars with an infinite language. If all search paths of the grammar are terminated before an ambiguous string is found, the grammar is not ambiguous. In [13] the author proves the pruning is always done in a safe way, so no ambiguities are overlooked.

2.4.3 AMBER

AMBER is an ambiguity detection tool developed by Schroer [14]. It uses an Earley parser [15] to generate all possible strings of a grammar and checks for duplicates. All possible paths through the parse automaton are systematically followed which results in the generation of all derivations of the grammar. Basically this is the same as Gorn’s method [12]. However, the tool can also be parametrized to use some variations in the search method. For instance there is the ellipsis option to compare all generated sentential forms to each other, instead of only the ones consisting solely of terminals. This way it might take less steps to find an ambiguous string. It is also

possible to bound the search space and make the searching stop at a certain point. One can specify the maximum length of the generated strings, or the maximum number of strings it should generate [15]. This last option is useful in combination with another option that specifies the percentage of possible expansions to apply each step. This will result in a random search which will reach strings of greater length earlier.

Termination AMBER will not terminate on recursive grammars unless a maximum string length is given.

Correctness With its default parameters this method can correctly identify ambiguities, but it cannot report non-ambiguity (just like Gorn). This also holds when it compares the incomplete sentential forms too. In the case of a bounded search or a random search, the method might report false negatives, because it overlooks certain derivations [29].

Scalability This method is scalable in two ways: 1) Sentential forms holding non terminals can also be compared and 2) the percentage of possible derivations to expand each level is adjustable [29]. The first option might result in ambiguities being found in fewer steps. The correctness of the reports remains the same. In the second case the longer strings are searched earlier, but at the price of potential false positives.

2.4.4 Jampana Method

Jampana's ambiguity detection method is based on the assumption that all ambiguities of a grammar in Chomsky Normal Form (CNF) will occur in derivations in which every live production is used at most once. (The live productions of a CNF grammar are those of the form $A \rightarrow BC$) His algorithm consists of searching those derivations for duplicate strings (like Gorn), after the input grammar is converted to CNF [16].

Termination This *ADM* is guaranteed to terminate on every grammar. The set of derivations of a CNF grammar with no duplicate live productions is always finite. Correctness Unfortunately Jampana's [16] assumption about the repetition of the application of productions is incorrect. If it was true then the number of strings that had to be checked for ambiguity is finite, and the CFG ambiguity problem would

become decidable [29]. Failing to search strings with repetitive derivations will leave ambiguities undetected, so this algorithm might report false negatives.

This can be shown with the following simple expression grammar:

$$E \rightarrow E + E \mid a$$

When converted into CNF

$$E \rightarrow EO \mid a\,,\ O \rightarrow PE\,,\ P \rightarrow +$$

The live productions here are E→EO and O →PE. The strings that do not use a live production more than once are: ‘a’ (Figure 2.1.a) and ‘a+a’ (Figure 2.1.b). These strings are not ambiguous, but the grammar is. The shortest ambiguous string for this grammar uses a live production twice ‘a+a+a’. Its different derivation trees are shown in figures 1.1.c and 1.1.d.

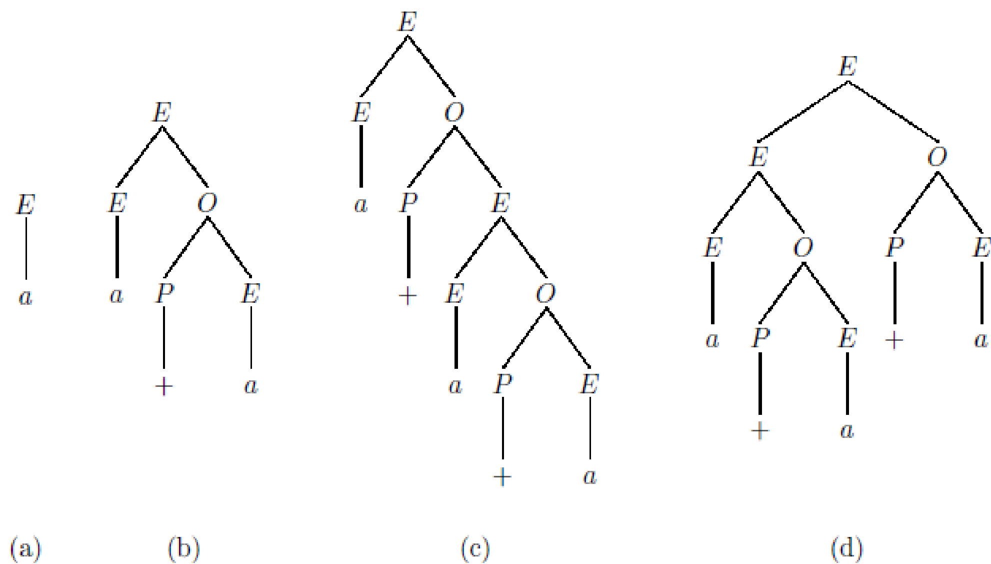


Figure: 2.1 Example to clarify ADM [29]

Scalability Like AMBER, this method can also be optimized by comparing the sentential forms that still contain non terminals.

2.4.5 LR(k) test

LR(k) parsing is a bottom-up parsing technique that makes decisions based on k input symbols of lookahead. A parse table is used to look up the action to perform for the current lookahead, or the next state to go to after the reduction of a nonterminal [17]. The possible actions are shifting an input symbol, reducing with a production rule, accepting the input string or reporting an error.

The class of grammars that can be deterministically parsed with this algorithm are called LR(k) grammars. This means there is a value of k for which a parse table can be constructed of the grammar, without conflicts. A conflict is an entry in the parse table where multiple actions are possible in a state for the same lookahead. These are either shift/reduce or reduce/reduce conflicts. A conflict means there is no deterministic choice to be made at a certain point during parsing [17].

A parse table without conflicts will always result in a single derivation for every string in the language of the grammar. So if a grammar is LR(k) then it is unambiguous. Unfortunately the class of LR(k) grammars is smaller than the class of unambiguous grammars [29]. If a grammar is not LR(k) this is no guarantee for ambiguity. Testing if a grammar is in the LR(k) class can be done by generating its parse table for a certain value of k. If the parse table contains no conflicts then the grammar is LR(k). This test can also be used as an ambiguity detection method. It can be used to look for a value of k for which a parse table can be generated without conflicts. It will have to be applied with increasing k. If a k is found for which the test passes then the grammar is known to be unambiguous.

Termination If the input grammar is LR(k) for some value of k, then the test will eventually pass. If the grammar is not LR(k) for any k, then the search will continue forever.

Correctness The LR(k) test can only report non-ambiguity, because in the case of an ambiguous grammar it does not terminate. It also does not terminate on unambiguous grammars that are not LR(k). If it does terminate then its report about the unambiguity of a grammar is 100% correct.

