Ambiguity Detection Methods for Context-Free Grammars

H.J.S. Basten

Master's thesis

August 17, 2007

One Year Master Software Engineering
Universiteit van Amsterdam
Thesis and Internship Supervisor: Prof. Dr. P. Klint
Institute: Centrum voor Wiskunde en Informatica

Contents

Abstract

The Meta-Environment enables the creation of grammars using the SDF formalism. From these grammars an SGLR parser can be generated. One of the advantages of these parsers is that they can handle the entire class of context-free grammars (CFGs). The grammar developer does not have to squeeze his grammar into a specific subclass of CFGs that is deterministically parsable. Instead, he can now design his grammar to best describe the structure of his language. The downside of allowing the entire class of CFGs is the danger of ambiguities. An ambiguous grammar prevents some sentences from having a unique meaning, depending on the semantics of the used language. It is best to remove all ambiguities from a grammar before it is used.

Unfortunately, the detection of ambiguities in a grammar is an undecidable problem. For a recursive grammar the number of possibilities that have to be checked might be infinite. Various ambiguity detection methods (ADMs) exist, but none can always correctly identify the (un)ambiguity of a grammar. They all try to attack the problem from different angles, which results in different characteristics like termination, accuracy and performance. The goal of this project was to find out which method has the best practical usability.

In particular, we investigated their usability in common use cases of the Meta-Environment, which we try to represent with a collection of about 120 grammars with different numbers of ambiguity. We distinguish three categories: small (less than 17 production rules), medium (below 200 production rules) and large (between 200 and 500 production rules). On these grammars we have benchmarked three implementations of ADMs: AMBER (a derivation generator), MSTA (a parse table generator used as the LR(k) test) and a modified Bison tool which implements the ADM of Schmitz. We have measured their accuracy, performance and termination on the grammar collections. From the results we analyzed their scalability (the scale with which accuracy can be traded for performance) and their practical usability.

The conclusion of this project is that AMBER was the most practically usable on our grammars. If it terminates, which it did on most of our grammars, then all its other characteristics are very helpful. The LR(1) precision of Schmitz was also pretty useable on the medium grammars, but needed too much memory on the large ones. Its downside is that its reports are hard to comprehend and verify.

The insights gained during this project have led to the development of a new hybrid ADM. It uses Schmitz' method to filter out parts of a grammar that are guaranteed to be unambiguous. The remainder of the grammar is then tested with a derivation generator, which might find ambiguities in less time. We have built a small prototype which was indeed faster than AMBER on the tested grammars, making it the most usable ADM of all.

Preface

This thesis is the result of the research project I did for the one year Master's education in Software Engineering, at the Centrum voor Wiskunde en Informatica in Amsterdam.

I would like to thank Paul Klint for offering me a chance to conduct this project at the CWI, and for his valuable feedback and supervision.

My work would also have been a lot harder without the help of Jurgen Vinju and Rob Economopoulos. I'm very grateful to them for their helpful tips and answers to my questions, and for always having an open door policy.

Also, I thank Jan Heering and Ron Valkering for reviewing the previous versions of this thesis.

Finally I would like to thank Sylvain Schmitz for our insightful e-mail conversations.

Bas Basten
Amsterdam, August 2007

Chapter 1

Introduction

1.1 Motivation

The use of context-free grammars (CFGs) for syntax definition is already widespread and keeps growing. Partly due to the increasing popularity of Domain Specific Languages and the reverse engineering of source code. Also (S)GLR parsers [28] are being used more often, because they are able to handle all classes of CFGs. One of the advantages of this is that the grammar developer can design his grammar to best describe the structure of the intended language. He does not have to squeeze his grammar into LALR(1) form to have it accepted by Yacc for instance. The downside of using the entire class of CFGs is the possibility of ambiguities. An ambiguity means that a string which is in the language of a CFG, can be derived from the grammar in more than one way. This usually means that the string has multiple meanings, depending on the semantics of the used language. Using an ambiguous grammar makes it impossible to properly specify certain types of data.
A better solution would be to use an unambiguous grammar in the first place. There can be no confusion about the meaning of a particular string. Neither for the computer nor for the reader/author. However the problem of determining whether a given grammar is ambiguous or not, is undecidable [2].

The development of a grammar usually is a cumbersome process for which there is no standardized approach. It is an ad hoc process of trial and error, sometimes called 'grammarware hacking' [18]. The danger of introducing ambiguities complicates it even more, because the ambiguity of a grammar is usually hard to see. The different derivations of an ambiguous string quickly become too complex to imagine by just reading the individual production rules. This makes it hard to predict the consequences of a modification of the grammar. There are some heuristics however which can be followed to guarantee non unambiguity, but these might result in unwanted string patterns (for instance abundant bracket pairs). And even then there is always the possibility of making mistakes.
To make sure a grammar is unambiguous it needs to be verified afterwards. However searching for ambiguities in a grammar by hand is very hard. It involves constructing and checking complex combinations of production rules. Also the number of cases which have to be checked might be infinite.
An automated ambiguity detection method could be of great help. It relieves the grammar

developer of this complex, repetitive and time consuming task. Of course there is the possibility that it will run forever too, but it might be able to find more ambiguities in a reasonable amount of time.

Currently, various ambiguity detection methods (ADM) for CFGs exist in literature. They all have different characteristics, because they approach the ambiguity problem in different ways. For instance, some are very accurate but time-consuming, while others are faster but not always correct. All these characteristics influence the practical usability of an ADM. The goal of this project is to explore the various existing methods and to investigate their practical usability. The motivation behind this is to find a suitable ADM for use in the Meta-Environment. The Meta Environment is a framework for language development, source code analysis and source code transformation. It is developed at the Centrum voor Wiskunde en Informatica (CWI) in Amsterdam. The Meta-Environment allows the specification of context-free grammars using SDF notation (Syntax Definition Formalism). With these grammars a development environment can be created in which source code can be written, parsed, compiled, analyzed, transformed, etc. A useful addition to the Meta-Environment would be an ambiguity detection tool that could aid the grammar developer in writing unambiguous grammars.

1.2 Scope

Trying to determine the practical usability of an ADM in general will be very hard. It depends on a lot of different factors. The size of the grammar, the parameters of the method, the computation power of the used PC, the experience of the grammar developer, etc. Carefully considering all these factors will take a lot of time. And still, because the ambiguity problem is undecidable there will always be grammars on which a certain method will fail. Because of the limited time of this research project, we will look at the behavior of the investigated ADMs in common use cases of the Meta-Environment.

The Meta-Environment is mainly used for the reverse engineering of existing source code, and the creation of Domain Specific Languages. For the DSLs a development environment can be generated from a grammar. These grammars are usually developed from scratch, and have an average size of 50-200 production rules.
For reverse engineering, grammars are needed to parse source code of existing industry standard programming languages, like COBOL, Fortran, C, Java, etc. These grammars usually have a maximum size of around 500 production rules. Most of the languages are documented in standards. However, these can almost never be directly converted into a usable grammar for parser generation. The ambiguity of the developed grammar still has to be checked by hand.
An advantage of developing a grammar for an already existing language, is that actual source code also exists. This code has to be accepted by the grammar (without ambiguities), and can be used as a test set. However, this does not rule out the need of an ambiguity detection method. The code of the test set has a unique meaning, but sometimes this cannot be recognized by a CFG parser alone. For a lot of programming languages a type checker is needed to disambiguate the parse trees of a piece of source code. In such a case, the grammar will have to be intentionally ambiguous at some points. An ambiguity detection method that identifies

the individual cases of ambiguity in a grammar, might then be a better option than writing a complete development toolkit.

We can imagine an ADM being used in multiple ways during the development of a grammar. The first is a quick check after some modifications of the grammar, like the way a compiler can be used to check the type correctness of a piece of code during programming. The second is a more extensive check which is run overnight, by means of a regression test. Finally it could also be used as an acceptance test before the grammar will be taken into use. The quick check would ideally take at most a few minutes, so grammar programming will not be interrupted too much. An overnight check should take at most 15 hours. An acceptable duration for an acceptance test might be a couple of days, because it is not run very often.

The Meta-Environment is used to develop grammars of different sizes. We distinguish two categories: 1) *medium grammars* with a size of below 200 production rules, which are used in DSL development or reverse engineering, and 2) *large grammars* with a size between 200 and 500 production rules, mainly used for reverse engineering. For both these categories we will investigate if the current ADMs are practically usable. We will focus on the use of an ADM as a quick check and a more extensive check overnight. We will not look at the use of an ADM as a very long acceptance test, because its time constraints are less strict.

This results in the following 4 use cases. For each of these we will test if the methods are practically usable.

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.

1.3 Criteria for practical usability

The currently existing ADMs 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 against. In this project we will use the following ones:

Termination Because the ambiguity detection problem is undecidable, the search space of some methods can be infinite. On certain grammars they might run forever, but could also take a very long time to terminate. 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 (un)ambiguity of the tested grammar remains uncertain. To be practically usable, an ADM has to terminate in a reasonable amount of time.

Accuracy After a method has terminated it reports if it finds the inspected grammar ambiguous or not. Some existing methods do not always produce the right answer. Every report of these methods will have to be verified by the user. If this is a complex task, then the verification of a lot of potential false reports might take a long time. The amount of correct reports thus has a great impact on the usability of a method.

The accuracy of a method can be determined by the percentage of correct reports. A method that is guaranteed to correctly identify the (un)ambiguity of an arbitrary grammar, is 100% accurate. Because the ambiguity problem is undecidable, 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.

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. 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 criteria is closely related to termination, because it affects the amount of time one should wait before halting a method.

Scalability In the use cases above we saw that an ADM might be used as a quick check or an extensive check. 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.

Usefulness of the output 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.

1.4 Existing ambiguity detection methods

The following ambiguity detection methods currently exist. They will be described in chapter 3.

- Gorn [13], 1963
- Knuth's LR(k) test [20], 1965
- Cheung and Uzgalis [7], 1995
- AMBER [25], 2001

- Jampana [17], 2005
- Brabrand, Giegerich and Møller [6], 2006
- Schmitz [23], 2006

Some of these methods are very similar to each other. For others there were no tools available that implement them. Because of this we will only benchmark three tools. An implementation of the LR(k) test, AMBER and Schmitz. A more extensive motivation for this is given in section 4.3.

1.5 Research questions

The goal of this project is to compare the practical usability of various existing ambiguity detection methods. The main research question is:

What ambiguity detection method has the best practical usability?

In this project we will define the practical usability of an ADM as follows. To be practically usable for a grammar, a method should:

- terminate in an acceptable amount of time
- correctly report the (un)ambiguity of the grammar
- report useful information for the disambiguation of the grammar

To assess the practical usability of the investigated ADMs we will try to determine the extend in which they meet these requirements. To evaluate the termination, accuracy and performance of the ADMs, we will set up a collection of grammars. On these grammars we will benchmark implementations of the methods, and measure these three criteria. The evaluation of the usefulness of the output of the ADMs will be a more theoretical survey. We will then analyze which of the investigated ADMs are practically usable in the use cases stated above.

This results in the following subquestions for this project:

1. What is the accuracy of each method on the sample grammars?
2. What is the performance of each method on the sample grammars?
3. What is the termination of each method on the sample grammars?
4. How useful is the output of each method?
5. Which methods are practically usable in the stated use cases?

It is important to keep in mind that the collection of grammars can only be used as a sample survey. Our research is thus only empirical. As mentioned before, the behavior of a method depends on a lot of different factors, which we cannot all take into account. That is why we try to focus on average use cases of the Meta-Environment, which we try to represent with our collection of grammars.

It is unlikely that our findings will be valid in all other cases. Still, we might be able to come up with some valuable findings. The outcomes of our tests can prove that certain situations are able to occur. They can be used to verify if a method behaves as expected. Maybe also unexpected outcomes show up, which can give more insight into a method's algorithm. For instance, it might be interesting to consider the possibility of combining two methods, if they turn out to perform well in different situations.

