The Usability of Ambiguity Detection Methods for Context-Free Grammars

H.J.S. Basten[1]

Centrum voor Wiskunde en Informatica
P.O. Box 94079, NL-1090 GB
Amsterdam, The Netherlands

Abstract

One way of verifying a grammar is the detection of ambiguities. Ambiguities are not always unwanted, but they can only be controlled if their sources are known. Unfortunately, the ambiguity problem for context-free grammars is undecidable in the general case. Various ambiguity detection methods (ADMs) exist, but they can never be perfect. In this paper we explore three ADMs to test whether they still can be of any practical value: the derivation generator AMBER, the LR(k) test and the Noncanonical Unambiguity test. We benchmarked their implementations on a collection of ambiguous and unambiguous grammars of different sizes and compared their practical usability. We measured the accuracy, termination and performance of the methods, and analyzed how their accuracy could be traded for performance.

Key words: Ambiguity detection methods, context-free grammars, practical usability, LR(k) test, AMBER, Noncanonical Unambiguity test

1 Introduction

Generalized parsing techniques allow the use of the entire class of context-free grammars (CFGs) for the specification of programming languages. This grants the grammar developer the freedom of structuring his grammar to best fit his needs. He does not have to squeeze his grammar into LL, LALR or LR(k) form for instance. However, this freedom also introduces the danger of unwanted ambiguities.

Some grammars are designed to be completely unambiguous, while others are intended to contain a certain degree of ambiguity (for instance programming languages that will be type checked after parsing). In both cases it is

[1] Email: H.J.S.Basten@cwi.nl

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

important to know the sources of ambiguity in the developed grammar, so they can be resolved or verified.

Unfortunately, detecting the (un)ambiguity of a grammar is undecidable in the general case [4,6,5]. Still, several Ambiguity Detection Methods (ADMs) exist that approach the problem from different angles, all with their own strengths and weaknesses. A straightforward one is to start generating all sentences of the grammar's language and checking them for ambiguity. The results of this method are 100% correct, but its problem is that it might never terminate. Other methods test for inclusion in unambiguous grammar classes (LALR, LR(k), LR Regular, etc.), but these do not cover the entire set of unambiguous grammars. More recent methods (Noncanonical Unambiguity [12], Grambiguity [3]) search conservative approximations of a grammar or its language, leaving the original ambiguities intact. They are able to terminate in finite time, but at the expense of accuracy.

All these different characteristics result in differences in the practical usability of the ADMs. Whether a method is useful in a certain situation also depends on other factors like the tested grammar, the parameters of the method, the computation power of the used PC, the experience of the grammar developer, etc. In this paper we investigate the practical usability of three ADM implementations in a series of use cases and compare them to each other. The investigated implementations are: AMBER [14] (a derivation generator), MSTA [10] (a parse table generator used as LR(k) test [9]) and a modified version of Bison that implements the Noncanonical Unambiguity test [13].

Overview

In section 2 we describe the criteria for practical usability and how they were measured. In sections 3 to 5 we discuss the measurement results of the three investigated methods and analyze their practical usability. The methods are compared to each other in section 6. Section 7 contains a short evaluation and we conclude in section 8.

2 Comparison framework

In order to measure and compare the practical 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.

2.1 Criteria for Practical Usability

Termination

Methods that apply exhaustive searching can run into an infinite search space for certain grammars. They might run forever, but could also take a very long time before terminating. In practice they will always have to be halted at some point. To be practically usable on a grammar, a method has to produce an answer in a reasonable amount of time.

Accuracy

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.

Performance

Knowing 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. How well a method performs on an average desktop PC also influences its usability.

Usefulness of reports

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.

Scalability

Some ADMs can be executed with various levels of accuracy. There is usually a trade-off between their accuracy and performance. 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.

2.2 Measurements

We tested the implementations of the methods for these criteria on three collections of ambiguous and unambiguous grammars, named *small*, *medium* and *large*. The small collection contained 84 'toy' grammars with a maximum size of 17 production rules. They were gathered from (parsing) literature and known ambiguities in existing programming languages. This way various cases that are known to be difficult or unsolvable for certain methods are tested and compared with the other methods. Also some problematic grammar constructs, like the dangling else, are included.