2.4.6 Brabrand, Giegerich and Møller Method

The method of Brabrand et al. [18] searches for the individual cases of horizontal and vertical ambiguities in a grammar. All individual productions are searched for horizontal ambiguities and all combinations of productions of the same non terminal are searched for vertical ambiguities. They describe horizontal and vertical ambiguity not in terms of a grammar but in terms of languages. Two productions form a vertical ambiguity when the intersection of their languages is non-empty. A production forms

a horizontal ambiguity when it can be split up in two parts whose languages overlap. So instead of the actual productions this method checks their languages. The languages of the productions are approximated to make the intersection and overlap problems decidable. The method is actually a framework in which various approximation techniques can be used and combined. Algorithm by Mohri et al. [19] extends the original language into a language that can be expressed with a regular grammar. This is called a conservative approximation, because all strings of the original language are also included in the regular one. When the regular approximations have been constructed their intersection or overlap is computed. If the resulting set is non-empty then a horizontal or vertical ambiguity is found.

Termination Because the intersection and overlap problems are decidable for regular languages, these operations will always terminate.

Correctness Because of the regular approximation, the intersections or overlaps might contain strings that are not in the original language. Because the regular languages are bigger than the original languages, they might contain more ambiguous strings. In such cases the algorithm thus report false positives. It will never report false negatives because there are no strings removed. All ambiguous strings in the original language are also present in the approximations and they will always be checked. Grammar unfolding can be applied to decrease the number of false positives [29]. All non terminals in the right-hand sides of the productions can be substituted with their productions to create a new and bigger set of productions. These new productions might have smaller approximated regular languages, which might result in smaller intersections and overlaps. The unfolding can be applied multiple times to increase the accuracy even more. Unfortunately there are also grammars for which repetitive unfolding will never gain better results.

Scalability This method clearly gains performance by sacrificing accuracy. Because of the regular approximation its search space becomes bounded and the algorithm will always terminate. The languages of the productions are extended to a form in which they can be searched more easily, but the downside is that extra ambiguous strings might be introduced. By choosing the right approximation algorithm the accuracy and performance can be scale [29].

2.4.5 Schmitz Method

Schmitz [20] describes another ambiguity detection method that uses approximations to create a finite search space. It is also a framework in which various approximation techniques can be used and combined. The algorithm searches for different derivations of the same string in an approximated grammar.

It starts by converting the input grammar into a bracketed grammar. For every production two terminals are introduced: d_i (derivation) and r_i (reduction), where i is the number of the production. They are placed respectively in front and at the end of every production rule, resulting in a new set of productions. The language of this grammar holds a unique string for every derivation of the original grammar. With a homomorphism every bracketed string can be mapped to its corresponding string from the original language [20].When multiple bracketed strings map to the same string, that string has multiple derivations and is ambiguous. The bracketed grammar cannot be ambiguous, because the d_i and r_i terminals always indicate what production rule the included terminals should be parsed with.

The following grammar

$E \rightarrow E + E \mid a$

Converted in bracketed form:

$E \rightarrow d_1\ E + E\ r_1 \mid d_2\ a\ r_2$

The goal of the ambiguity detection method is to find different bracketed strings that map to the same un-bracketed string. To do this a position graph is constructed. This graph contains a node for every position in every sentential form of the bracketed grammar. Their labels are of form a·+a. The edges of the graph are the valid transitions between the positions [29].

There are three types of transitions: shifts (of terminals or non terminals from the original grammar), derivations and reductions. The shift edges are labeled with their respective terminals or non terminals. The derivations are labeled with d_i symbols and the reductions with r_i symbols.

The graph has two sets of special nodes: the start nodes (q_s) and the end nodes (q_f). The start nodes are all nodes with the position at the beginning of a string and the end nodes are nodes with the position dot at the end.

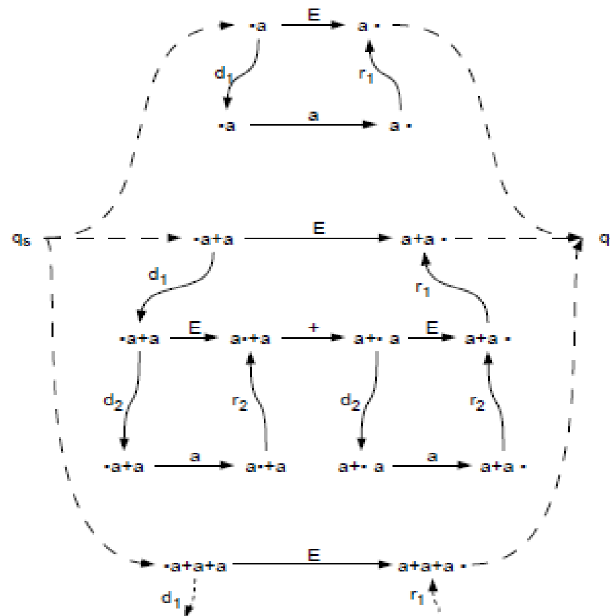


Figure: 1.2 Position Graph [29]

Every path through this graph from a start node to an end node corresponds to a bracketed string, which is formed by the labels of the followed edges. If two different paths that correspond to the same un-bracketed string can be found then the input grammar is ambiguous. Unfortunately, if the language of the input grammar is infinite, then so is its position graph. Finding an ambiguity in it might take forever.

Termination The position graph can be made finite using equivalence relations between the positions. The nodes of the position graph will then represent the equivalence classes of positions, instead of the individual positions. With the right equivalence relation the number of equivalence classes will be finite and the position graph will also be finite. Searching for ambiguities will then always terminate [29].

Correctness The application of the equivalence relations will also result in a new set of edges. For every edge in the previous graph there will be an edge in the reduced graph that connects the equivalence classes of the individual positions it was connected to. The consequence of the alteration of the position graph is that the bracketed and unbracketed grammars are also altered. Transitions that can be

followed from one equivalence class to another might originally not exist between two individual positions of those classes. It is like there are extra transitions added to the first position graph. This corresponds to the addition of extra productions to the bracketed and also the unbracketed grammar. The language of the input grammar is thus also extended, but conservatively. All strings of the original language are also included in the new approximated language. Just as in the method of Brabrand et al. [18] there might be extra ambiguous strings introduced, which will result in false positives. Because the approximation is conservative there can be no false negatives.

Scalability The position graph is finite the searching for two different paths with the same unbracketed string can be done in finite time. The size of the graph depends on the chosen equivalence relation. The number of false positives is usually inversely proportional to the size of the graph. According to Schmitz 'the relation item0 is reasonably accurate'. With the item0 relation all positions that are using the same production and are at the same position in that production are equal. The resulting new positions are the LR(0) items of the grammar[29].