1.6 Thesis overview

The remainder of this thesis is set up as follows:
Chapter 2 contains background information about languages and context-free grammars. It discusses the problem of ambiguity in context-free grammars, and the implications for trying to detect it.
Chapter 3 gives an overview of the currently available ambiguity detection methods.
Chapter 4 describes the research method used to measure the above criteria for the investigated ADMs. It also motivates the choice of the tools that implement the investigated methods.
The results of the measurements are presented in appendices A, B and C. They are analyzed in chapters 5, 6 and 7. These chapters form the answers to the research questions.
In chapter 8 we present an idea for a new hybrid ADM, which we developed during this project. It is not part of the main research project.
Chapter 9 summarizes measurement results and compares the investigated ADMs to each other. It answers the main research question and contains the final conclusion. It also sketches some ideas for future work.

Chapter 2

Background and context

2.1 Grammars and languages

This section gives a quick overview of the theory of grammars and languages. At the same time we introduce the notational convention of [2] that is used throughout this thesis. For a more detailed introduction we refer to [1], [2] and [14].

A context-free grammar G is a 4-tuple (N, Σ, P, S) consisting of:

- N, a finite set of *nonterminals*
- Σ, a finite set of *terminals* (the alphabet)
- P, a finite subset of N $\times$ (N $\cup$ Σ)*, called the *production rules*
- S, the *start symbol*, an element from N

Usually only the production rules of a grammar are given. The sets of terminals and nonterminals can be derived from them, because of their different notations. The nonterminal on the right-hand side of the first production rule is the start symbol.
The following characters are used to represent different symbols and strings:

- a, b, ... t and 0, 1, ... 9 represent terminals.
- A, B, C, ... T represent nonterminals, S often is the start symbol.
- U, V, ... Z represent either nonterminals or terminals.
- α, β, ... represent strings of nonterminals and terminals.
- u, v, ..., z represent strings of terminals only.
- ε represents the empty string.

A production (A, α) of P is usually written as $A \to \alpha$. Another arrow, $\Rightarrow$, is used to denote the derivation of *sentential forms*. A sentential form of a grammar is a string in (N $\cup$ Σ)*, that can be derived from the start symbol in zero or more steps. If $\alpha\beta\gamma$ is a sentential form of G, and $\beta \to \delta$ is a production in P, we can write $\alpha\beta\gamma \Rightarrow \alpha\delta\gamma$ (read $\alpha\beta\gamma$ directly derives

$\alpha\delta\gamma$). The first sentential form of a grammar is the start symbol.
The derivation of a sentential form can be visualized with a *parse tree*. This is a graph which shows the hierarchy of the used productions. Figure 2.1 shows two parse trees.
A sentential form consisting only of terminals is called a *sentence*. The set of all sentences that can be derived from the start symbol of a grammar G is called the *language* of that grammar, denoted $L(G)$.
A grammar is being *augmented* by adding an extra production rule $S' \rightarrow S\$$. When a parser reduces with this production it knows it can accept the input string. The symbol \$ marks the end of the string.

2.2 Ambiguity in context-free grammars

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 2.1 shows two derivation trees of an ambiguous sentence of the following grammar:

$$E \rightarrow E + E \mid E * E \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9.$$

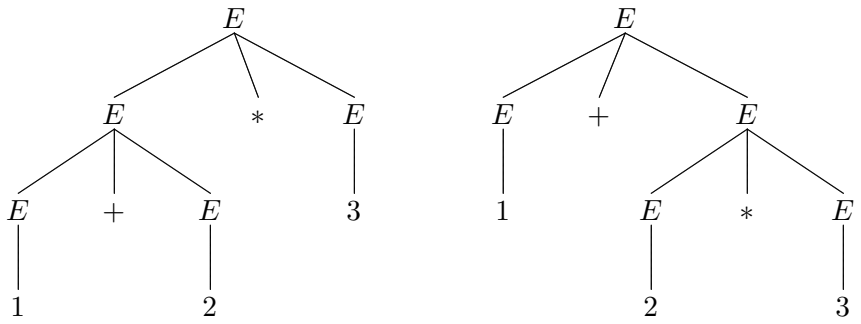

Figure 2.1: Example of two parse trees of the sentence '$1 + 2 * 3$'

In this thesis methods that detect ambiguities in grammars are investigated. But what exactly is *one* ambiguity? An ambiguous sentence is called an ambiguity. However, this might not be a useful definition while talking about ADMs. A recursive grammar might produce an infinite number of ambiguous strings. Should an ADM produce them all?
The word 'ambiguity' is also often used with a different meaning. For instance when talking about the famous 'dangling else' ambiguity. In this case it is used to refer to all ambiguous strings that contain a dangling (else-like) tail. It is like some sort of pattern is meant which covers an infinite number of ambiguous strings. Talking about the ambiguities of a grammar in this way is much easier than treating every ambiguous sentence individually.

In the ACCENT manual [26], Schröer presents a way to group the ambiguities of a grammar into classes. He defines two classes of ambiguous strings: *disjunctive* ambiguities and *conjunctive* ambiguities. A disjunctive ambiguity is a string that has multiple derivations from a particular nonterminal, each starting with different production rules. In other words, two different production rules of a nonterminal can produce the same sentence. For instance:

$$S \rightarrow A \mid B, A \rightarrow a, B \rightarrow a.$$

In this example the string 'a' can be derived from S, starting with either $S \rightarrow A$ or $S \rightarrow B$. A conjunctive ambiguity is a string that can be split up in different ways, such that its parts match the symbols of a single production rule. For instance:

$$S \rightarrow AB, A \rightarrow a \mid aa, B \rightarrow a \mid aa.$$

In this example the string 'aaa' matches $S \rightarrow AB$ in two ways. It can be split up into 'a' and 'aa' to match A and B respectively, but 'aa' and 'a' also match A and B.
In [6] Brabrand, Giegerich and Møller give the same classification, but with different names. A disjunctive ambiguity is called a *vertical* ambiguity, because two different production rules of the same nonterminal are involved, and these are usually written below each other. A conjunctive ambiguity is called a *horizontal* ambiguity, because it involves only a single production, which is usually written on one line.

Specifying ambiguities with the definitions of Schröer or Brabrand *et al.* makes the number of ambiguities in a grammar finite. An ambiguity can be either **a**) a pair of production rules of the same nonterminal that can both produce the same string (disjunctive ambiguity), or **b**) a single production rule that can produce a string using different derivations of its right-hand side (conjunctive ambiguity). Since the number of production rules of a grammar is always finite, the maximum number of ambiguities in a grammar is also finite.

However, this classification might be too coarse-grained. Suppose we added the following production rules to the conjunctively ambiguous grammar mentioned above:

$$A \rightarrow b \mid ba.$$

The string 'baa' is now also conjunctively ambiguous. But according to the definition, the number of ambiguities in the grammar stays the same. The production $S \rightarrow AB$ still forms the only conjunctive ambiguity, although the derivation of the new string continues from it with different production rules. Does this addition of production rules introduce a new ambiguity, or has it only extended the set of ambiguous strings from the already existing ambiguity?

We could also look at this definition problem from the grammar developer's perspective. He might see an ambiguity as an unwanted phenomenon in a grammar that could be removed by applying a disambiguation construct. In this sense, the number of ambiguities in a grammar is the number of disambiguation rules that have to be applied to completely disambiguate it. ACCENT provides priority declarations to solve disjunctive ambiguities, and a longest or shortest match annotation to solve conjunctive ambiguities. Unfortunately, the disambiguation rules of SDF [28] are not capable of solving every ambiguity. Sometimes the production rules will also have to be altered. This definition too is thus not watertight, because it depends on the used disambiguation constructs.

So what definition of an ambiguity should we use when counting the number of ambiguities in a grammar? At this time we have to conclude there is no generally accepted definition. The different ADMs use different definitions.

2.3 Ambiguity detection for context-free grammars

Determining whether a particular CFG is (un)ambiguous is an undecidable problem [1]. Because of this an ambiguity detection method (ADM) that is guaranteed to identify the (un)ambiguity of a grammar cannot exist. Different approaches are possible to attack the problem, but none can produce the right answer in all cases. It is also not possible to find all individual ambiguities in each grammar. One approach would be to simply start searching the endless number of possible sentences, but for a grammar with an infinite language this would never end. When the searching is broken off after running a certain amount of time, there might still be some undiscovered ambiguities left in the grammar.

To gain faster results heuristics can be used which trade accuracy for performance. They could produce more results in less time, but might also report false positives and/or false negatives. False positives are reports of ambiguities which are not really ambiguities. A false negative is the failure to report an ambiguity that does exist. The problem with these false positives and negatives is that they have to be verified before it is certain they are real ambiguities or not. In some cases this is an undecidable problem on its own, because otherwise the ambiguity problem would be decidable.

Suppose an ambiguity detection algorithm exists that always terminates in finite time. Its reports are not always correct, so it might give false positives or false negatives. Also suppose that all of its reports could be correctly verified to be either true or false in finite time. This algorithm (together with the verification algorithm) would then be able to report all ambiguities in an arbitrary grammar, in finite time. However the ambiguity problem is undecidable so such an algorithm can never exist.

If a method might output false reports then all reports will have to be verified to filter out the real ambiguities. Preferably this is done automatically. The amount of reports that are left for verification by the user greatly influences the usability of a method. One can imagine that verifying a lot of potential false reports can be a complex and time consuming task. An algorithm that only reports potentially false positives is easier to verify than an algorithm with only false negatives. A false negative is the failure to report an ambiguity, so it is the absence of a report. Verifying false negatives thus involves verifying all possible absent reports.

Chapter 3

Current ambiguity detection methods

This chapter will describe the currently existing ambiguity detection methods. They are ordered as follows: The first four are (variations of) derivation generators. The fifth is a test for inclusion in an unambiguous grammar class. The last two are both methods that test approximations of the grammar to be able to terminate in finite time.

3.1 Gorn

Description In [13] Gorn 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.
The 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 nonterminals with their productions. Strings containing only terminals (the sentences) are compared to previously generated strings. Strings containing nonterminals 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.

Correctness If the method finds an ambiguous string then this report is 100% correct. The two (or more) derivations can prove this. 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 nontermination. Running the method forever is impossible so it will always have to be halted at some point. 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.

3.2 Cheung and Uzgalis

Description The method of Cheung and Uzgalis [7] 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 postfixes 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.
The following example is an unambiguous grammar with an infinite number of derivations, for which this methods terminates and correctly reports non-ambiguity:

$$S \to A \mid B, \quad A \to aA \mid a, \quad B \to bB \mid b.$$

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 [8] the author proves the pruning is always done in a safe way, so no ambiguities are overlooked.

3.3 AMBER

Description AMBER is an ambiguity detection tool developed by Schröer [25]. It uses an Earley parser [11] 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. 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,

in stead 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. 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.

Scalability 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 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.

3.4 Jampana

Description Jampana also investigated ambiguity detection methods for his Master's project. In his thesis [17] he presents his own ambiguity detection method. It 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.

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 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. 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:

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

which looks like this when converted to CNF:

$$E \to EO \mid a, \quad O \to PE, \quad P \to +$$

The live productions here are $E \to EO$ and $O \to PE$. The strings that do not use a live production more than once are: 'a' (Figure 3.1.a) and '$a+a$' (Figure 3.1.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 3.1.c and 3.1.d.

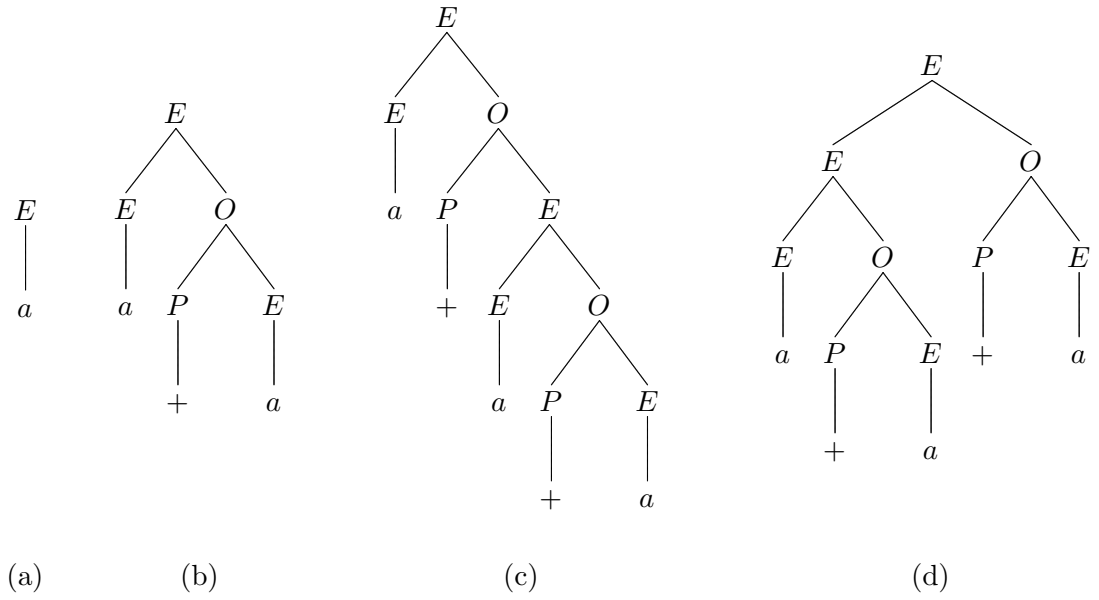

Figure 3.1: Examples to clarify Jampana's ADM

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

3.5 LR(k) test

Description Knuth is the author of the LR(k) parsing algorithm, which he describes in [20]. 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. 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.