The medium and large collections were formed of grammars from the real life languages HTML[1], SQL[2], Pascal[3], C[3] and Java[4]. Their respective sizes were 29, 79, 176, 212 and 349 productions rules, with the line between medium and large drawn at 200 productions. Of each grammar (except HTML) five ambiguous versions were created through minor modifications. Again, some of the ambiguities introduced were common ambiguous grammar constructs. Complete references of the grammars and modifications made can be found in [1].

The termination, accuracy and performance of an ADM are closely related. To measure the accuracy of a method it first has to terminate. How long this takes is determined by its performance. We measured the accuracy of the methods on the collection of small grammars to minimize computation time and memory usage.

The termination and performance of an ADM are more relevant in relation to real life grammars, so we measured them on the collection of medium and large grammars. For each grammar we measured the computation time and needed virtual memory of the implementations. The PC used was an AMD Athlon 64 3500 with 1024 MB of RAM (400DDR) running Fedora Core 6. Different time limits were used to test if the methods would be usable as a quick check (5 min.) and a more extensive check (15 hrs.). This results in the following four use cases:

Use case	**Grammar size**	**Time limit**
1. Medium grammars - quick check	$\|P\| < 200$	$t < 5$ m.
2. Medium grammars - extensive check	$\|P\| < 200$	$t < 15$ h.
3. Large grammars - quick check	$200 \leq \|P\| < 500$	$t < 5$ m.
4. Large grammars - extensive check	$200 \leq \|P\| < 500$	$t < 15$ h.

2.3 Analysis

From the accuracy and performance measurements of each ADM we have analyzed the scale with which its accuracy and performance can be exchanged. We also analyzed the usefulness of the reports of the methods, but only as a theoretical survey.

We finally analyzed the practical usability of the ADMs in each of the stated use cases. For each ADM we determined their critical properties that

[1] Taken from the open source project GraphViz, http://www.graphviz.org/

[2] Taken from the open source project GRASS, http://grass.itc.it/

[3] Taken from the comp.compilers FTP, ftp://ftp.iecc.com/

[4] Taken from GCJ: The GNU Compiler for Java, http://gcc.gnu.org/java/

were most decisive for this, and tested if their values were within the constraints of the use cases. Our main focus will be on the ambiguous grammars. We will assume that a grammar developed for a generalized parser has a high chance of being ambiguous the first time it is tested.

3 AMBER

AMBER is a so called derivation generator. It uses an Earley parser to generate sentences of a grammar up to a certain length. With a depth-first search of all valid parse actions it builds up derivations of strings in parallel. When a sentence is completed and has multiple derivations, an ambiguity is found. The string and its derivations are presented to the user, which is a very useful ambiguity report.

We have tested AMBER in its default mode and in *ellipsis* mode. This latter mode also checks the sentential forms of incomplete derivations for ambiguity, in stead of only the actual sentences of a grammar's language. This way it might find ambiguities in strings of shorter length.

3.1 Measurements and Analysis

We have iteratively executed the two modes of AMBER on the grammars with increasing maximum string length. If it found an ambiguity then we stopped iterating. In this way it checks every sentence of the language of a grammar, and can never miss an ambiguity. On the small ambiguous grammars both modes had an accuracy of 100%.

The downside of this exhaustive checking is that it will never terminate on unambiguous cyclic grammars, and that it might take very long on larger ambiguous grammars. However, the ellipsis mode terminated on all medium grammars within both time limits, and the default mode on all but one. This made them highly usable as both a quick check and an extensive check.

The large grammars were a bigger challenge for the tool's computation speed. On this collection both modes were only moderately usable as a quick check, but still very usable as an extensive check. Because of its depth first searching AMBER is very memory efficient. In all cases it consumed less than 7 MB.

The ellipsis mode always took more time than the default mode for the same maximum string length. It only needed a smaller string length for 4 grammars of the 17 medium and large grammars that both modes terminated on. Most of the nonterminals in our grammars could thus derive a string of a very short length. On all but 2 grammars the default mode terminated the fastest, making it the most practically usable of the two.

By our definition, AMBER is not scalable by using the default or ellipsis mode. The termination times of both modes varied, but their accuracy is the same.