The position graph is then very similar to the LR(0) parsing automaton for the input grammar. The main difference is that it generates all strings that could be parsed by an LR(0) parser without a stack. During normal LR parsing the next state after a reduction is determined by a lookup in the goto table with the reduced non terminal and the state on top of the stack. This is the state from where the derivation of the reduced non terminal was started. The item corresponding to this state is of form $A \rightarrow \alpha.B\beta$. After the reduction of the non terminal B it is only allowed to go to the state that corresponds with $A \rightarrow \alpha B.\beta$. The position graph however allows a derivation transition to every state with an item with the dot right after the B. So if there would also be a production $C \rightarrow B\gamma$, then it is possible to go from position $A \rightarrow \alpha .B\beta$ to $C \rightarrow B\gamma$ by shifting aB. This makes it possible to parse the first half of a string with one part of a production rule, derive and reduce a non terminal, and then parse the remaining half with the end of another production rule. Just like $item_k$ there is also the relation item k, where k is a positive integer [29]. These result in the positions being the LR(k) items of the grammar. The number of LR(k) items usually grows with increasing k, and thus also the size of the position graph. The original bracketed

positions are divided into more equivalence classes. This results in less extra transitions and also less extra potentially ambiguous strings. Searching a bigger position graph however takes more time and space. The accuracy and performance of this method can thus be scaled by choosing the right equivalence relation.

Table: 2.1 Summary of existing methods of ambiguity Detection [30]

Method / Factors	M_1	M_2	M_3	M_4	M_5	M_6
Termination	N	N	N	N	Y	Y
Scalability	N	N	N	Y	N	Y
False Positive	N	-	N	N	N	Y
False Negative	-	N	N	-	Y	N
Ambiguity	Y	N	Y	Y	Y	Y
Unambiguity	N	Y	Y	N	Y	Y
Optimize	N	N	Y	Y	Y	Y

M_1-Gorn Method M_2- Kruse Method M_3 Chung Method
M_4- Schorer's Method M_5- Jampana's Method M_6- Brabrand's Method
Y-Yes N-No

Chapter 3

Ambiguity Detection with Language Approximation

The ambiguity detection algorithm "Ambiguity Checking with Language Approximations" (ACLA) has been proposed by Claus Brabrand et al. [18]. Instead of dealing with the ambiguity problem as a whole, it is separated into two similar problems, namely vertical and horizontal ambiguity. Both problems are undecidable again. At first, we define horizontal and vertical ambiguity. Then, we show that a superset of the original grammar allows an approximation of the ambiguity problem, if the intersection and the overlap operators can be computed on the superset. Finally, we show how regular grammars can be used as such supersets.

3.1. Horizontal and Vertical Ambiguity

Definition: 3.1 A grammar $G = (V, T, P, S)$ is horizontally ambiguous iff it has a production $A \rightarrow \gamma$ with a partition $\gamma \rightarrow \alpha\beta$ such that the languages L(α) and L(β) overlap, i.e. there is a sequence of terminals that could be the ending of L(α) or the beginning of L(β). As a formula [18]:

$$\exists A \in V, (A \rightarrow \alpha\beta) \in P : L_G(\alpha) \bowtie L_G(\beta) \neq \phi$$

Where *overlap* $\bowtie$ of two languages X and Y is defined by

$$X \bowtie Y = \{ xvy \mid x, y \in \Sigma^* \wedge v \in \Sigma^+ \wedge x, xv \in X \wedge vy, y \in Y \}$$

Definition: 3.2 A grammar $G = (V, T, P, S)$ is vertically ambiguous iff it has two different productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ which can be derived to the same word. As a formula [18]:

$$\exists A \in V, (A \rightarrow \alpha), (A \rightarrow \beta) \in \mathrm{P}, \alpha \neq \beta : L_G(\alpha) \cap L_G(\beta) \neq \phi$$

Theorem 3.1 If grammar G is horizontally ambiguous then it is also ambiguous

Proof Assume that G has a horizontal ambiguity in production $A \rightarrow \alpha\beta$ so $L_G(\alpha) \cap L_G(\beta) \neq \phi$. Let w be an element of $L_G(\alpha) \cap L_G(\beta)$. By definition of the overlap operator w=xvy with $|v| \geq 0$, $x, xv \in L(\alpha)$ and $y, vy \in L(\beta)$ holds. Since A exists in at least one parse tree , two different leftmost derivation shown here

$S \Rightarrow^* \gamma A\delta \Rightarrow^* z_1 A \delta \Rightarrow z_1 \alpha\beta\delta \Rightarrow^* z_1 xv\beta\delta \Rightarrow^* z_1 xvy\delta \Rightarrow^* z_1 xvyz_2 = z_1 wz_2$

$S \Rightarrow^* \gamma A\delta \Rightarrow^* z_1 A \delta \Rightarrow z_1 \alpha\beta\delta \Rightarrow^* z_1 x\beta\delta \Rightarrow^* z_1 xvy\delta \Rightarrow^* z_1 xvyz_2 = z_1 wz_2$

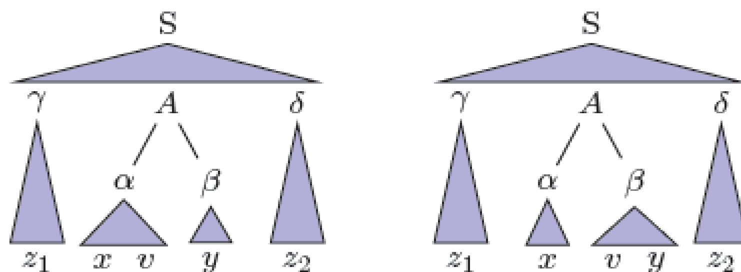


Figure: 3.1 Two parse tree for horizontal Ambiguity [24]

Theorem 3.2 If grammar G is vertically ambiguous then it is also ambiguous

Proof Assume that *G* has a vertical ambiguity in nonterminal *A*. Hence, there are two different productions $A \rightarrow \alpha$ and $A \rightarrow \alpha'$ with M= $L(\alpha) \cap L(\alpha') \neq \phi$. Let $w \in M$ be an element of this intersection. The word *w* can be used to build two different leftmost derivations because by definition of a context-free grammar, *A* exists in at least one derivation.

$S \Rightarrow^* \beta A\gamma \Rightarrow^* xA\gamma \Rightarrow x\alpha\gamma \Rightarrow^* xw\gamma \Rightarrow^* xwy$

$S \Rightarrow^* \beta A\gamma \Rightarrow^* xA\gamma \Rightarrow x\alpha'\gamma \Rightarrow^* xw\gamma \Rightarrow^* xwy$

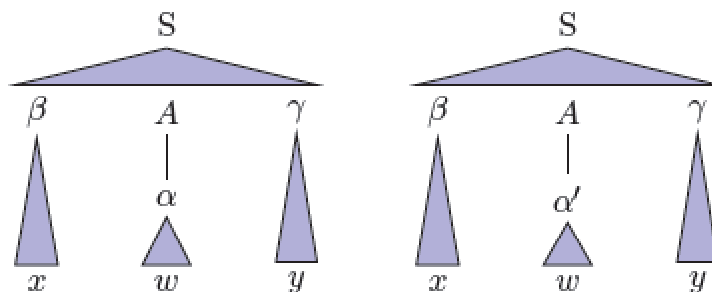


Figure: 3.2 Two parse tree for vertical Ambiguity [24]

Theorem 3.3 If grammar G is ambiguous then it is either horizontally or vertically ambiguous (or both).