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.

3.6 Brabrand, Giegerich and Møller

Description The method of Brabrand, Giegerich and Møller [6] 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 nonterminal are searched for vertical ambiguities. In their paper 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. The authors use an algorithm by Mohri and Nederhof [22] for demonstration. It 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. All nonterminals 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 scaled.

3.7 Schmitz

Description In [23] Schmitz 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. 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

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

looks like this 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 unbracketed 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 $a \cdot + a$. The edges of the graph are the valid transitions between the positions.

There are three types of transitions: shifts (of terminals or nonterminals from the original grammar), derivations and reductions. The shift edges are labeled with their respective terminals or nonterminals. The derivations are labeled with d_i symbols and the reductions with r_i symbols. Figure 3.2 shows an example of the first part of the derivation graph for the above grammar.

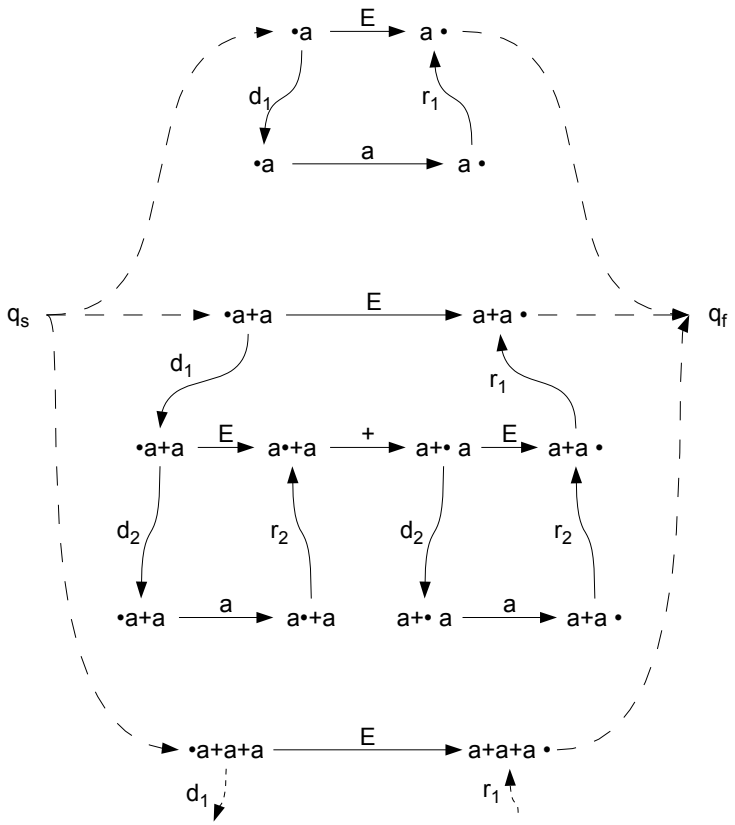

Figure 3.2: Example position graph

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.
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 unbracketed 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, in stead 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.

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.* 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 Now that 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 item_0 is reasonably accurate'. With the item_0 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.

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 nonterminal and the state on top of the stack. This is the state from where the derivation of the reduced nonterminal was started. The item corresponding to this state is of form $A \to \alpha \cdot B\beta$. After the reduction of the nonterminal B it is only allowed to go to the state that corresponds with $A \to \alpha B \cdot \beta$. 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 $C \to B\gamma$, then it is possible to go from position $A \to \alpha \cdot B\beta$ to $C \to B \cdot \gamma$ by shifting a B. This makes it possible to parse the first half of a string with one part of a production rule, derive and reduce a nonterminal, and then parse the remaining half with the end of another production rule.

Just like item_0 there is also the relation item_k, where k is a positive integer. 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.

3.8 Summary

The following table captures the important properties of the described methods:

Method	Guaranteed termination	Decides ambiguity	Decides non-ambiguity	False positives	False negatives	Scalable
1. Gorn	no	yes	no[1]	no	–	no
2. Cheung	no	yes	yes	no	no	no
3. AMBER	no	yes	no[1]	no	–	yes[2]
4. Jampana	yes	yes	yes	no	yes	no
5. LR(k) test	no	no	yes	–	no	no
6. Brabrand *et al.*	yes	yes	yes	yes	no	yes[3]
7. Schmitz	yes	yes	yes	yes	no	yes[4]

[1] except for non-recursive grammars
[2] nr. or % of derivations; checking of sentential forms holding nonterminals
[3] with approximation technique; level of grammar unfolding
[4] with equivalence relation

Table 3.1: Summary of existing ambiguity detection methods

Chapter 4

Research method

In section 1.3 we described the various criteria with which the ambiguity detection methods will be compared. In this chapter we will describe how we will measure these criteria and analyze the results. Also the choice for the investigated ADM implementations is explained.

4.1 Measurements

For our measurements we have set up 2 collections of grammars. The first contains 84 ambiguous and unambiguous grammars with a maximum size of 17 production rules. We call this the collection of *small* grammars. The second collection contains grammars of real-life programming languages, which we call the *medium* and *large* grammars. It contains 5 unambiguous grammars for HTML, SQL, Pascal, C and Java. Their respective sizes are 29, 79, 176, 212 and 349 production rules (in BNF notation). For 4 of them we have also created 5 ambiguous versions. According to section 1.2, the HTML, SQL and Pascal grammars are medium grammars, and those of C and Java are large grammars. Both collections are included in appendices D.1 and D.2.

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 termination and performance. The collection of small grammars is set up to measure the accuracy of an ADM separately of its termination and performance.
The termination and performance of an ADM are more relevant in relation to real-life grammars, so we will measure them on the collection of medium and large grammars.
The next sections describe in more detail how each criteria will be measured.

4.1.1 Accuracy

Definition The accuracy of a method is the percentage of correct reports. It is calculated by dividing the number of correct reports by the total number of reports. It can only be calculated for cases in which a method terminates. Here the word *report* can have multiple meanings. It could mean a report about the entire grammar being ambiguous or unambiguous,

or a report that pinpoints the individual ambiguities in a grammar. We would like to measure the accuracy of an ADM on both of these levels.

Measurement We will try to measure the accuracy of an ADM independent of its termination and performance. We will do this on the collection of small grammars, to minimize the computation time and memory usage. The small grammars are collected from literature and known ambiguities in 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 a lot of common grammar 'mistakes' are included.
We will run the tools on these grammars and verify if they correctly report (un)ambiguity on the grammar level. Not all ADMs are able to report both ambiguity and unambiguity. To make a better comparison, we will calculate the accuracy separately for ambiguous and unambiguous grammars.

Unfortunately, measuring the amount of correct reports on the lower level might be too complicated for this project. As described in section 2.2 the meaning of the word ambiguity is not always clear. The investigated ADMs all have different ways of reporting an ambiguity, because they approach the ambiguity problem from different angles. This definition problem brings two difficulties: **1**) how to specify the ambiguities in a grammar, and **2**) how to relate an ambiguity report of an ADM to an actual ambiguity?
Because we have no common 'ambiguity unit', it is not possible to count the exact number of ambiguities in our grammars. However, to only get an idea of the degree of ambiguity of our grammars, we could choose a practicable definition and count the number of ambiguities with it. We have chosen to count the number of disjunctive and conjunctive ambiguities of all grammars. A pair of production rules of the same nonterminal that can produce the same string, is counted as one disjunctive ambiguity. A production rule which parts can match different parts of an ambiguous string is counted as a conjunctive ambiguity.
This also enables us to verify the results of AMBER, solving part of the second difficulty. However we will not try to verify the ambiguities as reported by MSTA and Schmitz. Except in the special situation in which a method reports ambiguities for an unambiguous grammar. Then we know these are all false.

The number of disjunctive and conjunctive ambiguities of a grammar could be used as an indication of the degree of ambiguity of the grammars. We could compare it to the number of ambiguity reports that Schmitz' method gives on a grammar. This way we might be able to tell something about the usefulness of the reports. For instance, if a grammar contains 6 ambiguities and an ADM gives only one report, this report is not likely to indicate all causes of ambiguity with much detail. On the other hand, if an ADM gives a lot more reports than the number of ambiguities in the grammar, this too might not be very helpful. Every report will have to be verified by the user. It makes no difference if the report is a false positive or a duplicate report of an already found ambiguity.

Verification Practically it is not possible to know of all our grammars if they are ambiguous or not. However most of these grammars are designed to demonstrate special properties of grammars or ambiguity detection methods. Others contain known ambiguities of existing programming languages. The (un)ambiguity of the used grammars has already been proven

in most cases. Still, if our initial conception of a grammar would be wrong we hope one of the (accurate) methods will show this.

4.1.2 Performance

Definition We will express the performance of an ADM on a certain grammar in two ways: **1**) the time it takes to complete its calculation on a certain PC, and **2**) the maximum amount of virtual memory it needs.

Measurement Our goal is to test if a method is suitable for use with an average sized grammar on an average PC. As the average PC we will use a workstation at the CWI: an AMD Athlon 64 3500 with 1024 MB of RAM (400DDR) running Fedora Core 6. As average grammars we will use the collection of medium and large grammars.

With the `time` utility, we measure the elapsed real time an ADM tool uses to check a grammar. The amount of virtual memory of the tools process is measured every second with the `ps` utility, executed from a bash script.

Verification To minimize the influence on the running time of a tool, the memory usage is not measured continuously. The maximum amount of memory we will find might thus not be 100% accurate. Also if a program run ends in less than a second, we cannot measure its memory usage. However it is unlikely that the program will allocate gigabytes of memory in this short time.

4.1.3 Termination

Definition Not every ADM is guaranteed to terminate on all grammars. On some grammars they do terminate, but they might take a very long time. Practically they have to be halted after some time. We will measure if the investigated ADMs successfully terminate within the time constraints of the use cases (section 1.2).

Measurement This criteria will be analyzed from the performance measurements. All measurements will be limited to 15 hours, the time allowed for an extensive overnight regression test. For each medium and large grammar, we will test if the ADM tool successfully terminates within 5 minutes or within 15 hours.

4.1.4 Usefulness of the output

Definition The usefulness of a report that identifies a specific ambiguity in a grammar, is not as formally expressible as the accuracy or performance of a method. Whether a message is useful for the reader depends for a great deal on the knowledge and intelligence of that reader. However there are also certain criteria which make it easier for every user to understand the reported ambiguity.

Horning [15] describes seven characteristics of a good error message from a parser. Yang, Michaelson, Trinder and Wells [29] also describe seven properties of good error messages from a compiler. Several of them overlap with those of Horning.
Most of these characteristics are also applicable to ambiguity reports. However some are more dependent on the way an ADM is implemented, and are not direct properties of the method itself ('Good error messages will ... be restrained and polite' [15]). From the characteristics of Horning and Yang *et al.* we have distilled five characteristics that the output of an ADM should have to enable its implementation to produce useful ambiguity reports:

- **Correct**: the reported ambiguity should really exist in the examined grammar.
- **Grammar-oriented**: it should report an ambiguity in terms of the grammar, not in terms of the used algorithm.
- **Localizing**: a report should pinpoint the exact locations in the grammar that cause the ambiguity.
- **Succinct**: a report should maximize useful information and minimize non-useful information.

Some of these characteristics depend on other characteristics. If a report is not correct or grammar-oriented, it can never be localizing or succinct.