4 LR(k) Test

One of the first tests for unambiguity was the LR(k) test by Knuth [9]. If an LR(k) parse table without conflicts can be constructed for a grammar, every string of its language is deterministically parsable and thus unambiguous. To test for LR(k)-ness of a grammar we used the parser generator MSTA [10]. We ran it iteratively on a grammar with increasing k, until the test succeeded or the set time limit was reached. This test has a fixed precision and is therefore not scalable.

If running this method is aborted then it remains uncertain whether the investigated grammar is really ambiguous or simply not LR(k). In some cases the conflicts in the intermediate parse tables might offer clues to sources of ambiguity, but this can be hard to tell. The conflict reports are not grammar oriented, which makes them fairly incomprehensible. The numerous posts about them in the `comp.compilers` newsgroup also show this.

4.1 Measurements and Analysis

MSTA did not terminate on the ambiguous small grammars for values of k as high as 50. On the unambiguous small grammars it terminated in 75% of the cases. These reports thus had an accuracy of 100%.

The computation time and memory consumption of the test both grew exponentially with increasing k. However, the computation time reached an unpractical level faster. The highest memory consumption measured was 320 MB on an ambiguous SQL grammar with $k = 6$.

The maximum values of k that could be tested for within the time limits of the extensive check were 6 for the medium grammars, and only 3 for the large grammars. This should not be a problem, because chances are minimal that a grammar that is not LR(1) is LR(k) for higher values of k [7]. However, experience also shows that grammars written for generalized parsers are virtually never suitable for deterministic parsing. They are aimed to best describe the structure of their language, instead of meeting specific parser requirements [7]. The LR(k) test will probably not be a very useful ambiguity detection method for them.

5 Noncanonical Unambiguity Test

Schmitz' 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.

In the other case it remains uncertain whether the grammar is really ambiguous or not. For each ambiguous string in the approximated language, the investigated tool reports the conflicts in the constructed NFA. They resemble those of the LR(k) test and are also hard to trace in the direction of an ambiguity. These ambiguity reports are not very useful, especially if they contain a high number of conflicts.

The accuracy of the test depends on the used approximation technique. Stricter approximations are usually more accurate, but result in larger NFAs. We tested an implementation [13] of the NU test, which offered LR(0), SLR(1) and LR(1) precision. Their NFAs resemble the LR(0), SLR(1) or LR(1) parsing automata of a grammar, but without the use of a stack. Those of LR(0) and SLR(1) have the same amount of nodes, but the latter is more deterministic because the transitions are constrained by lookahead. The LR(1) automata are fairly bigger.

5.1 Measurements and Analysis

The LR(0), SLR(1) and LR(1) precisions obtained respective accuracies of 61%, 69% and 86%. The LR(1) precision was obviously the most accurate, but also had the largest NFAs. The tool could barely cope with our large grammars, running out of physical memory (1GB) and sometimes even out of virtual memory. When its memory consumption did stay within bounds, its computation time was very low. It remained below 4 seconds on the small and medium grammars. The LR(0) and SLR(1) precisions tested all grammars under 3 seconds, needing at most 68 MB of memory.

Comparing the three precisions of the tool, LR(1) was the most practical on the grammars of our collection. It was pretty usable as both a quick check and extensive check on the medium grammars. On the large grammars it was only moderately usable as an extensive check, because of its high memory usage. The other two precisions were not convincing alternatives, because of their high number of incomprehensible conflicts. The accuracy and performance of the investigated tool could thus be scaled, but only in large intervals. A precision between SLR(1) and LR(1) might be a solution for this, which Schmitz reckons to be LALR(1).

6 Comparison

In the previous three sections we have discussed the measurement results of the three methods. They are summarized in table 1. From these results we have analyzed the practical usability of the methods in each of the stated use cases. In this chapter we will compare the methods to each other. Table 2 presents a (subjective) summary of this comparison.

