ISBN: 978-93-83758-09-8

Different Approaches for Ambiguity Detection

Garima Anand
BBSBEC, Fatehgarh Sahib, Punjab

vrvg2006@gmail.com

Abstract: Context-free grammars(CFGs) are widely used and are suitable for the definition of a wide range of programming languages. A common requirement is that grammar should be unambiguous. The problem of ambiguity in CFGs is undecidable in general, but various methods have been developed to detect ambiguity. In this paper, different ambiguity detection methods are compared.

Keywords: **Context-free grammars; automata; ambiguity; ambiguity detection methods.**

I. INTRODUCTION

Context-free grammars (CFGs) are widely used in various fields, like for instance programming language development, natural language processing, or bioinformatics(Basten, 2010). Context-free grammars are an important form of source code, both for the development of source code analysis and manipulation tools, and for prototyping (domain specific) languages (Klint, Lämmel, and Verhoef, 2005).In bioinformatics, context-free grammars in various guises have important applications, for example in sequence comparison, motif search, and RNA secondary structure analysis (Giegerich, Meyer, and Steffen, 2004) and (Durbin, Eddy, Krogh, and Mitchison, 1998). They are suitable for the definition of a wide range of languages, but their possible ambiguity can hinder their use. The presence of ambiguities in a context-free grammar hampers the reliability or the performance of the tools build from it(Schmitz, 2007).

A CFG is said to be ambiguous if it allows multiple derivations for the same string. If a grammar is ambiguous this often indicates a grammar bug that needs to be fixed. It is, however, very hard to establish whether a grammar is ambiguous or not. Ambiguity detection tools are used to statically try and find ambiguities. To be practical, such tools should (1) provide feedback that is comprehensible to the grammar developer, and (2) come up with an answer within reasonable time(Basten and Storm, 2010).Despite the fact the CFG ambiguity problem is undecidable in general(Basten, 2010), (Cantor, 1962), (Chomsky and Sch¨utzenberger, 1963), (Floyd, 1962) ,various detection schemes exist. They can be divided into two categories:

1) Exhaustive searching: (Axelsson, Heljanko, and Lange, 2008) start generating all sentences in the language of a grammar and check if they have multiple parse trees. If an ambiguous sentence is found, the productions involved in the ambiguity can be derived from the parse trees. Generating strings of increasing length is of exponential complexity and will not terminate if the grammar is unambiguous. These methods are accurate and suffer from the problem of halting.
2) Approximative searching: (Schmitz, 2007) the grammar is approximated into an alternative form(Brabrand, Giegerich, and Møller, 2010) that that can be analyzed in finite time. These methods always terminate but at the expense of precision. The ambiguity reports of these tools are hard to understand. The approximations may result in two types of errors: *false negatives* if some ambiguities are left undetected, or *false positives* if some detected "ambiguities" are not actual ones. They do not halt.

II. PRELIMINARIES

Context-Free Grammars:

A context-free grammar G is a 4-tuple (N, T, P, S) consisting of:
– N, a finite set of *non terminals*,
– T , a finite set of *terminals* (the alphabet),
– P, a finite subset of $N \times (N \cup T)^*$, called the *production rules*,
– S, the *start symbol*, an element from N.
A production (A, α) in P is written as $A \rightarrow \alpha$. We use the function pid :$P \rightarrow$N to relate each production to a unique identifier. An *item* indicates a position in the right hand side of a production using a dot. Items are written like $A \rightarrow \alpha.\beta$.
The relation $\Rightarrow$ denotes direct derivation, or derivation in one step. Given the string $\alpha B\gamma$ and a production rule $B \rightarrow \beta$, we can write $\alpha B\gamma \Rightarrow \alpha\beta\gamma$

ISBN: 978-93-83758-09-8

(read $\alpha B\gamma$ directly derives $\alpha\beta\gamma$). The symbol $\Rightarrow^*$ means "derives in zero or more steps". A sequence of derivation steps is simply called a *derivation*. Strings in V^* are called *sentential forms*. We call the set of sentential forms that can be derived from S of a grammar G, the *sentential language* of G, denoted $S(G)$. A sentential form in T^* is called a *sentence*. The set of all sentences that can be derived from S of a grammar G is called the *language* of G, denoted $L(G)$.

We assume every nonterminal A is *reachable* from S, that is $\exists\ \alpha A\beta \in S(G)$. We also assume every nonterminal is *productive*, meaning $\exists u : A \Rightarrow^* u$.