Unfortunately we have no definition of what an ambiguity exactly is (section 2.2). This makes it also difficult to determine how well the output of a method meets the above characteristics. However, something that will definitely clarify an ambiguity report, is the presentation of an ambiguous example string. Preferably together with the different derivation trees and production rules involved. If there are various different example strings for the same 'ambiguity' then the shortest one is preferred. Also, the most specific report would state the lowest nonterminal in the dependency graph for which the string is ambiguous. If a string is ambiguous for a certain nonterminal, then it is also ambiguous (with some context) for the nonterminals higher up in the dependency graph. Reporting the lowest nonterminal also results in derivation trees with the smallest size. For an example see figure 4.1.

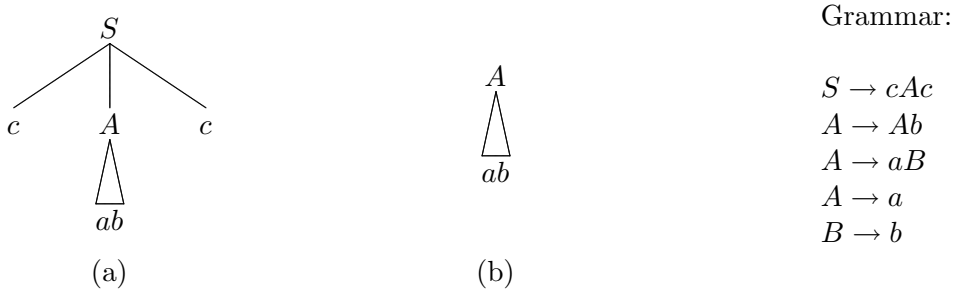

Figure 4.1: Two example derivation of an ambiguous substring. If 'ab' is ambiguous for A, then '$cabc$' is also ambiguous for S. The smallest possible derivation tree is (b).

Measurement As mentioned in section 4.1.1 we cannot measure the correctness of the reports. However, we are able to assess the other characteristics. Because most of them are dependent on the correctness of the reports, we assume that they are indeed correct.

We also look if the given reports contain ambiguous example strings, derivations or involved production rules, and if these are as short as possible.

Verification Section 2.2 explains we have no exact definition of an individual ambiguity. This complicates determining the degree in which an ambiguity report is localizingor succinct. Our measurements will have a certain degree of subjectivity.

4.2 Analysis

4.2.1 Scalability

The scalability of a method is the scale with which the level of accuracy and performance can be interchanged. For the scalable methods we will compare the accuracy, performance and termination of their various precisions, on the collection of medium and large grammars (appendix D.2). We will then try to assess the scale with which their performance can be traded for accuracy.

4.2.2 Practical usability

From results of the measurements stated above, we will analyze whether the investigated ADMs are practically usable in the use cases stated in section 1.2. For each ADM we will try to determine the critical properties that will be most decisive for this. We then try to assess if they are high enough in contrast to the constraints of each use case.

We will assume that there is a high chance that a tested grammar will be ambiguous the first time it is tested. Especially in the Meta-Environment, where the grammar developer does not have to mould his grammar into a class that is deterministically parseable. Therefore we will mainly focus on how usable the ADMs are for the ambiguous versions of the medium and large grammars.

Verification Initially we wanted to take medium and large unambiguous grammars in SDF format and translate these to the various input formats of the investigated tools. The translation would have to include the conversion of the different disambiguation constructs SDF offers, to guarantee that the grammars would remain unambiguous. Unfortunately, this turned out to be not so straightforward. The priority declarations of SDF and Yacc have different semantical meanings [5], and the parse tree filters [19] of SDF are not supported at all by Yacc. We then chose to take grammars already in Yacc format, which is also easily translated to AMBER's input format. Some of them included a few shift/reduce or reduce/reduce conflicts, which we solved by removing certain productions. The modifications, together with the introduced ambiguities, are stated in appendix D.2.

If a grammar is converted by Yacc into a parse table with no conflicts, the grammar is LALR(1). This class of grammars is a subclass of LR(1). All our medium and large grammars are thus also LR(1), which makes our comparison less fair. The LR(k) test will certainly pass

on them the first time, and also Schmitz' LR(1) precision is proven to find no ambiguities in LR(1) grammars. However, the ambiguous versions are not LR(1) of course, but they are still close. Large parts of their production rules would still be deterministically parseable. SDF grammars are usually not designed with deterministic parsing in mind. They would probably come less close to being LR(1). Using them would make for a better comparison.

4.3 Investigated ADM implementations

In chapter 3 the currently existing ambiguity detection methods are described. From these seven methods we will only benchmark implementations of three of them. The main reason is that there were no more tools available at the time of this project. Implementing new tools would take too much time of this relatively short project.

Looking at the described methods, the first four are very alike. They are all derivation generators with different optimizations. From these we will only investigate AMBER. Its algorithm is very similar to Gorn's. The ADM of Jampana is again very similar to AMBER (having the same optimizations). He included the source code of his implementation in his thesis [17], but we choose not to use it. Because it only searches a very limited amount of derivations it is very likely to produce false negatives. To search these derivations AMBER can be used as well.

We benchmark two different precisions of AMBER. The first is its default method of searching and comparing. It generates all sentences (of only terminals) of a grammar up to a given length, and checks for duplicates. The second method also checks if the intermediate sentential forms (containing nonterminals) have been generated before. It is called the *ellipsis* mode. AMBER can also be parametrized to consider only a predefined percentage of derivations at each level, which it selects randomly. We will not test this mode. Because it gives random results it is hard to evaluate it.

For the LR(k) test we used MSTA [21], an LR(k) parser generator by Vladimir Makarov. There are a lot of free parser generators available in the Internet, but MSTA seemed to be the only true LR(k) one with adjustable k. At first it reported some ambiguous grammars as being LR(k). But after a series of bugreports and quick fixes by the author it produced no unexpected results anymore.
In [16], Hunt III, Szymanski and Ullman present an LR(k) test that is faster than constructing the complete parse tables. Unfortunately, there was no implementation available of it.

The two youngest ADMs (Schmitz and Brabrand) both use conservative approximations. They guarantee termination of the searching, and never return false negatives. It would be very interesting to compare them with each other. Unfortunately only Schmitz made his implementation publicly available.
He has extended GNU Bison [9] with an ambiguity checker that implements his algorithm. It can be used with three precisions: using LR(0), SLR(1) or LR(1) items. In this order their precision increases. A more detailed description of this tool is given in [24].

Both MSTA and Schmitz' modified Bison tool use the Yacc input format. AMBER uses the ACCENT [26] input format, which is very similar to Yacc (apart from the priority declarations, but we will not use these).

Chapter 5

Analysis of AMBER

Being a derivation generator, AMBER is only able to determine the ambiguity of a grammar. For a recursive unambiguous grammar it will run forever. It gives no false positives or negatives. AMBER is the only method for which we also verified its reports on the ambiguity level. We ran it in default and in ellipsis mode (in ellipsis mode the sentential forms containing nonterminals are also compared). The results of the measurements are included in appendix A.

5.1 Accuracy

Ambiguous small grammars - grammar level Tables A.1 and A.2 show the number of ambiguity reports AMBER gives for the ambiguous grammars, in default and ellipsis mode. The first thing that catches the eye is that in all ambiguous grammars at least one ambiguous string is found. AMBER correctly reported all grammar to be ambiguous. This is just as expected, provided that the tool would be given enough time. In default mode all grammars took a maximum of 0.01 seconds to find the first ambiguity, except grammar 55, which took 56.37 seconds. In ellipsis mode all grammars took below 1 second, but grammar 55 took 277.24 seconds. The accuracy on the grammar level of both modes of AMBER is 100%, for our collection of small ambiguous grammars.

Unambiguous small grammars - grammar level Table A.3 presents the results of running AMBER on the small grammars that are not ambiguous. The table shows that it found no ambiguities in all grammars, at least not in strings up to a certain length. The maximum string length that was searched is shown in columns 3 and 5. These lengths vary because the durations of the different tests varied. During the benchmarking we ordered the test results by duration. The grammar that took the least time on its previous test, was run first. A maximum string length of 80 was chosen for the default mode, and 50 for ellipsis mode (the default mode was faster in most cases).
These results look as expected, because there were no false positives. However, we cannot be certain that AMBER would never report an ambiguity of a length greater than our checks.

Ambiguous small grammars - ambiguity level Columns 4 and 6 show the difference between the number of disjunctive and conjunctive ambiguities found by AMBER and our own inspection. In most cases AMBER is able to find all ambiguities. For the grammars that have three production rules that each are disjunctively ambiguous with each other (marked with ^{a}), AMBER reports only two ambiguities. Three production rules can be combined into three different pairs, thus forming three disjunctive ambiguities. If two of them would be resolved using ACCENT's priority declaration, the third one would automatically be resolved too, because the priorities work transitively. Therefore AMBER chooses to report only two ambiguities. This is clearly a case of defining ambiguities from the grammar developer's perspective (see section 2.2).

In some cases AMBER also reports more ambiguities (marked with ^{b}). This is because it uses a different definition of a conjunctive ambiguity. We counted the production rules that could produce a conjunctively ambiguous string, as being just one ambiguity. But with AMBER, every subdivision of the right-hand side of a production rule that can lead to a conjunctive ambiguous string, is reported as a separate ambiguity. For instance, in the grammar

$$S \to AAA, A \to a \mid aa,$$

the first production rule can be split up in two ways to match parts of the string '$aaaa$': **1**) into 'A' and 'AA', which match as shown in (a) and (b) of figure 5.1, and **2**) into 'AA' and 'A', which match as shown in (c) and (d).

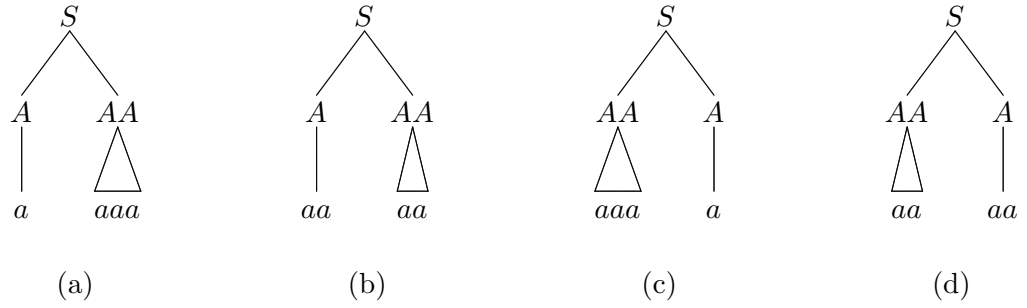

Figure 5.1: Derivations of conjunctive ambiguous string '$aaaa$'

In the cases where the investigated grammar included a cyclic production (of form $A \to A$) this was also reported as an extra ambiguity, but with an empty example tree (marked with ^{c}). Another unexpected output was generated for grammars 67 and 68 (marked with ^{d}). These each include the following production rules $S \to SS$ and $S \to \varepsilon$. This combination seems to trigger the ellipsis mode to report an example derivation tree of infinite length. It starts outputting a tree in which the production $S \to SS$ is applied to its own members over and over again, seemingly without end. Because of this the program did not end, and no measurement could be made.

5.2 Performance

Computation time Tables A.4 and A.5 show the results of the time measurements on the medium and large grammars in default and ellipsis mode. The results of measuring

the unambiguous grammars are also visualized in figures A.1 and A.2. The table shows the duration of testing a grammar for each individual string length. Testing a grammar is stopped if it is reported ambiguous, or if the total duration of the tests of all string lengths exceeded 15 hours.
The duration of generating and checking derivations of increasing length growed exponentially. This can be explained by the number of derivations growing also exponentially by increasing string length. The duration seems also to increase (a little) with the size of the grammars, except for the C grammar. Checking it for ambiguous strings with a maximum length of 7 took 6 hours in default mode, while checking the Java grammar with length 11 took just 2 hours.

For all string lengths of all grammars the ellipsis mode took more time than the default mode. Also checking the strings containing nonterminals for duplicates, is obviously more work.

Memory usage The results of the memory measurements on the unambiguous medium and large grammars are shown in tables A.6 and A.6. Cells that read a '-' indicate the memory could not be measured because the program ran in less than one second. The memory consumption looks to be dependent on the size of the grammar, but only with a very small increase.
It is interesting to see that the memory usage is not dependent on the maximum string length. AMBER needs no extra memory to generate and check more derivations. This could be explained by the fact that it performs a depth-first search of all derivations.