Usability criteria	**AMBER**		**MSTA**	**Noncanonical Unambiguity**		
	Default mode	Ellipsis mode	LR(k) test	LR(0) precision	SLR(1) precision	LR(1) precision
Accuracy						
• ambiguous	100 %	100 %	n.a.	100 %	100 %	100 %
• unambiguous	n.a.	n.a.	100 %	61 %	69 %	86 %
Performance [1]						
• computation time	−	− −	− −	++	++	++
• memory consumption	++	++	−	++	++	−
Termination (amb)						
Use cases:						
1. medium/quick	90 %	100 %	0 %	100 %	100 %	100 %
2. medium/extensive	100 %	100 %	0 %	100 %	100 %	100 %
3. large/quick	60 %	50 %	0 %	100 %	100 %	20 %
4. large/extensive	80 %	70 %	0 %	100 %	100 %	70 %
Termination (unamb)						
• all 4 use cases	0 %	0 %	100 %	100 %	100 %	100 % [2]
Usefulness of output [1]	++		−	−		

[1] Scores range from − − to ++

[2] Except 50 % in use case 3 (large/quick).

Table 1
Summary of measurement results

AMBER was the most practically usable ADM in all four use cases. Its ambiguity reports are correct and very helpful. Its has exponential performance, but still managed to find most ambiguities within the set time limits. Its biggest drawback is its nontermination on unambiguous grammars. AMBER's ellipsis mode was not superior to the default mode. It is able to find ambiguities in strings of shorter length, but usually took more time to do so.

The NU test with LR(1) precision was also helpful on grammars smaller than 200 productions. It offered a pretty high accuracy, guaranteed termination and fast computation time. However, it suffered from high memory consumption and incomprehensible reports. On the large grammars it started swapping or completely ran out of virtual memory. The LR(0) and SLR(1) precisions were no real alternatives because their high number of conflicts.

Use case	**AMBER**		**MSTA**	**Noncanonical Unambiguity**		
	Default mode	Ellipsis mode	LR(k) test	LR(0) precision	SLR(1) precision	LR(1) precision
1. medium/quick	+++	+++	− −	+/−	+	++
2. medium/extensive	+++	+++	− −	+/−	+	++
3. large/quick	+/−	+/−	− −	− − −	− −	− −
4. large/extensive	++	+	− −	− − −	− −	+/−

Scores range from − − − to +++

Table 2
Usability of investigated ADMs on ambiguous grammars of use cases

Another precision between SLR(1) and LR(1) would make the tested implementation more scalable.

The LR(k) test was the least usable of the three. It is actually only helpful for grammars that are LR(k) the first time they are tested. In all other cases it will never terminate and its intermediate reports are hard to trace to (possible) sources of ambiguity. Also, in general the LR(k) precision of the NU test is guaranteed to be stronger than the LR(k) test, for every value of k [12]. The LR(1) precision of the NU test did indeed find no ambiguities in grammars that MSTA identified as LR(1).

7 Evaluation

As a consequence of the different input formats of the investigated implementations we used only grammars in BNF notation. All three ADMs support the use of priority and associativity declarations, but it is wrong to assume they all adopt the same semantics [2]. It was hard finding grammars for generalized parsers that do not use any form of disambiguation, so we gathered only YACC grammars. To create unambiguous base lines for the medium and large grammars we removed all their conflicts, effectively making them LALR(1) and thus also LR(1). The ambiguous versions we created are of course not LR(1), but might still be close. This should not be a problem for the analysis of AMBER and the LR(k) test (since we focus on ambiguous grammars), but it might result in the NU test reporting less conflicts. However, it will not influence its accuracy measurements because the NU test gives conservative answers.

We have investigated implementations of three ADMs, but it would also be interesting to compare them to other existing methods. Unfortunately, the investigated implementations were the only ones readily available at the start of this project, with the exception of the derivation generator of Jampana [8].

However, we choose not to include it because it closely resembles AMBER, and it is very likely to produce incorrect results. It only generates derivations in which a production rule is never used more than once, and assumes all ambiguities of a grammar will show up.

A method that is particulary interesting is the recent Grambiguity ADM by Brabrand et al. [3]. After our own project had ended, Schmitz compared the accuracy of his ADM to that of the Grambiguity ADM on an extension of our collection of small unambiguous grammars [11]. His LR(0), SLR(1) and LR(1) precisions achieved accuracies of 65%, 75% and 87%, compared to 69% of the Grambiguity ADM. However, he did not apply any grammar unfolding, which might improve this latter score. 69% of the grammars were LR(k).