The *parse tree* of a sentential form α describes how α is derived from S, but disregards the order of the derivation steps.

Automata:

An automaton is a 5-tuple < Q , Σ, q_0 , δ, A >

Where, Q: a non-empty finite set of states present in the finite control($q_0, q_1, \ldots q_n$)

∑: a non-empty finite set of input symbols

q: starting state, q ε Q

F: a non-empty set of final or accepting states

δ : transition function

δ is a function. Thus for each state q of Q and for each symbol a of Σ, δ(q, a) must be specified.

Ambiguity:

A grammar is called ambiguous if at least one sentence in its language can be derived from the start symbol in multiple ways. Such a sentence is also called ambiguous. Figure 1 shows two derivation trees of an ambiguous sentence of the following grammar:

A→ A + A / A*A/ 0/ 1/ 2/ 3/ 4/ 5/ 6/ 7/ 8/ 9.

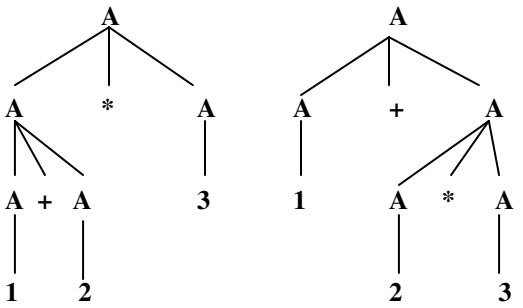

Figure1: example of two parse trees of the sentence '1+2*3'

Vertical and horizontal ambiguity:

A vertically ambiguous grammar.

```
Z : A 'y'
    |
  | 'x' B ;
A : 'x' 'a' ;
B : 'a' 'y' ;
```

The string xay can be parsed in two ways by choosing either the first or the second production of Z. The name *vertical* ambiguity comes from the fact that productions are often written on separate lines as in this example.

A horizontally ambiguous grammar.

```
Z : 'x' A $ B ;
A : 'a'
  | '' ;
B : 'a' 'y'
  | 'y' ;
```

Also here, the string xay can be parsed in two ways, by parsing the a either in 'x' A (using the first production of A and the second of B) or in B (using the second production of A and the first of B). Here, the ambiguity is at a split-point between entities on the right-hand side of a particular production, hence the name *horizontal* ambiguity.

III. DETECTION METHODS

1) *RU Testing*: The Regular Unambiguity(RU) Test was introduced by Schmitz. The RU Test is an approximative test for the existence of two parse trees for the same string, allowing only false positives (Schmitz, 2007).

2) *NU Testing:* This approximative method always terminates and can provide relatively accurate results in little time. The method can be tuned to trade accuracy for performance. Its memory usage grows to impractical levels much faster than its running time. For example, with the best available accuracy, it took more than 3Gb to fully analyze Java(Basten and Vinju, 2010). A downside is that its reports can be hard to understand due to their abstractness. It enables the identification of a larger set of irrelevant parse trees. From these parse trees, we can also identify a larger set of harmless production rules and tree patterns. The approximation it applies is always conservative, so it can only find a grammar to be *unambiguous* or *potentially ambiguous*(Basten, 2010).

3) *AMBER:* The exhaustive derivation generator Amber (Schr¨oer, 2001) was the most practical in finding ambiguities for real programming languages. The main reasons for this are its accurate reports that contain examples of ambiguous strings, and its impressive efficiency.

ISBN: 978-93-83758-09-8

But its drawback was its possible non-termination. It uses an Earley parser (Earley, 1970) 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. 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. 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. AMBER will not terminate on recursive grammars unless a maximum string length is given. With its default parameters this method can correctly identify ambiguities, but it cannot report non-ambiguity. 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. This method is scalable in two ways: 1) Sentential forms holding nonterminals can also be compared, and 2) the percentage of possible derivations to expand each level is adjustable.

The ambiguity report returned is hard to interpret.

4) AMBIDEXTER: It combines exhaustive and approximative searching to benefit from both their strengths(Basten and Storm, 2010). The goal is to produce precise and comprehensible ambiguity reports as fast as possible. We use approximative filtering to narrow down the search space for an exhaustive checker. This also allows us to detect both ambiguity and unambiguity. The tool operates in two stages:

4.1) Harmless production filtering: harmless productions are productions that cannot be involved in ambiguity. Using an extension of the approximative technique of (Klint, Lämmel, and Verhoef, 2005) such productions are identified and removed from the grammar.