Both tables show that there were no significant differences in the memory consumption of both modes. Generating extra derivations only takes more time, not memory. The extra derivations that the ellipsis mode generates only affects its computation time.

5.3 Termination

Columns 3 and 5 of table A.8 show for all medium and large grammars the time both modes needed to terminate. These values are the sum of the durations of all consecutive runs with increasing string lengths. Columns 4 and 6 show the maximum string length that was checked. On all unambiguous grammars both modes did never terminate, which is as expected. The default mode terminated withing 15 hours on 17 of the 20 ambiguous grammars, the ellipsis mode on 16. The last column shows which of the two versions terminated the fastest. The default mode was fastest in 15 cases, the ellipsis mode in 2 cases.

Both modes were not able to find the introduced ambiguity in grammars C.2 and Java.1. C.2 contains a dangling-else that will only appear in ambiguous strings with a minimum length of 14. When we extrapolate the time measurements of the default mode, it would need around 750.000 years to check all strings of this length!

5.4 Usefulness of the output

A derivation generator like AMBER has detected an ambiguity, when it has found two derivations for the same string. This information holds enough information to construct a user-friendly report:

- **Correct**: the derivations prove the found string is ambiguous.
- **Grammar-oriented**: the derivations are made out of production rules.
- **Localizing**: the derivations include the involved production rules.
- **Succinct**: if the common tops are taken from the derivation trees this results in the shortest possible string and derivations. However, it is uncertain if more derivations exist for the same string.

5.5 Analysis

Scalability In ellipsis mode AMBER should be able to find more ambiguities while given a smaller length parameter. It checks the sentential forms containing nonterminals for duplicates, which might be smaller than their expanded sentences. Our measurements have indeed demonstrated this. Looking at the last column of table A.8, the ellipsis mode always took a smaller or equal string length to terminate than the default mode. But the default mode was much faster in most cases. For most grammars the ellipsis mode was no improvement over the default mode.

Of our grammars, the length of most ambiguous sentential forms with nonterminals, did not differ much from those without nonterminals. Most of the nonterminals could obviously derive a string of only one terminal. Checking sentential forms that contain nonterminals becomes redundant then, because they could be expanded into terminal strings of about the same length.

By our definition, AMBER is not really scalable by using the default or ellipsis mode. The performance of both modes might differ, but their accuracy is the same. Which one of the two will terminate faster depends on the tested grammar. Maybe heuristics could be developed to predict if the checking of nonterminal sentential forms will be useful for a grammar. It should not be hard to compute for each nonterminal of a grammar, the minimal length of the terminal strings that it could derive. If a large number of nonterminals can derive strings of only one terminal, then the default mode of AMBER would probably find ambiguities faster than the ellipsis mode. More research is needed here.

Usability If AMBER finds a grammar ambiguous then this is 100% correct. On the ambiguity level its accuracy is also nearly 100%. Its reports are very useful in locating the sources of ambiguity in a grammar. An example of an ambiguous string and its derivations provide a grammar-oriented, localizing and succinct report. The only deciding factor for practical usability in the use cases is its termination. For unambiguous grammars it will never terminate. How long it takes to terminate on ambiguous grammars is determined by its performance,

and only its computation time. The memory consumption of AMBER is very low, and grows only marginally with the tested string length. All tests remained under 7 MB.

	Default		Ellipsis	
Grammarsize	$t < 5$m.	$t < 15$h.	$t < 5$m.	$t < 15$h.
$\|P\| < 200$	9	10	10	10
$200 \leq \|P\| < 500$	6	8	5	7

Table 5.1: Termination summary of AMBER on medium and large ambiguous grammars (Each category contained 10 grammars)

Table 5.1 shows the number of ambiguous medium and large grammars of our collection that both modes of AMBER were able to terminate on. The ellipsis mode terminated within 5 minutes on all medium grammars. The default mode on all but one. Grammar SQL.1 took just under 1 hour, which is easily within the time limit of the extensive checks (15 hours).

The C and Java grammars could not all be checked within both time limits. The default mode terminated on 6 of the 10 grammars within 5 minutes. The ellipsis mode on only 5 (we also included the measurements on grammars C.1 and C.4, which both took 5.5 minutes). Also not all grammars could be checked within 15 hours. The default mode failed on 1 C and 1 Java grammar, and ellipsis mode on 1 C and 2 Java grammars. Whether this is due to the size and structure of the grammars or the introduced ambiguities is hard to tell.

Both modes were thus practically usable in the tests for use cases 1 and 2, but less in use cases 3 and 4. The default mode did better on the majority of the grammars, because of its higher performance.

Chapter 6

Analysis of MSTA

We used MSTA as the LR(k) test. It tests grammars for inclusion in the LR(k) class, and is thus only able to determine the unambiguity of grammars. The measurement results of MSTA are included in appendix B.

6.1 Accuracy

Ambiguous small grammars - grammar level Table B.1 presents the outcomes of the LR(k) test for the small ambiguous grammars. For all grammars the LR(k) test failed for all tested values of k. The second columns show the maximum value of k with which the LR(k) test was run on each grammar. Just as with the measurements of AMBER on the unambiguous grammars, these results cannot be used to conclude that MSTA will never find a suitable k for each of these grammars. However, ambiguous grammars are never LR(k) so the test should never pass on them. Our measurements are thus as expected.

Unambiguous small grammars - grammar level Table B.2 shows the measurements of MSTA on the unambiguous small grammars. 27 of the 36 grammars passed the LR(k) test, which is 75%. The accuracy on these grammars is 100%. In the other 9 cases the grammars are apparently not LR(k) for the maximum values of k that we tested. However, we can't be certain that there are no larger values of k for which the grammars are LR(k).

6.2 Performance

Computation time The performance of MSTA was measured on the collection of medium and large grammars. The timing results are shown in table B.3. The duration of the tests increased with the size of the grammar and the value of k. But not in a very predictable way. Adding production rules to the grammars did not increase the duration of the tests in a uniform way. For instance, grammars SQL.3 and SQL.5 take much more time to test than the other SQL grammars. Also, testing grammar C.2 is 1.5 times as expensive as the other C grammars, for $k = 3$.

Memory usage The memory consumption on the medium and large grammars is shown in table B.4. Cells that read a '-' indicate that the memory could not be measured because the program ran in less than one second. In general the memory usage also increased with higher numbers of production rules and higher values for k . But again, not in a uniform way. For instance, testing the Pascal grammars for $k = 3$ takes about twice as much memory as the C grammars.

For all tests the memory consumption is fairly low. Just as with AMBER, the computation time reaches an unpractical level faster than the memory consumption. The value of k is more responsible for the increase in both time and memory, than the number of production rules, which is as expected. The amount of LR(k) -items usually grows exponentially with increasing k, but only linearly with the number of production rules.

6.3 Termination

Table B.5 shows the termination times of MSTA on the medium and large grammars. It didn't terminate on the ambiguous grammars within 15 hours, which is as expected. For all unambiguous grammars it terminated within seconds. It turned out that all these grammars are LR(1), which can be explained because they were gathered in Yacc format. Yacc expects its grammars to be in the LALR(1) class, which is a subclass of LR(1). This does not make a very fair comparison, but as explained earlier in section 4.2 we were not able to convert grammars from SDF format.

6.4 Usefulness of the output

The LR(k) test is not an actual ambiguity detection method. It is a test to determine if a grammar belongs to a certain class, which is a subclass of the unambiguous grammars. It can be used to determine the ambiguity on the grammar level, but it is not designed to find individual ambiguities.

If a grammar is found not to be LR(k) for a certain k, then this is detected by the conflicts in the parse table. To understand these conflicts, knowledge is needed about the working of an LR parser. They are not always very comprehensible, as can be seen by the numerous posts about them on the `comp.compilers` newsgroup.
They certainly are not grammar-oriented. Therefore they are also not very localizing and succinct.

6.5 Analysis

Scalability The accuracy and performance of the LR(k) test cannot be traded, so it is not scalable.

Usability Contrary to AMBER, MSTA is only able to decide the non-ambiguity of a grammar. If it does so then it is 100% correct. However, it will never terminate on grammars that are not LR(k). The rightmost column of table B.5 shows the values of k that we were able to test within 15 hours. For the ambiguous medium grammars the maximum value of k was 6. For the large grammars this value was only 3. According to [14], the chances are minimal that a grammar that is not LR(k) for $k = 1$, will be LR(k) for higher values of k. The exponential performance is thus acceptable, because testing for higher values of k will probably be useless.

The LR(k) test was thus only able to test a medium or large grammar for low values of k. If it terminated then this was within the time limits of both use cases. However, it the input grammar is not LR(k), then the test will probably not be very useful. It cannot correctly decide whether the input grammar is ambiguous or not. Its reports also do not help much with finding the sources of ambiguity in the grammar.
In [14] Grune and Jacobs also mention that experience shows that a grammar that is designed to best describe a language without concern for parsing, is virtually never suitable to generate a linear-time parser automatically. This means that a grammar designed with the Meta-environment is unlikely to be LR(k). Even if the tested grammar is unambiguous, the LR(k) test would probably not be useful in the Meta-Environment.

Chapter 7

Analysis of Schmitz' method

The method of Schmitz is the only one we investigated that allows false positives. Because of its conservative approximation it is guaranteed to report no false negatives. It can decide the ambiguity and non-ambiguity of grammars. The measurement results of Schmitz are included in appendix C.

7.1 Accuracy

Ambiguous small grammars - grammar level On the grammar level Schmitz' algorithm should never report non-ambiguity when a grammar is ambiguous. Table C.1 shows the results of our measurements on the small grammars that are ambiguous. The last three columns mention the number of potential ambiguities that were reported using the three precisions that the tool currently supports. Every precision reported ambiguity for every grammar, which is as expected. The accuracy on the ambiguous grammars is thus 100% for each precision. The table also shows that every higher precision reports a lower or equal number of potential ambiguities on all grammars. This is also as expected. The sets of LR(0) and SLR(1) items of a grammar are of equal size, but the SLR(1) position graph contains less transitions. Just as in an SLR(1) parse table, the shift and reduce actions are constrained by the look-ahead. The number of LR(1) items of a grammar is considerably larger, resulting in more fine-grained transitions.

Unambiguous small grammars - grammar level Table C.2 shows the results of the measurements on the unambiguous small grammars. In the case where there were no ambiguities reported, the cells are left blank. Just as with the ambiguous grammars, the higher precisions reported less potential ambiguities than the lower ones, on all grammars. The higher precisions also reported non-ambiguity in more cases. On the collection of unambiguous small grammars, the LR(0), SLR(1) and LR(1) precisions have an accuracy of respectively 61%, 69%, and 86%.

Medium and large grammars - grammar level The results of the measurements on the medium and large grammars are shown in table C.3. No measurements could be made with precision LR(1) for the grammars C.5, Java.2 and Java.4, because the program ran out of memory.
Again all precisions were correct on all ambiguous grammars. Also the number of reported potential ambiguities is inversely proportional with the level of precision. But not necessarily proportional with the grammar size. All Pascal and Java grammars, and some of the C grammars, trigger a large amount of reports in contrast to the other ones.

The LR(1) precision was the only one to correctly reports all unambiguous grammars to be unambiguous. Could this be because these grammars are actually LR(1), as reported by MSTA? If we compare the measurements of MSTA and Schmitz LR(1) on the unambiguous small grammars, there are also no cases in which Schmitz reports potential ambiguities in LR(1) grammars. In [23] Sylvain Schmitz indeed proves that LR(1) grammars will be found unambiguous with LR(1) precision. Our results are thus as expected.

7.2 Performance

The results of the performance measurements of Schmitz' method are shown in table C.4. For nearly all grammars the duration of the LR(0) and SLR(1) tests stayed below 1 second. In these cases the memory consumption could not be measured (the cells read '-'). In the other cases the program needed around 60 MB.

Also the LR(1) test of the HTML, SQL and Pascal grammars were very fast, they stayed under 4 seconds. Running LR(1) on the C and Java grammars took considerably more time. For these grammars the amount of memory that was needed exceeded the amount of physical RAM of the used PC. The operating system started swapping to the hard disk, which caused a big drop in performance. The grammars C.5, Java.2 and Java.4 took even more memory than is supported by the operating system (3GB) and the program crashed. The tests on grammars C.3 and C.4 only stayed 50MB below this limit.