8 Conclusions

In this paper we have evaluated the practical usability of three ambiguity detection methods on a set of use cases. We have measured their accuracy, termination and performance, and analyzed their scalability and the usefulness of their reports. AMBER was very useful in three out of the four use cases, despite its exponential performance. The tested Noncanonical Unambiguity implementation was also quite useful on the medium sized grammars. It still has room for improvement but looks very promising. The LR(k) test was the least usable of the three. It appeared only useful on grammars that are actually LR(k).

8.1 Discussion

The practical usability of an ambiguity detection method depends largely on the grammar being tested. It is important to keep in mind that our measurements are only empirical and that the results cannot be easily extended to other grammars of the same sizes. However, they do give an insight into the practical implications of the differences between the tested methods, opening up new ways for improvements or optimizations. For instance heuristics that help to choose between AMBER's default or ellipsis mode, by calculating the minimum string lengths of the derivations of nonterminals.

Our results could also lead to new ADMs that combine existing methods, adopting their best characteristics. For instance, the NU test of Schmitz is very fast and pretty accurate, but it is not yet able to pinpoint exact sources of ambiguity in a grammar. On the other hand, derivation generators like AMBER are exact, but they have the problem of possible nontermination. A more elegant solution would be an iterative approach that gradually narrows locations of ambiguity in a grammar, testing in more detail with each step.

Acknowledgement

This project was conducted at the Centrum voor Wiskunde en Informatica, as part of the one year Master Software Engineering of the Universiteit van Amsterdam. The author thanks Paul Klint, Jurgen Vinju, Rob Economopoulos and Sylvain Schmitz for their valuable input during this research.

References

[1] H. J. S. Basten. Ambiguity detection methods for context-free grammars. Master's thesis, Universiteit van Amsterdam, August 2007. See http://homepages.cwi.nl/~basten/publications/BastenMaster.pdf.

[2] E. Bouwers, M. Bravenboer, and E. Visser. Grammar engineering support for precedence rule recovery and compatibility checking. In A. Sloane and A. Johnstone, editors, *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA'07)*, Braga, Portugal, March 2007.

[3] C. Brabrand, R. Giegerich, and A. Møller. Analyzing ambiguity of context-free grammars. In Miroslav Balík and Jan Holub, editors, *Proc. 12th International Conference on Implementation and Application of Automata, CIAA '07*, July 2007.

[4] D. G. Cantor. On the ambiguity problem of backus systems. *Journal of the ACM*, 9(4):477–479, 1962.

[5] N. Chomsky and M.P. Schützenberger. The algebraic theory of context-free languages. In P. Braffort, editor, *Computer Programming and Formal Systems*, pages 118–161. North-Holland, Amsterdam, 1963.

[6] R. W. Floyd. On ambiguity in phrase structure languages. *Communications of the ACM*, 5(10):526–534, 1962.

[7] D. Grune and C. J. H. Jacobs. *Parsing techniques a practical guide.* Ellis Horwood Limited, Chichester, England, 1990.

[8] S. Jampana. Exploring the problem of ambiguity in context-free grammars. Master's thesis, Oklahoma State University, July 2005.

[9] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[10] V. Makarov. *MSTA (syntax description translator)*, May 1995. See http://cocom.sourceforge.net/msta.html.

[11] S. Schmitz. *Approximating Context-Free Grammars for Parsing and Verification.* PhD thesis, Université de Nice - Sophia Antipolis, 2007.

[12] S. Schmitz. Conservative ambiguity detection in context-free grammars. In Lars Arge, Christian Cachin, Tomasz Jurdziński, and Andrzej Tarlecki,

editors, *ICALP'07: 34th International Colloquium on Automata, Languages and Programming*, volume 4596 of *Lecture Notes in Computer Science*, pages 692–703. Springer, 2007.

[13] S. Schmitz. An experimental ambiguity detection tool. In A. Sloane and A. Johnstone, editors, *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA'07)*, Braga, Portugal, March 2007.

[14] F. W. Schröer. AMBER, an ambiguity checker for context-free grammars. Technical report, Fraunhofer Institute for Computer Architecture and Software Technology, 2001. See http://accent.compilertools.net/Amber.html.