Proof Assume that $G = (V, T, P, S)$ is ambiguous. We show by mathematical induction over the height of the derivation tree that G must be vertically or horizontally ambiguous. The start symbol is always a nonterminal so the height of a parse tree is at least 1

- Induction Basis: Tree height = 1

  Only one production $S \rightarrow w_1 ....w_n = w$ has been applied. There can be only one such production, therefore this tree is unambiguous. This does not violate the implication in the induction hypothesis because its antecedent is false.

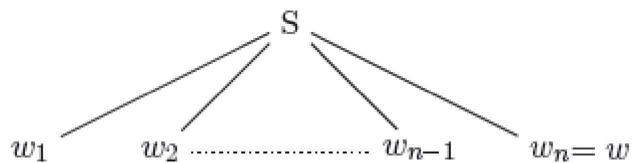


Figure: 3.3 Induction basis(h=1) [24]

- Induction Hypothesis:

  Any parse tree with height $h-1$ or less, which is ambiguous, is either vertically or horizontally ambiguous [24].

- Inductive step:

  Without loss of generality, we assume that all productions have n symbols on the right hand side with n being the largest number of symbols in all productions. Productions having less symbols can be filled up using ε-symbols.

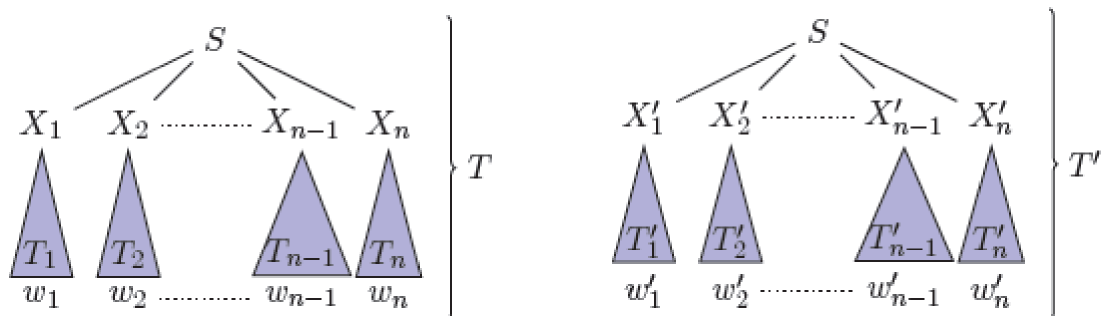


Figure: 3.4 Inductive step of theorem 3.3 [24]

We assumed that grammar G is ambiguous and therefore a word w exists, which has two different parse trees.

$T = (S \Rightarrow X_1.........X_n \Rightarrow^* w_1.......w_n = w)$

$T^{'} = (S^{'} \Rightarrow X_1^{'}...........X_n^{'} \Rightarrow^* w_1^{'}......w_n^{'} = w)$

Furthermore, let h be the height of the higher tree of T and T' .Now there are three different cases we have to handle

Case $\exists i = \{1,.........n\}: X_i \neq X_i^{'}$

(There is difference in top most derivation)

Two different productions $S \rightarrow X_1.............X_n$ and $S \rightarrow X_1^{'}...........X_n^{'}$ have been applied. Hence the definition of vertical ambiguity, we show that the grammar is vertically ambiguous.

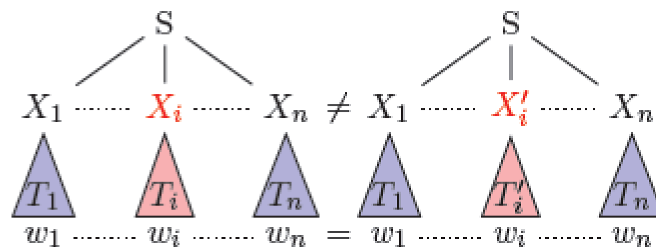


Figure: 3.5 Case of Vertical Ambiguity [24]

Case $\forall\ i = \{1,.......,n\}: X_i = X_i^{'}$ and $\forall i = \{1,.......,n\}: w_i = w_i^{'}$

(The topmost derivation is equal and its subtrees derive the same terminal Sequences)

All the sub trees T_i and $T_i^{'}$ start with the same symbol $X_i = X_i^{'}$ However, since we assumed that $T = T^{`}$ there is at least one i such that $T_i \neq T_i^{'}$ .The height of both trees is less than h and therefore the induction hypothesis can be applied to all of them [24]. When applying the hypothesis to w_i which has two different parse trees T_i and $T_i^{'}$ with starting symbol $X_i = X_i^{'}$, we get that it is vertically or horizontally ambiguous and hence the grammar, which contains these sub trees, too.

Case $\forall i = \{1,........,n\}: X_i = X_i^{'}$ and $\exists$ i={1,......n}: $w_i = w_i^{'}$

(The topmost derivation is equal, but its sub trees derive different sequences of terminals)

Let I be the lowest index such that $w_i = w_i^{'}$ we know that

$w_1.........w_n = w = w_1^{1}.......w_i^{'} w_{i+1}^{'} ....w_n$ 'but $w_1.........w_i \neq w^{'}_1.............w_i^{'}$ and

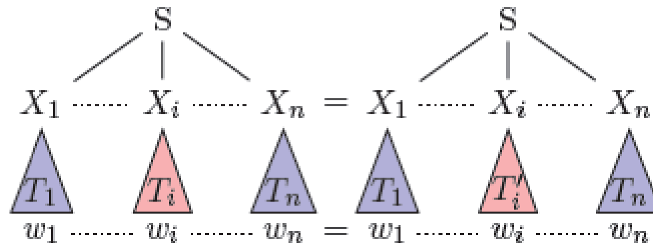


Figure: 3.6 Case of Horizontal Ambiguity [24]

Without loss of generality we can assume that $|w_{i+1}........w_n| \leq |w_1^{'}.........w_i^{'}|$ (otherwise we switch the roles of T and $T^{'}$). Then we also know that $|w_{i+1}........w_n| > |w_{i+1}^{'}.......w_n^{'}|$ holds. The lengths cannot be equal, because w_i would not be different from $w_i^{'}$.