The SLR(1) precision was faster than LR(0) in nearly all cases. The number of nodes in the position graph of both precisions is equal, but the SLR(1) graph contains less valid transitions. They are the same as those of the LR(0) graph, but constrained by look-ahead. Less paths will have to be considered when searching this graph, explaining the higher performance.

Contrary to the other two methods, the memory consumption is the critical factor here, in stead of time. The tool of Schmitz uses a breadth first search to find two paths that generate the same string. Therefore all combinations of paths will have to be stored in memory.

7.3 Termination

Table 7.1 summarizes the termination of Schmitz' tool on the medium and large grammars. It shows that all precisions successfully terminate within 5 minutes, on all ambiguous and unambiguous medium grammars. The LR(0) and SLR(1) also terminated on all large grammars within 5 minutes. Because of its high memory consumption, the LR(1) precision failed

Grammarsize	LR(0) precision		SLR(1) precision		LR(1) precision	
	$t < 5$m.	$t < 15$h.	$t < 5$m.	$t < 15$h.	$t < 5$m.	$t < 15$h.
Unambiguous (3)						
$\|P\| < 200$	3	3	3	3	3	3
$200 \leq \|P\| < 500$	2	2	2	2	1	2
Ambiguous (10)						
$\|P\| < 200$	10	10	10	10	10	10
$200 \leq \|P\| < 500$	10	10	10	10	2	7

Table 7.1: Termination summary of Schmitz method on medium and large ambiguous grammars

to do the same on most of these grammars. However, if it did not run out of virtual memory, its termination time remained widely below the limit of 15 hours. Grammar Java.1 took the longest with 1 hour and 20 minutes.

7.4 Usefulness of the output

This method detects a potential ambiguity by the existence of two paths through the position graph, that correspond to the same string in the approximated language (see section 3.7). All paths through the position graph will start at position $S' \rightarrow \cdot S\$$, and end at $S' \rightarrow S \cdot \$$. Because the two paths are different they split up and rejoin at certain points. At the moment two paths join it is certain that a potential ambiguity is found. From then on, both paths can always be continued to $S' \rightarrow S \cdot \$$ in the same way.

The current implementation only reports the position the two paths are at, just before they rejoin. The next transitions of both paths that lead to the common position, are either two reduces or a shift and a reduce. So there is a reduce/reduce or a shift/reduce conflict, just like in a LR parser.
Let us take grammar 2 of the collection of small grammars as an example. One of the ambiguities of grammar 2 is the + operator having no associativity. The string '$a+a+a$' has two derivation trees, shown in figure 7.1. Figure 7.2 shows the reports of the LR(1) precision of Schmitz on grammar 2. The first report corresponds to the shift/reduce conflict that would be encountered by a parser, when it has reached the second +. Choosing to shift would result in parse tree (a), while reducing first would result in parse tree (b).

These reports are clearly not very grammar-oriented. However, they mention the production rules that might appear in the derivation of an ambiguous string. Still, this is not as localizing and succinct as the derivation trees given by a AMBER for instance.
At the time an ambiguity is found, there might be more useful information available in the state of the algorithm. For instance the two paths might be used to construct the complete derivations. However, Schmitz has not yet found a way to generate comprehensible ambiguity reports from them. The main problem is that the generated string is included in the approximated language, but does not necessarily have to be included in the original language. It can be easily verified whether the generated string is included in the original language, but if it is

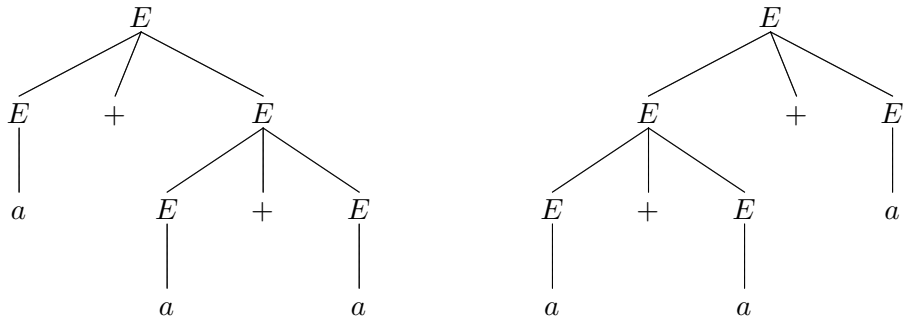

Figure 7.1: Example of two parse trees of the sentence '$a + a + a$', of grammar 2

```
4 potential ambiguities with LR(1) precision detected:
        (E -> E . + E , E -> E + E . )
        (E -> E * E . , E -> E . + E )
        (E -> E . * E , E -> E * E . )
        (E -> E . * E , E -> E + E . )
```

Figure 7.2: Output of Schmitz LR(1) on grammar 2.

not, this does not mean that the report is a false positive. It still might indicate an existing ambiguity. Because of the approximation, it could be a 'shortcut' for a longer ambiguous string in the original language. For instance, the string corresponding to the first report of figure 7.2 is '$E + E$'. Determining if a report is a false one, will be undecidable in some cases. Because else Schmitz would have found an algorithm that decides an undecidable problem (see section 2.3).

7.5 Analysis

Scalability Looking at the accuracy and performance, we see that the SLR(1) precision is superior to LR(0) in both areas. It finds less or equal ambiguities in all grammars, using less computation time and memory. It seems only sensible to compare SLR(1) and LR(1) to each other.

LR(1) obviously has the best accuracy on our grammars, while SLR(1) has the best performance. The accuracy on the unambiguous grammars of SLR(1) is reasonably high (69%), although the number of reports on the Pascal, C and Java grammars seems too impractical. LR(1) has an accuracy of 86% on the grammar level (unambiguous grammars), and produces a more reasonable amount of reports (although we do not know how accurate they are). However, the downside is that on the larger grammars it might need too much memory.

When its memory consumption remained low enough to avoid swapping, LR(1) offered the best accuracy for a low price in performance. In the other cases, SLR(1) would be the only sensible alternative of the tested implementation. However, its number of reports might be too high, considering their low usefulness. The scalability of the tool is probably too coarse-

grained. Another precision with an accuracy between those of SLR(1) and LR(1) might be more helpful when LR(1) fails. In [24] Schmitz draws the same conclusion and mentions this precision might be LALR(1).

Usability To be useful, a method first needs to terminate in an acceptable amount of time. We saw that the LR(0) and SLR(1) precisions terminated within the limits of all four use cases. The termination score of LR(1) was 100% in use cases 1 and 2, 70% in use case 4, and only 20% in use case 3. During the tests in which it did terminate in this last case, it needed to swap memory to the hard disk. It is arguable whether to allow this behaviour for a quick check. Also on two of the tests of use case 4 that it successfully terminated on, its memory consumption came very close to the 3GB limit of the operating system.

After a method has successfully terminated, its correctness of its report is important. If a grammar is ambiguous then Schmitz' method is guaranteed to report this. It is thus important that it will not give too much false positives on unambiguous grammars. The accuracy of the LR(0), SLR(1) and LR(1) precisions on our unambiguous small grammars is respectively 61%, 69%, and 86%.

A report does not only have to be correct, but it also has to help the user in disambiguating his grammar. The usefulness of Schmitz' output is not very comprehensible, because this method is fairly new. The conservative approximation makes it terminate in finite time, but the downside is that the ambiguity reports become hard to comprehend.

The big challenge is now to make a synthesis of all these results, and determine the practical usability of Schmitz' method. Unlike the analysis of the other two ADMs, the situation is less clear here. On nearly all criteria this method is not perfect. Which criteria (or combination) is the most decisive in determining its practical usability? Coming up with an absolute score might no be possible. However, determining which precision was most usable in which use case is something we can do:

Of the three precisions, LR(1) did best in use cases 1 and 2. It offered a high accuracy for low performance. But because of its low termination score, it was certainly not usable in use case 3. SLR(1) is the only sensible alternative of the tested tool. However, its number of reports on the C and Java grammars might be too impractical, given their low usefulness. None of the precisions of the current implementation of Schmitz' method might thus have been practically usable in use case 3. For the same reasons, SLR(1) might also not have been very useful in use case 4. It is questionable whether the termination of LR(1) was high enough to make it the best option for this use case. Obviously, more research is needed here.

Chapter 8

Hybrid ADM

While searching for a way to verify the reports of Schmitz' method, we have come up with a new hybrid ADM. It is a combination of Schmitz' method and a derivation generator. Because it was not not in the scope of this research project, it is still in its early development phase. In this intermezzo chapter we present our idea and demonstrate some measurement results from an experimental prototype.

8.1 Introduction

The idea is to use a derivation generator to investigate the potential ambiguities of a report from Schmitz' method in more detail. The potential ambiguities are used to filter out a potentially ambiguous subset of the grammar. This subset is then searched for real ambiguities with a derivation generator. Because only a subset of the grammar is checked, less derivations have to be generated. If the investigated grammar is ambiguous, it might be reported in less time compared to a normal derivation generator. Also, Schmitz' method is able to detect the unambiguity of a grammar, while a derivation generator would run forever. If Schmitz' method reports the investigated grammar to be unambiguous (which is then 100% correct), there is no need to run a derivation generator at all.

The method consists of three parts:

1. Schmitz' method is run on the grammar with a certain precision (preferably the highest).

2. From the reported potential ambiguities, a potentially ambiguous grammar subset is filtered.

3. The potential ambiguous subset is searched for ambiguities with a derivation generator.

Steps 1 and 3 are pretty straightforward. Step 2 is explained in the next section.

8.2 Filtering of potentially ambiguous grammar subset

Schmitz' method detects a potential ambiguity by the existence of an ambiguous string in the language of the approximated input grammar. This string is ambiguous for the start symbol of the approximated grammar, and substrings of it might be ambiguous for nonterminals lower in the dependency graph (for an example see figure 4.1). We will call these nonterminals (including the start symbol) the *potentially ambiguous nonterminals*. By replacing the start symbol of a grammar by a potentially ambiguous nonterminal, a *potentially ambiguous subset* of the grammar is acquired. It accepts all derivations of the potentially ambiguous nonterminal. For each reported potential ambiguity we try to find the potentially ambiguous nonterminal that is the lowest in the dependency graph of the grammar.

Schmitz' method makes a breadth first search through the position graph, to find two different paths that produce the same string (see section 3.7). These paths both start in q_s, and follow the same transitions until they split up. If they rejoin again later on, then an ambiguous string in the approximated language is found. From then on they follow the same positions to q_f. The lowest nonterminal that the generated string is ambiguous for, can be found by traversing the paths backwards from the split point and forwards from the join point, until a d_i (derive) and r_i (reduce) transition of the same nonterminal are found. Unfortunately, as described in section 3.7 (Scalability part), the d_i transitions do not have to match the r_i transitions in the approximated grammar. However, both paths will at least always derive from and reduce to the start symbol.

If a potentially ambiguous nonterminal is found, its potentially ambiguous grammar subset is checked with a derivation generator. It will generate all possible derivations of the nonterminal, and checks them for duplicates. In the worst case the potentially ambiguous nonterminal is the start symbol, and the whole grammar is potentially ambiguous. Checking it would be the same as applying a derivation generator without filtering, and no performance gain can be made. However, each level that this nonterminal appears lower in the dependency graph might exponentially decrease the number of derivations to search.

If Schmitz' method finds multiple potential ambiguities, then for each of these the corresponding potentially ambiguous nonterminal has to be extracted. What to do with these multiple nonterminals has yet to be investigated further. Checking them all in sequence might become more expensive than checking the entire grammar only once, especially if they depend on each other. Another option is to select a nonterminal that depends on all nonterminals, and do only one search with the derivation generator. This will always be faster or just as fast as checking the entire grammar from the start symbol.

The filtering process might also be applied incrementally to decrease the potentially ambiguous subset even further. The left out production rules do not interfere the approximation of the potentially ambiguous nonterminal, and an even smaller subset might be acquired.

8.3 Experimental prototype

We have modified the Bison tool of Schmitz, to filter out the potentially ambiguous subset of a grammar as described above. For this we had to continue the paths from the join points to q_f.