4.2) Derivation generator: for the productions that are not identified as harmless, an exhaustive derivation generator is applied to detect remaining ambiguities.

As a result, AMBIDEXTER leverages the strengths of both approaches to ambiguity detection:

- Unambiguity is detected if all productions are identified as harmless.
- Comprehensible ambiguity reports are produced as a consequence of employing a derivation generator.
- Performance is improved because the production filtering reduces the derivation generator's search space.

5) LR(k) and LR-Regular Testing : One of the strongest ambiguity tests available is the LR-Regular condition(Heilbrunner, 1983). Instead of merely checking the k next symbols of lookahead, a LRR parser considers regular equivalence classes on the entire remaining input to infer its decisions. Practical implementations (Farr´e and G´alvez, 2001) and (Bermudez and Schimpf, 1990) of the LRR parsing method actually compute, for each inadequate LR state, a finite state automaton that attempts to discriminate between the x and z regular lookaheads. 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. 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. 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. 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. 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. 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

ISBN: 978-93-83758-09-8

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.

IV. COMPARISON OF DETECTION METHODS

The comparison of RU test, AMBER, NU test and Ambidexter is shown in table 1.

Table 1: comparison of detection methods

Parameter	**RU**	**AMBER**	**NU**	**AMBIDEXTER**
Type	Approximative	Exhaustive	Approximative	Combination of both
Accuracy	Less	High	More than RU	High
Guarantee of termination	No	No	Yes	Yes
Memory Requirements	Less	Less	High	Less
Results delivered	False positives	Ambiguities	Unambiguities or potential ambiguities	Ambiguities and unambiguities
Reports generated	Abstract	Abstract	Abstract	Detailed

Type: RU and NU are approximative methods of ambiguity detection. AMBER is based on exhaustive method of ambiguity detection. AMBIDEXTER combines exhaustive and approximative searching to benefit from both their strengths.

Accuracy: RU is not much accurate method, NU is more accurate than RU. AMBER and AMBIDEXTER provide higher accuracy.

Termination: There is no guarantee of termination in RU and AMBER. On the other hand, termination is guaranteed in NU and AMBIDEXTER.

Memory Requirements: NU consumes much more memory than all other methods.

Results Delivered: RU method reports false positives. AMBER is used to find ambiguities only. NU can only find a grammar to be unambiguous or potentially ambiguous. AMBIDEXTER can find both ambiguities and unambiguities.

Reports Generated: RU, AMBER and NU provide abstract reports which are hard to understand. AMBIDEXTER generates detailed reports.

V. CONCLUSIONS

Context-free grammars are useful in various fields but their possible ambiguity can hinder their uses. This paper gives an overview of some ambiguity detection methods and compares all of them. To be able to efficiently use the context-free grammars, some new approach is required which would provide more accurate results and in which the termination is guaranteed.

VI. REFERENCES

[1]. Axelsson, R., Heljanko, K., Lange, M. “Analyzing context-free grammars using an incremental SAT solver,” Aceto, L., Damg˚ard, I., Goldberg, L.A., Halld´orsson, M.M., Ing´olfsd´ottir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, Vol. 5126, pp 410–422.

[2]. Basten, H.J.S. ”Tracking down the origins of ambiguity in context-free grammars ” *Theoretical Aspects of Computing*, LNCS, 2010,Vol. 6255, pp 76–90.

[3]. Basten, H.J.S., Vinju, J.J. “Faster ambiguity detection by grammar filtering,” *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications*, ACM, 2010.

[4]. Basten H.J.S. and Storm T. van der. “AmbiDexter: Practical Ambiguity Detection, Tool Demonstration,” Tenth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2010), Timisoara, Romania.

[5]. Brabrand C., Giegerich R., and Møller A. “Analyzing ambiguity of context-free grammars,” *Sci. Comput. Program, 2010,* pp 176–191.

[6]. Cantor, D.G. “On the ambiguity problem of Backus systems,” *Journal of the ACM 9(4)*, 1962, pp 477–479.

[7]. Chomsky, N., Sch¨utzenberger, M. “The algebraic theory of context-free language,”. *Braffort, P. (ed.) Computer Programming and Formal Systems*, 1963, pp 118–161.

[8]. Durbin Richard, Sean R. Eddy, Anders Krogh, and Graeme Mitchison, (1998). ”Biological Sequence Analysis,” *Cambridge University Press*.