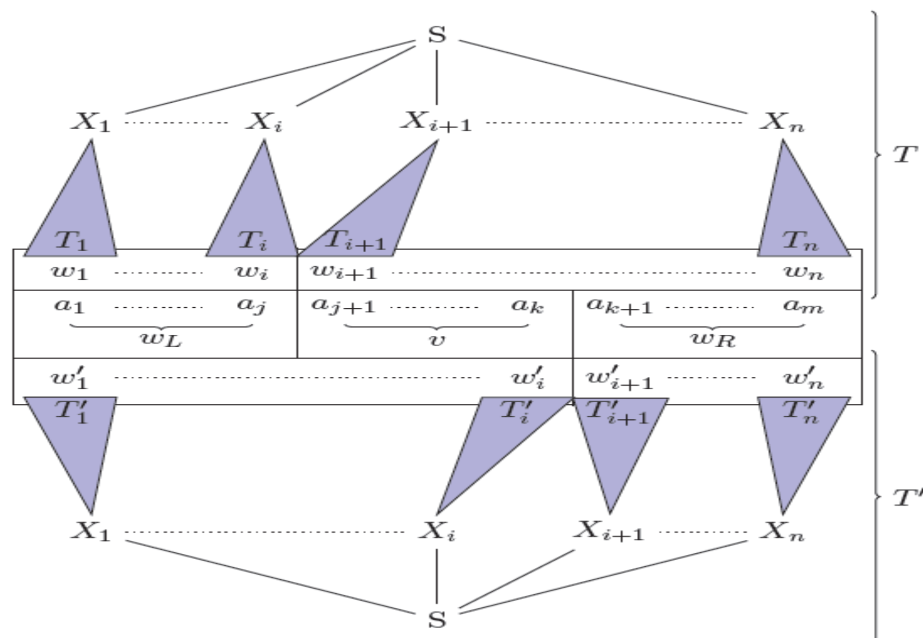


Figure: 3.7 Case of Vertical or Horizontal Ambiguity [24]

The existence of two different parse trees of height $\leq h$ implies that the grammar is vertically or horizontally ambiguous.

Chapter 4
Problem Analysis

In this chapter problem statement and objectives of the thesis are outlined.

4.1 Problem Statement

The ambiguity problem in context free grammar is studies by various researchers from decades, but the definite solution which used for every grammar is not derived. These methods use the parsing techniques or approximation techniques. There are recent works on ambiguity detection approximations by Claus Brabrand et al. [18] that divides the ambiguity problem into horizontal and vertical ambiguity, both are un-decidable as well. They can be approximated by using a regular superset of the original grammar. These techniques are called *Ambiguity Checking with Language Approximations* (ACLA). An approximation algorithm can return up to three values for the ambiguity problem: unambiguous, ambiguous and n/a, if it cannot make a decision

Ambiguity problem occurs in all languages which are presently used for grammar generation. This problem gives incorrect result when the parsing is performed. ACLA Techniques provide the proper termination after the grammar is processed.

The aim of this thesis is to study the notion of ambiguity in context-free grammars and grammars in general. ACLA techniques are divided the ambiguity problem into smaller problem and these provide the better results. This thesis derived the approach to solved ambiguity problem after converting the Context free Grammar into Chomsky Normal Form (CNF) which is much simpler than other methods.

4.2 Objectives

Following are the objectives of this thesis:

- To analyze and the ambiguity problem for context-free grammars.
- Comparison of various ambiguity detection techniques.
- To propose an algorithm for identification of vertical and horizontal ambiguity detection approximations.
- To verify using examples and implementation the proposed technique.

Chapter 5

Proposed Techniques for Ambiguity Detection

This chapter discusses about how the problem stated in previous chapter can be solved with the help of proposed method.

Following steps are used for ambiguity detection in the grammar.

- Convert given Grammar into the Chomsky normal form (CNF).
- Computation of first and last function for production P.
- Identify the productions which have a possibility of Vertical and Horizontal ambiguity.
- Check the Vertical and Horizontal Ambiguity for the productions.

5.1 Conversion of Grammar into Chomsky Normal Form

Following steps are used for converting CFG into CNF.

5.1.1 Removal of Useless Symbol

Useless symbol are the variables which do not appear in any string generated from the starting non terminal.

Algorithm: 5.1 Identify the useless non-terminal in grammar G

Input: A grammar G

Output: Productions which are not reachable from start symbol S.

1. $V_{old} \rightarrow \phi$
2. $V_{new} \leftarrow \{S\}$
3. Repeat step 4 to 7 **While** $V_{old} = V_{new}$ **do**
4. $V_{old} \leftarrow V_{new}$
5. **For** $X \in V_{old}$ **do**
6. **For** (X $\rightarrow$w) $\in$P **do**
7. Add all Variables appearing in w to V_{new}

return V_{new}

Algorithm: 5.2 Remove the useless Non-terminal from G

Input: A grammar G

Output: Variables which generates nothing.

1. $V_{old} \leftarrow \phi$
2. $V_{new} \leftarrow V_{old}$
3. **do**
4. $V_{old} \leftarrow V_{new}$
5. **For** $X \in V$ **do**
6. **For** $((X \rightarrow w) \in P)$ **do**
7. **If** $w \in (T \cup V_{old})^*$ **then**
8. $V_{new} \leftarrow V_{new} \cup \{X\}$
9. **While** $(V_{old} \neq V_{new})$

Return V_{new}

The output grammar does not containing any useless production which is not used to string generation.

5.1.2 Removal of Null Production

Given a grammar G= (V,T,P,S) then the production V $\rightarrow T$ is generate nullable,or there is way to generate the null string, then apply the following algorithms

Algorithm 5.3: CompNullable(G)

Input: A grammar G

Output: Productions which generates null productions.

1 $V_{null} \leftarrow \phi$

2 **do**

3 $V_{old} \leftarrow V_{null}$

4 **For** $X \in V$ **do**

5 **For** $((X \rightarrow w) \in P)$ **do**

6 **If** $w = \varepsilon$ or $w \in (V_{null})^*$ **then**

7 $V_{null} \leftarrow V_{null} \cup \{X\}$

8 **While** $(V_{null} \neq V_{old})$

Return V_{null}

5.1.3 Removal of unit production

A unit production is form of $X \rightarrow Y$ where X,Y $\in V$

Algorithm: 5.4 Identification of unit production

Input: A grammar G

Output:Identify Unit Productions.

1 $P_{new} \leftarrow \{(X \leftarrow Y) \mid (X \rightarrow Y) \in P\}$

2 **do**

3 $P_{old} \leftarrow P_{new}$

4 **For** $(X \rightarrow Y) \in P_{new}$ **do**

5 **For** $(Y \rightarrow Z) \in P_{new}$ **do**

6 $P_{new} \leftarrow P_{new} \cup \{X \rightarrow Z\}$

Return P_{new}

Algorithms: 5.5 Removal of unit production

Input: A grammar G

Output: Production which does not contain the Unit production

1 $U \rightarrow CompUnitPairs(G)$

2 $P \leftarrow P \setminus U$

3 **For** $(X \rightarrow A) \in U$ **do**

4 **For** $(A \rightarrow w) \in P_{old}$

5 $P \leftarrow P \cup \{X \rightarrow w\}$

Return (G)

5.1.4 Making Production with two variables on the right side

It could be possible that the grammar of the form

$X \rightarrow B_1 B_2 B_3 ....B_K$

Rules are converted into CNF as below:

$X \rightarrow B_1 Z_1$

$Z_1 \rightarrow B_2 Z_2$

$Z_{K-3} \rightarrow B_{K-2}Z_{K-2}$

$Z_{K-2} \rightarrow B_{K-1}B_{K}$

Example 5.1 Given Grammar G

$S \rightarrow ASA \mid aB$

$A \rightarrow B \mid S$

$B \rightarrow b \mid \in$

Following steps are used for conversion of CFG to CNF

Step 1: Removal of Useless Symbol

For the given grammar there is no production which is unreachable from the start symbol S

Step 2: Removal of Null Production

For the given grammar there exist some null production

$B \rightarrow \in$ and replace B with A

$A \rightarrow \in$

After Null production Removal, Grammar G is

$S \rightarrow ASA \mid aB \mid a \mid SA \mid AS$

$A \rightarrow B \mid S$

$B \rightarrow b$

Step 3: Removal of Unit Production

After removal of Unit production, grammar G is

$S \rightarrow ASA \mid aS \mid a \mid SA \mid AS$

$A \rightarrow b \mid ASA \mid Ab \mid a \mid SA \mid AS$

$B \rightarrow b$

Step 4: Making Production with two variables on the right side

These production which consists of more than two variable on right hand side such as($S \rightarrow ASA$ and $A \rightarrow ASA$), make them into maximum two variables.

Final CNF form is

$S \to AZ \mid UB \mid a \mid SA \mid AS$

$A \to b \mid AZ \mid UB \mid a \mid SA \mid AS$

$B \to b$

$U \to a$

$Z \to SA$

5.2 Computation of First and Last Function

Following are the functions used in the algorithm

Production ( ): Returns the each production in the grammar.

Terminal (T): Adds T to the set of terminals. An error occurs if T is already a Non terminal symbol..

Nonterminal ( ): Returns the set of non terminal

Isterminal (T): Returns true if T is a terminal; otherwise, returns false.

RHS (P): Returns an iterator for the symbols on the RHS of production P

LHS (P): Returns the non terminal defined by production P

InternalFirst (V): Returns an iterator that visits each production for non terminal V.

VisitedFirst (X): Returns an iterator that visits each occurrence of X in the RHS of all rules.

SymbolDerivesEmpty( ): Check the non terminal symbol derive empty string or not.

5.2.1 Computation of First function

First Function- First function of a non-terminal is the set of first symbols from the right hand side of the production. First function are used from scanning from left to right of the production.

Algorithm 5.6: ComputeFirst (α)

Input- Grammar G'

Output- First functions corresponding to Grammar G.

First (α): Set

1. **For** each A $\in$ Nonterminal() **do** Visited First(A) $\leftarrow$ false
2. First $\leftarrow$ InternalFirst()

3. **Return** (First)

End

Function InternalFirst(Xβ):Set

1. **If** Xβ = empty

then return (ϕ)

2. **If** X =V
3. **then** return ({X})

X is a nonterminal.

4. First $\leftarrow \phi$
5. **If** not VisitedFirst(X)

then

6. VisitedFirst(X)
7. **For** each rhs $\in$ ProductionsFor(X) **do**
8. First $\leftarrow$ First $\cup$ InternalFirst(rhs)
9. **If** SymbolDerivesEmpty(X)
10. **then** First $\leftarrow$ First $\cup$ InternalFirst(β)

Return (ans)

First(α) is computed by invoking FIRST(α). Before any sets are computed, VisitedFirst(X) for each nonterminal A. VisitedFirst(X) is to indicate that the productions of X already participate in the computation of First(α).

3.2.2 Computation of Last function

Last function are computing by reversing the production and calculating the first function which is the last function for the given productions

3.3 Detection of Vertical Ambiguity Productions

Given a grammar G' in CNF form. First of all check all the productions which have the possibility of ambiguity. All productions have been checked with function CheckProduction() and then return the LHS value which consists of non terminals.

Algorithm: 5.6 CheckVProduction()

Input: Grammar G' containing production P_1, P_2----------P_i

Output: Production $P \in P_i$ containing Vertical Ambiguity

Procedure CheckVProduction()

1. **For** each productions ()
2. Visited LHS(P) $\leftarrow$ true
3. Count RHS(P)>1

Return LHS(P)

// This method checks the vertical ambiguity in the Input grammar

Procedure CheckVambiguity()

1. **For** each LHS(P) containing type production for non terminal $A \rightarrow P_1|P_2$
2. RHS(P) $\in$ FirstLast(P_1,P_2)
3. CALL [FirstLast(P_1)]
4. CALL [FirstLast(P_2)]
5. **If** First,Last(P1) $\cap$ First,Last(P2) $\neq \phi$

LHS(P) contain vertical ambiguity

3.4 Detection of Horizontal Ambiguity Production

Given a grammar in CNF form. First of all check all production which have the possibility of ambiguity and then detect the horizontal ambiguity in these production

Algorithm: 5.7 CheckHProduction()

Input: Grammar G' containing productions P_1,P_2--------P_i

Output: Productions $P \in P_i$ containing Horizontal Ambiguity

Procedure CheckHProduction()

1. **For** each production ()
2. Visited LHS(P) $\leftarrow$ true
3. Corresponding RHS(P) $\leftarrow$ true
4. RHS(P) $\in$ [Nonterminal(P) $\geq$ 1]
5. Production(Nonterminal(P)>1)

Return LHS(P)

// This method check the Horizontal ambiguity in the input Grammar

Procedure CheckHambiguity()

1. **For** each production() containing type $A \rightarrow P_1P_2$
2. CallNonterminal()
3. **For** each Nonterminal(P_1,P_2)
4. Call First(P_1,P_2)
5. Call last(P_1,P_2)
6. FirstLast(P_1)=First(P_1) $\cup$ Last(P_1)
7. FirstLast(P_2)=First(P_1) $\cup$ Last(P_2)
8. Production$(P_1)_h$= FirstLast(P_1)
9. Production$(P_2)_h$=FirstLast(P_2)
10. **If** [Production$(P_1)_h \cap$ Production$(P_2)_h$] $\neq \phi$

    LHS (P) contain Horizontal Ambiguity

Example 5.2 Let the Grammar G in CNF form generated by Example 5.1

Step 1: Computation of First and Last function

First (S) = First (A) = First (Z) = First = {a,b}

First (B) = {b}

First (U) = {a}

Last (S) = Last (A) = Last (Z) = {a,b}

Last (B) = {b}

Last (U) = {a}

Step 2: Checking Vertical ambiguity production

$S \rightarrow AZ \mid UB \mid a \mid SA \mid AS$

$A \rightarrow b \mid AZ \mid UB \mid a \mid SA \mid AS$

$B \rightarrow b$

$U \rightarrow a$

$Z \rightarrow SA$

Compare with $A \rightarrow P_1 \mid P_2$

FirstLast(P_1)= FirstLast(AZ) ={a,b}

FirstLast(P_2)=FirstLast(UB)={a,b}

$FirstLast(P_1) \cap FirstLast(P_2) \neq \Phi$

It means Vertical Ambiguity appears in the Grammar.