The original version terminated two paths at a join point, because then it is already certain a potential ambiguity is found. For each potential ambiguity, we traverse its paths in both directions to find the potentially ambiguous nonterminal. If multiple potentially ambiguous subsets are found then we select the smallest subset of the grammar which includes all these subsets. We then apply this filtering process again until the potentially ambiguous subset does not change anymore. Finally we use AMBER to search the subset for ambiguous strings.

8.4 Measurement results

Grammar	**AMBER default**		**Hybrid ADM**						**Improvement**	
			Schmitz			AMBER default		Total		
	Time[1]	Length[2]	Prec.	Runs	Time[1]	Time[1]	Length[2]	Time[1]	Time	Length
SQL.1	3478.18	15	LR(1)	2	1.06	2355.38	11	2356.44	1.5x	4
Pascal.3	53.11	11	LR(1)	2	7.59	48.60	7	56.19	0.95x	4
C.1	60.29	5	SLR(1)	2	1.19	0.22	3	1.41	43x	2
C.2	22946.77[3]	14	SLR(1)	2	1.30	8338.18[3]	11	8339.48	2.8x	3
C.4	60.57	5	SLR(1)	3	21.94	53.42	3	22.28	2.7x	2
C.5	1622.29	6	SLR(1)	3	15.25	7.12	5	22.37	73x	1
Java.1	7594.87[4]	13	SLR(1)	2	48.08	7160.37[4]	13	7208.45	1.1x	0
Java.3	7562.72	11	SLR(1)	2	50.08	7111.20	11	7161.28	1.1x	0

[1]Termination time in seconds.
[2]Minimal string length needed to find an ambiguity.
[3]Time measurement up to string length 7.
[4]Time measurement up to string length 11.

Table 8.1: Differences in termination time on medium and large grammars between AMBER (default) and our hybrid ADM.

To test the potential performance gain over AMBER, we checked the ambiguous medium and large grammars with our new prototype (only the ones for which AMBER default took over one minute). The results are shown in table 8.1. Columns 2 and 3 show the termination time and needed string length of AMBER default, of our original measurements. The next 6 columns show the results of our hybrid prototype. It shows which precision of Schmitz' method we ran, how often, and the total computation time the runs needed for the original algorithm and the filtering of the potentially ambiguous subsets. The ambiguities in the C.2 and Java.1 grammars appear at a string length that would take too long to actually test. Therefore we only tested these grammars up to a lower string length.

Columns 7 and 8 show the termination time and needed string length of AMBER default on the filtered subsets of the grammars. It was always faster than on the entire grammar. Also the needed string length was shorter in 6 of the 8 cases. For all grammars the potentially ambiguous nonterminal was indeed lower in the dependency graph than the start symbol, resulting in less derivations to check.

Column 9 shows the total time of running both Schmitz and AMBER. Column 10 shows the factor with which our method was faster than running AMBER default alone. On all but one

grammar it was faster, with the biggest improvement on grammar C.5 (73x). On grammar Pascal.3 AMBER was 5 seconds faster, but Schmitz' method took 8 seconds, undoing this gain.

The timing results of Schmitz' method are higher than those of the original method (appendix 7.2), because of the continuations of the paths to q_f and searching them for the potentially ambiguous nonterminal. The continuations of the paths also took more memory (about twice as much), and the LR(1) precision could not be used for the large C and Java grammars. Therefore we used SLR(1) instead. Despite the high number of potential ambiguities reported by the SLR(1) precision, it found a much more reasonable amount of potentially ambiguous nonterminals. It turned out that most of the potential ambiguities corresponded to the same nonterminal. Also, on grammars C.4 and C.5 the potentially ambiguous subset could be reduced even further by applying our filter with SLR(1) precision again.

In all cases, running Schmitz and filtering its reports took below 1 minute. This is only a small amount of time compared to what can be gained by running AMBER on a subgrammar. The only downside of the filtering is that the memory consumption of LR(1) becomes even bigger. However, despite its high number of potential ambiguities, SLR(1) reported only a low number of potentially ambiguous nonterminals, and could sometimes be applied incrementally. Still, we expect that using a higher precision of Schmitz' method will result in the smallest potentially ambiguous subset.

8.5 Contribution

We have presented a new hybrid ambiguity detection method, which is a combination of Schmitz' ADM and a derivation generator. It can be used to verify the potential ambiguities reported by Schmitz, or to speed up the search process of a derivation generator. Our contribution is the use of Schmitz' ADM to filter out parts of the grammar that are guaranteed to be unambiguous.

It has the best characteristics of both methods. It is able to detect both ambiguity and unambiguity. In the first case this answer is always correct. In the second case the potential ambiguities of Schmitz are verified with 100% accuracy. A derivation generater also gives the most comprehensible ambiguity reports, which locate the exact source of ambiguity. If the grammar is indeed ambiguous the computation time of the derivation generator can be shortened several factors, because of the subset filtering. Running Schmitz method usually takes only a small amount of extra time, but it can exponentially shorten the computation time of the derivation generator.

8.6 Future work

Because of the limited time of this project, and this subject not being in the initial scope, we were not able to do a thourough investigation of the possibilities of this combination. We have built a small prototype to show its potential gain, but more research is needed to optimize it further. More ways have to be explored to filter more detailed information out of Schmitz'

algorithm, to find even smaller subsets. For instance, we now define the potentially ambiguous subset with a nonterminal, but it could also be a single production of the nonterminal. Also a proof has to be constructed to show that no ambiguities are overlooked.

The current idea uses Schmitz as a coarse-grained filter, and a derivation generator as a very intensive check. Maybe more stages of filtering can be developed, which gradually exclude small unambiguous portions of the grammar.

Chapter 9

Summary and conclusions

In this project three implementations of ambiguity detection methods where investigated: AMBER [25] (a derivation generator), MSTA [21] (a parse table generator used as LR(k) test) and a modified Bison tool that implements the ADM of Schmitz [23]. The goal was to find out which ADM has the best practical usability in common use cases of the Meta-Environment. For this we have set up a collection of sample grammars of different sizes, with which we benchmark the implementations of the ADMs. We try to answer the following research questions:

1. What is the accuracy of each method on the sample grammars?
2. What is the performance of each method on the sample grammars?
3. What is the termination of each method on the sample grammars?
4. How useful is the output of each method?
5. Which methods are practically usable in the stated use cases?

Chapters 5, 6 and 7 answered all these questions for each investigated ADM individually. In this chapter we will compare the results of all ADMs and try to answer the main research question. It ends with ideas for future work.

9.1 Overview of measurement results

Table 9.1 summarizes the measurement results on our collections of small, medium and large grammars. It shows that AMBER was always correct on our ambiguous small grammars and that its reports are very useful. Also its memory consumption was very low in all situations. Its only two problems are that it never terminates on unambiguous grammars and that it might take very long to terminate on ambiguous grammars. It was not able to terminate on all ambiguous grammars with a size between 200 and 500 production rules within the time limits of all use cases. However, both modes terminated on nearly all grammars below 200 production rules, within 5 minutes. The default mode was the fastest of the two in most situations.

	AMBER		MSTA	Schmitz		
	Default mode	Ellipsis mode		LR(0) precision	SLR(1) precision	LR(1) precision
Accuracy (grammar level)						
• ambiguous	100%	100%	n.a.	100%	100%	100%
• unambiguous	n.a.	n.a.	100%[1]	61%	69%	86%
Performance[2]						
• computation time	–	– –	– –	++	++	++
• memory consumption	++	++	–	++	++	–
Termination (unambiguous)						
• use case 1	0%	0%	100%	100%	100%	100%
• use case 2	0%	0%	100%	100%	100%	100%
• use case 3	0%	0%	100%	100%	100%	50%
• use case 4	0%	0%	100%	100%	100%	100%
Termination (ambiguous)						
• use case 1	90%	100%	0%	100%	100%	100%
• use case 2	100%	100%	0%	100%	100%	100%
• use case 3	60%	50%	0%	100%	100%	20%
• use case 4	80%	70%	0%	100%	100%	70%
Usefulness of output[2]						
• grammar-oriented	++		–	–		
• localizing	++		–	–		
• succinct	++		–	–		

[1]MSTA terminated on 75% of the unambiguous small grammars.
[2]Scores range from – – to ++

Table 9.1: Summary of measurement results

MSTA terminated on 75% of the small unambiguous grammars and on all medium and large unambiguous grammars. If it terminates it is 100% correct. However, it never terminated on the ambiguous grammars. This fact, together with its incomprehensible reports, makes it the least usable ADM of the three.

Schmitz' method was 100% correct on the ambiguous small grammars, because of its conservative approach. On the unambiguous grammars, each higher precision had a higher accuracy. Each higher precision also produced a lower or equal amount of ambiguity reports on all grammars.
The LR(0) and SLR(1) precisions needed very little computation time and memory. On the grammars below 200 production rules this was also the case for LR(1). However, on most of the larger ambiguous grammars its memory consumption became too high and it started swapping. It sometimes needed even more than is supported by the OS and the program crashed. In the cases that it did terminate, it produced the least amount of ambiguity reports. The report count of the other two precisions seemed too impractical in most cases, because they are hard to verify.

9.2 Comparison of practical usability

In the analysis chapters we have determined the practical usability of each individual ADM. We can now determine which ADM was most practically usable in each of the following 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.

We will assume that there is a high chance that a grammar developed with the Meta-Environment will be ambiguous the first time it is checked. Therefore we will compare the practical usability of the investigated ADMs on our collection of medium and large ambiguous grammars. But first, it is very important to keep in mind this thesis is about methods that try to solve an undecidable problem. There will always be grammars for which there is no proper solution. Whether a method is usable in a certain situation depends for a great deal on the tested grammar, so an ADM that is useful in every situation can never exist. The findings on our grammar collections may thus only have a limited range.

Use cases 1 and 2 On all medium grammars (size < 200 production rules), AMBER was the most practically usable. The default mode terminated within 5 minutes on 9 of the 10 grammars, and the ellipsis mode on all 10. The default mode was faster in most cases. If AMBER terminates then its reports are 100% accurate and very helpful in resolving the ambiguity.
Of Schmitz' method, the LR(1) precision was the most usable in use cases 1 and 2. It also terminated within 5 minutes on all ambiguous grammars, but its reports are not very comprehensible. Therefore it was not as usable as AMBER.

Use case 3 For the large grammars there seems to be no ADM that is really usable as a quick check. It looks like they become too large to make the investigated methods terminate within 5 minutes. None of the precisions of Schmitz' tool was really usable in this case. LR(1) only terminated on 20% of the ambiguous grammars, because it needed to swap memory to the hard disk. The SLR(1) precision terminated on all grammars, but its number of (incomprehensible) reports looks too impractical. The default mode of AMBER might be the most usable here, with a termination score of 60%.

Use case 4 Because of its less strict time limit of this use case, the ADMs successfully terminated on a lot more ambiguous grammars. The default mode of AMBER in 80% and Schmitz' LR(1) in 70%. Because of the same reasons as in use cases 1 and 2, AMBER was the most practically usable ADM here.

These results are summarized in table 9.2. It shows that the LR(k) test (in the form of MSTA) was the least usable of the three. The reason for this is that it never terminates on grammars

that are not LR(k). The chance that a grammar developed with the Meta-Environment is LR(k) is very low. Also, its reports are not designed to indicate the sources of ambiguity in a grammar. With a slightly lower performance but higher accuracy, Schmitz LR(1) outweighed the LR(k) test for $k = 1$. Only in the case a grammar is LR(k) for $k > 1$, the LR(k) test is sure to prove this. However, according to [14], the chances for this are minimal. If Schmitz LR(1) memory consumption is not too high, the LR(k) test will probably not be an option anymore.

AMBER was the most practically usable of the three investigated ADMs on the grammars of our use cases. In chapter 8 we showed that the prototype of our new hybrid ADM was faster than the default mode of AMBER, on the medium and large grammars that AMBER default originally need more than 1 minute for. Grammar C.5 could even be tested within 5 minutes. All its other characteristics are of course the same as AMBER, with the exception that it is also able to detect the unambiguity of grammars, with the accuracy of the used precision of Schmitz. Therefore we can conclude that the hybrid ADM was the most usable in all our tested use cases.