Step 3: Checking Horizontal Ambiguity Production

Compare with $A \rightarrow P_1P_2$

FirstLast(A)= {a,b}

FirstLast(Z)= {a,b}

$FirstLast(P1) \cap FirstLast(P2) \neq \Phi$

It means Horizontal Ambiguity appears in the Grammar.

Chapter 6
Experimental Results

This section is given for the implementation of the proposed algorithms on various grammars. The proposed algorithms have been applied on set of grammar after converting into CNF form. For the test grammars are included of different sizes, ambiguous grammars which contain horizontal, vertical ambiguity or unambiguous grammar.

6.1 Structure of Input Grammar

Grammars are stored as text files which should end with the .txt extension. They are specified as follows:

- Each line consists of space separated strings which represent productions from a common variable
- The length first string of any line must have one and is a type of variable.
- The corresponding strings on a given line are the right hand sides of productions from the variable for that line
- The first variable on the first line is always the start variable
- Any character which is not a variable, is a terminal

Example 6.1:

Grammar file for the language $\{a^n b^n \mid n > 0\} \cup \{b^n a^n \mid n > 0\}$ has the context free grammar

S → X | Y

X → aXb | ab

Y → bYa | ba

Grammar G is stored in the txt file as follows:

S X Y

X aXb ab

Y bYa ba

6.2 Implementation of the Proposed Technique

The proposed technique for ambiguity detection in context free grammar implemented in Java. The output shown in following snapshots.

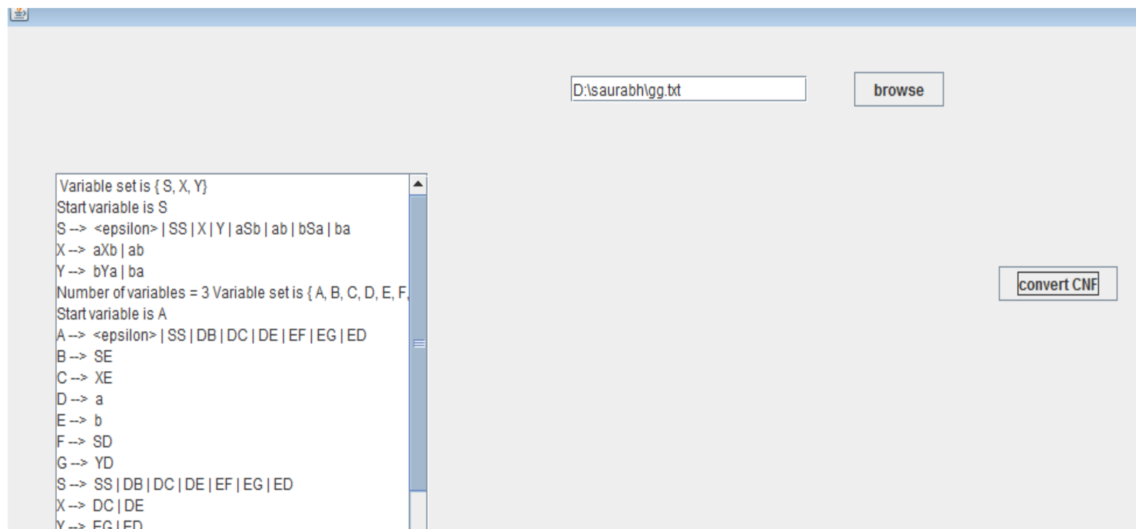


Figure 6.1 Demonstration of input grammar into CNF

Figure 6.1 shows the grammar information of the input grammar file gg.txt and their CNF form. If the input grammar already in CNF then unchanged grammar will be shown. The grammar information such as no of variables, start symbol, no of production is also shown in the output.

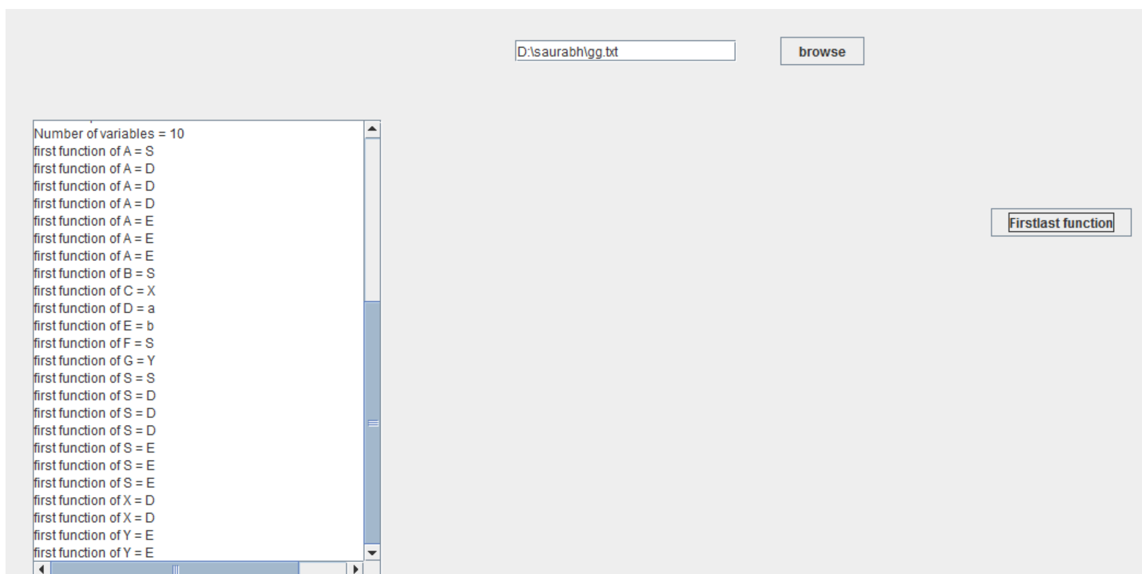


Figure 6.2 First function of the Input grammar

Figure 6.2 shows the First and Last symbol of the input grammar which is now in CNF, with the help of First and Last symbol of the given grammar horizontal and vertical ambiguity will be calculated.

Figure 6.3: Vertical or Horizontal Ambiguity type detection

Figure 6.3 shows the type of ambiguity present in grammar or not, The grammar input could be contain horizontal or vertical ambiguity.

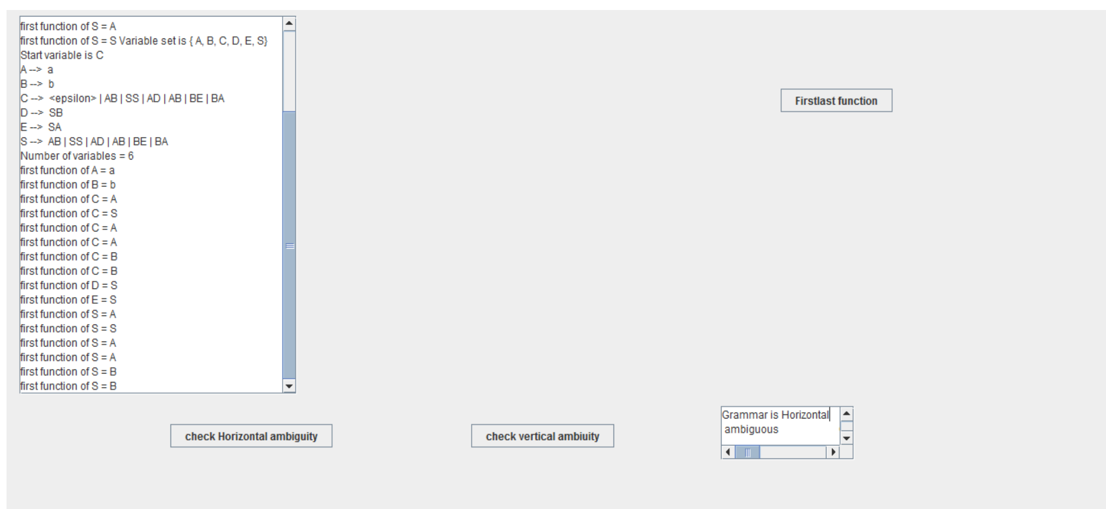


Figure 6.4 Another Scenario the Ambiguity test (Vertical or Horizontal)

Figure 6.4 shown an another grammar is taken as input for checking Horizontal ambiguity.

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 any context-free grammar, the problem of detecting ambiguity is un-decidable. 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

Chapter 7

Conclusion and Future Scope

This chapter discusses the conclusions of the work presented in this thesis. This chapter ends with a discussion of the future direction which can be taken further.

7.1 Conclusion

We have presented a technique for statically analyzing ambiguity of context-free grammars, which is based on a linguistic characterization. This thesis gives the ambiguity detection technique with their background work. The presented algorithms implemented in Java, identify the horizontal and vertical ambiguity for the context free grammar after converting into CNF because CNF form easily implemented for the parsing and beneficial in terms of computation. The presented algorithm is less complex than others and applicable for simple grammar.

7.2 Future Scope

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

References

1. J.E. Hopcroft, R. Motwani, and J. D. Ullman, "Introduction to Automata Theory, Languages, and Computation", Pearson Education Asia, Inc. Delhi, India, 2001.
2. D. I. A. Cohen, "Inroduction to Computer Theory", John Wiley & Sons, Inc. Canada, 1986.
3. S. C. Johnson, B. W. Kernighan, and M. D. Mcllroy, "YACC: YetAnother compiler-compiler", UNIX Programmar's manual, Bell laboratories, 7th edition murray hills, 1978.
4. D. Parks, Appalachian State University, Lecture Notes, "Chapter 16: Non-Context-Free Languages", 2004.
5. W. Kuich and A. Salomaa, "Semirings, Automata, Languages", Springer-Verlag, London, UK, 1985.
6. S. Ginsburg and J. Ullian, "Ambiguity in Context Free Languages" , Journal of the ACM (JACM*),* vol. 13, no. 1, pp. 62-89, 1966.
7. A. Frank, Tracy H. King, Jonas Kuhn, and John Maxwell, Stanford University, "Optimality Theory Style Constraint Ranking in Large-Scale LFG Grammars", 1998.
8. L. Paulson, Stanford University, "A Compiler Generator for Semantic Grammars", Ph.D. Dissertation Report, Department of Computer Science, 2005.
9. A. Gacek, University of Minnesota, IT Lab Forums, "Ambiguous Method Call in Java", 2005.
10. J. C. Cleaveland and R. C. Uzgalis, "Grammars for Programming Languages", Elsevier, New York, 1977.
11. B. Wilson, The University of New South Wales, "Grammars and Parsing", 2004.
12. S. Gorn, "Detection of generative ambiguities in context-free mechanical languages", J. ACM, vol.10, no.2, pp.196–208, 1963.

13. B. S. N. Cheung and Robert C. Uzgalis, “Ambiguity in context-free grammars”, In SAC ’95: Proceedings of the 1995 ACM symposium on Applied computing, pp. 272–276, New York, USA, 1995.
14. F. W. Schroer, “AMBER, an ambiguity checker for context-free grammars”, Technical report, 2001.
15. J. Earley. An efficient context-free parsing algorithm. Communications of the ACM, vol.13, no.2, pp. 94–102, 1970.
16. S. Jampana. “Exploring the problem of ambiguity in context-free grammars”, Master’s thesis, Oklahoma State University, 2005
17. D. E. Knuth, “On the translation of languages from left to right. Information and Control”, vol.8, no.6, pp. 607–639, 1965.
18. C. Brabrand, R. Giegerich, and A. Møller, “Analyzing ambiguity of context free grammars”, Proc. 12th International Conference on Implementation and Application of Automata, CIAA, 2007.
19. M. Mohri and M. Nederhof, “Regular approximations of context-free grammars through transformation”, Robustness in Language and Speech Technology, chapter 9, pp.153–163, Kluwer Academic Publishers, 2001.
20. S. Schmitz. “Conservative ambiguity detection in context-free grammars”, ICALP: 34th International Colloquium on Automata, Languages and Programming, vol. 4596, pp. 692–703, Springer, 2007.
21. A. V. Aho and J. D. Ullman, “The Theory of Parsing, Translation, and Compiling”, Prentice Hall Professional Technical Reference, 1972.
22. B.S.N. Cheung, “A Theory of automatic language acquisition”, PhD thesis, University of Hong Kong, 1994.
23. S. Schmitz, “An experimental ambiguity detection tool”, Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA’07), Braga, Portugal, 2007.
24. M. Kruse, “Ambiguity Detection for Context free grammar in Eli”, Bechlor’s thesis, University of Paderborn, 2008.
25. F. W. Schroer, “ACCENT, a compiler compiler for the entire class of context-free grammars”, second edition, Technical report, 2006.
26. H. J. S. Basten, “The usability of ambiguity detection methods for context-free grammars”, In 8th Workshop on Language Descriptions, Tools and Applications, 2008

27. J. Eickel and M. Paul, “The Parsing and Ambiguity Problem for Chomsky Languages”, Formal Language Description Languages for Computer Programming, North-Holland Publishing Company, London, UK, 1966.
28. J. Aycock and R. N. Horspool, Schrodinger’s token. Software: Practice & Experience, vol.31, no.8, pp. 803–814, 2001.
29. H. J. S. Basten, “Ambiguity Detection methods for context free grammars”, Master’s thesis, University of Amsterdam, 2007.
30. H. M. Pandey, “Case study on Ambiguity Detection Methods using some set of Grammars”, Pacific Journal of Science and Technology, vol.13, no.1, pp. 277-286, 2012.

Published

1. S. K. Jain and A. Kumar, "Ambiguity Detection Algorithm for Context free Grammar", International Journal of computer science and Technology (IJCST), ISSN 0976-8491, June, 2013.

Communicated

2. S.K. Jain and A. Kumar, "Comparison between Ambiguity Detection Techniques for Context free Grammar", International Journal of computer application, vol.75, August, 2013.