9.3 Conclusion

Use case			**AMBER**		**MSTA**	**Schmitz**		
Nr	Grammar size	Time limit	Default mode	Ellipsis mode		LR(0) precision	SLR(1) precision	LR(1) precision
1	$\|P\| < 200$	$t <$ 5m.	+++	+++	– –	+/–	+	++
2	$\|P\| < 200$	$t <$ 15h.	+++	+++	– –	+/–	+	++
3	$200 \leq \|P\| < 500$	$t <$ 5m.	+/–	+/–	– –	– – –	– –	– –
4	$200 \leq \|P\| < 500$	$t <$ 15h.	++	+	– –	– – –	– –	+/–

Usability scores range from – – – to +++

Table 9.2: Usability on ambiguous grammars of investigated ADMs in use cases

In this project we have investigated which ambiguity detection method was most usable in common use cases of the Meta-Environment. We have tested if the methods were usable as a quick check (termination within 5 minutes) or as an extensive overnight check (termination within 15 hours) on grammars of different sizes. It is important to keep in mind that the usability of an ADM depends for a large part on the tested grammar. Our findings may thus not be easily extended to other grammars of the same size.

Table 9.2 summarizes the practical usability of the investigated ADMs on our grammars of the use cases. AMBER was the most usable method in all cases. In use cases 1, 2 and 4 it was very usable, but its performance prevented it from terminating on some grammars within the time limit of use case 3. This and its nontermination on unambiguous grammars are its biggest problems. All its other characteristics were excellent.
The LR(1) precision of Schmitz was also pretty usable on the medium grammars. Its only drawback was that its reports are hard to comprehend. For most of the large grammars its memory consumption was too high. The SLR(1) precision did always finish in a couple of seconds, but its high number of reports seemed too impractical in most cases.
The LR(k) test (MSTA) was the least usable of the three, because it never terminates on grammars that are not LR(k).

The insights gained during this project have led to the development of a new hybrid ADM. It uses Schmitz' method to filter out parts of a grammar that are guaranteed to be unambiguous. The remainder of the grammar is then tested with a derivation generator, which might find ambiguities in less time. The method is also able to detect the unambiguity of a grammar, while a normal derivation generator would run forever. We have built a small prototype which was indeed faster than AMBER on the tested grammars, making it the most usable ADM of all.

Chapter 10

Future work

The initial goal of this project was to find a suitable ADM for the Meta-Environment. We started with making a comparison of the practical usability of the existing methods. This turned out to be enough work to fill up the entire amount of given time. A lot of interesting areas are left to explore.

Usability experiments For all investigated ADMs, we have measured various characteristics that influence their practical usability. For some of these characteristics we were able to measure absolute values, for others we applied a more subjective grading. Also it was not always clear how all characteristics should be combined. For instance, we were not fully able to determine the practical usability of the method of Schmitz in use case 3. Usability experiments might be performed to gather more accurate requirements for an ADM.

Meta-Environment We did not get to comparing the ADMs against all distinctive features of the Meta-Environment. More tests will have to be performed with grammars that are aimed at an SGLR parser. They are probably less close to being LR(k), which might give different results for the LR(k) test and Schmitz' method. Also more performance measurements would have to be done to get an impression of the costs of checking grammars without a separate lexical syntax definition. For they will generally contain more production rules.

There can also be looked into how the different disambiguation constructs of SDF can be taken into account while checking for ambiguities. A method should of course not mention any ambiguities that are already resolved by the user. AMBER turns an Earley parser into a derivation generator, but this can also be done with the SGLR parser of the Meta-Environment. An SGLR parse table includes the priority and associativity declarations and follow restrictions, which will then be respected while generating derivations. The other parse tree filters of SDF might be applied after the generator has found an ambiguous string.
Schmitz' method is effectively also a search through a parse table (depending on the used equivalence relation), and might also be applied to the SGLR parse tables. The current SGLR parser uses SLR(1) parse tables, so using them would become comparable to the current SLR(1) precision of Schmitz method. On our grammar collection it often produced a lot of potential ambiguities. However, it might still be useful with our hybrid ADM, to filter

out potentially ambiguous grammar subsets.

Hybrid ADM Our idea for the combination of Schmitz' method and a derivation generator, as described in chapter 8, also opens up a new area of research. More work is needed to fully explore its possibilities.

Schmitz' method Schmitz' tool and AMBER both search a parse table for (potentially) ambiguous strings. Their difference between storing all paths in memory or doing a depth first search shows off in their computation time and memory consumption. Schmitz LR(1) precision needed too much memory on our large grammars, which might be avoided if a depth first search was used. This will also make it more usable for our hybrid ADM, which needs even more memory for the continuations of the ambiguous paths. Its computation time will probably grow, but the question is how much.

Other ADMs Our overview of the currently available ADMs is not complete. Not all existing methods have been measured. It would be interesting to see if the method of Cheung and Uzgalis would be an improvement of AMBER. Will the premature termination of derivation trees lead to a substantial improvement in accuracy and performance? Also the approximating ADM of Brabrand, Giegerich and Møller looks promising. How well will it behave in comparison to the ADM of Schmitz?

Ambiguity definition But maybe first of all, more fundamental research might be done to find a proper definition for the source of an ambiguity in a grammar (see section 2.2). The ambiguity of a grammar can then be expressed more formally. This might lead to more insight into the probability of false reports of the approximating methods.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice Hall Professional Technical Reference, 1972.

[3] John Aycock and R. Nigel Horspool. Schrödinger's token. *Software: Practice & Experience*, 31(8):803–814, 2001.

[4] Sylvie Billot and Bernard Lang. The structure of shared forests in ambiguous parsing. In *ACL'89*, pages 143–151. ACL Press, 1989. `http://www.aclweb.org/anthology/P89-1018`.

[5] Eric Bouwers, Martin Bravenboer, and Eelco 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)*, pages 82–96, Braga, Portugal, March 2007. `http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2007-004.pdf`.

[6] Claus Brabrand, Robert Giegerich, and Anders 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. `http://www.brics.dk/~brabrand/grambiguity/`.

[7] Bruce S. N. Cheung and Robert C. Uzgalis. Ambiguity in context-free grammars. In *SAC '95: Proceedings of the 1995 ACM symposium on Applied computing*, pages 272–276, New York, NY, USA, 1995. ACM Press. `http://doi.acm.org/10.1145/315891.315991`.

[8] B.S.N. Cheung. *A Theory of automatic language acquisition*. PhD thesis, University of Hong Kong, 1994.

[9] Charles Donnely and Richard Stallman. *Bison version 2.1*, September 2005. `http://www.gnu.org/software/bison/manual/`.

[10] Robin D. Dowell and Sean R. Eddy. Evaluation of several lightweight stochastic context-free grammars for rna secondary structure prediction. *BMC Bioinformatics*, 5:71, 2004. `http://www.citebase.org/abstract?id=oai:biomedcentral.com:1471-2105-5-71`.

[11] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970. `http://doi.acm.org/10.1145/362007.362035`.

[12] Giorgios Robert Economopoulos. *Generalised LR parsing algorithms*. PhD thesis, Royal Holloway, University of London, August 2006. http://homepages.cwi.nl/~economop/new/thesis/publishedThesis.pdf.

[13] Saul Gorn. Detection of generative ambiguities in context-free mechanical languages. *J. ACM*, 10(2):196–208, 1963. http://doi.acm.org/10.1145/321160.321168.

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

[15] James J. Horning. What the compiler should tell the user. In *Compiler Construction, An Advanced Course, 2nd ed.*, pages 525–548, London, UK, 1976. Springer-Verlag.

[16] Harry B. Hunt III, Thomas G. Szymanski, and Jeffrey D. Ullman. On the complexity of LR(k) testing. *Communications of the ACM*, 18(12):707–716, 1975. http://doi.acm.org/10.1145/361227.361232.

[17] Saichaitanya Jampana. Exploring the problem of ambiguity in context-free grammars. Master's thesis, Oklahoma State University, July 2005. http://e-archive.library.okstate.edu/dissertations/AAI1427836/.

[18] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005. http://doi.acm.org/10.1145/1073000.

[19] Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *ASMICS Workshop on Parsing Theory*, Technical Report 126-1994, pages 89–100. Università di Milano, 1994. http://citeseer.ist.psu.edu/klint94using.html.

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

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

[22] Mehryar Mohri and Mark-Jan Nederhof. Regular approximations of context-free grammars through transformation. In Jean-Claude Junqua and Gertjan van Noord, editors, *Robustness in Language and Speech Technology*, chapter 9, pages 153–163. Kluwer Academic Publishers, 2001. http://citeseer.ist.psu.edu/mohri00regular.html.

[23] Sylvain 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.

[24] Sylvain 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. http://www.i3s.unice.fr/~mh/RR/2006/RR-06.37-S.SCHMITZ.pdf.

[25] Friedrich Wilhelm Schröer. AMBER, an ambiguity checker for context-free grammars. Technical report, compilertools.net, 2001. http://accent.compilertools.net/Amber.html.

[26] Friedrich Wilhelm Schröer. ACCENT, a compiler compiler for the entire class of context-free grammars, second edition. Technical report, compilertools.net, 2006. http://accent.compilertools.net/Accent.html.

[27] Mikkel Thorup. Controlled grammatic ambiguity. *ACM Trans. Program. Lang. Syst.*, 16(3):1024–1050, 1994. http://doi.acm.org/10.1145/177492.177759.

[28] Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized lr parsers. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 143–158, London, UK, 2002. Springer-Verlag.

[29] J. Yang, J. Wells, P. Trinder, and G. Michaelson. Improved type error reporting. In M. Mohnen and P. Koopman, editors, *Proceedings of 12th International Workshop on Implementation of Functional Languages*, pages 71–86. Aachner Informatik-Berichte, 2000.

Appendix A

Results: AMBER

A.1 Accuracy

		Default		Ellipsis	
Grammar	# A	Reports	Delta	Reports	Delta
1	3	3		3	
2	3	3		3	
3	2	2		2	
4	1	1		1	
5	3	2	-1[a]	2	-1[a]
6	2	2		2	
10	2	2		2	
11	5	5		5	
12	4	4		4	
13	2	2		2	
14	6	6		6	
18	1	1		1	
21	6	6		6	
22	5	5		5	
23	1	2	1[b]	2	1[b]
25	2	2		2	
27	2	2		2	
29	1	1		1	
30	2	2		2	
31	5	5		5	
32	4	4		4	
33	2	2		3	1[c]
34	8	7	-1[a]	8	0[ac]
38	6	6		6	

Table A.1: Number of ambiguities found by AMBER on ambiguous small grammars (cont'd below)

		Default		Ellipsis	
Grammar	# A	Reports	Delta	Reports	Delta
39	1	1		1	
41	1	1		1	
42	4	4		4	
43	1	2	1[b]	2	1[b]
44	3	4	1[b]	4	1[b]
46	3	3		3	
48	1	1		1	
49	5	8	3[b]	8	3[b]
50	1	1		1	
51	1	1		1	
52	1	1		1	
53	1	1		1	
55	1	1		1	
56	1	1		1	
59	1	1		1	
60	3	3		3	
61	1	1		1	
62	1	1		1	
63	3	3		3	
64	2	2		2	
66	1	1		1	
67	5	5		-[d]	
68	6	6		-[d]	
84	1	1		1	

[a]missed a disjunctive ambiguity
[b]extra conjunctive ambiguities because of different definition
[c]extra report of cyclic production
[d]no results because of infinite example derivation tree in report

Table A.2: Number of ambiguities found by AMBER on ambiguous small grammars (cont'd)

	Default		Ellipsis	
Grammar	Reports	Max length	Reports	Max length
7	0	77	0	50
8	0	28	0	29
9	0	80	0	26
15	0	32	0	17
16	0	80	0	35
17	0	55	0	19
19	0	39	0	17
20	0	80	0	50
24	0	19	0	13
26	0	80	0	50
28	0	44	0	31
35	0	80	0	50
36	0	80	0	50
37	0	80	0	50
40	0	80	0	50
45	0	39	0	39
47	0	41	0	23
54	0	80	0	50
57	0	80	0	50
58	0	80	0	50
65	0	80	0	50
69	0	23	0	17
70	0	20	0	18
71	0	20	0	20
72	0	25	0	17
73	0	34	0	20
74	0	30	0	24
75	0	80	0	50
76	0	15	0	15
77	0	32	0	21
78	0	80	0	50
79	0	80	0	50
80	0	80	0	50
81	0	28	0	29
82	0	80	0	50
83	0	80	0	50

Table A.3: Number of ambiguities found by AMBER on unambiguous small grammars

A.2 Performance

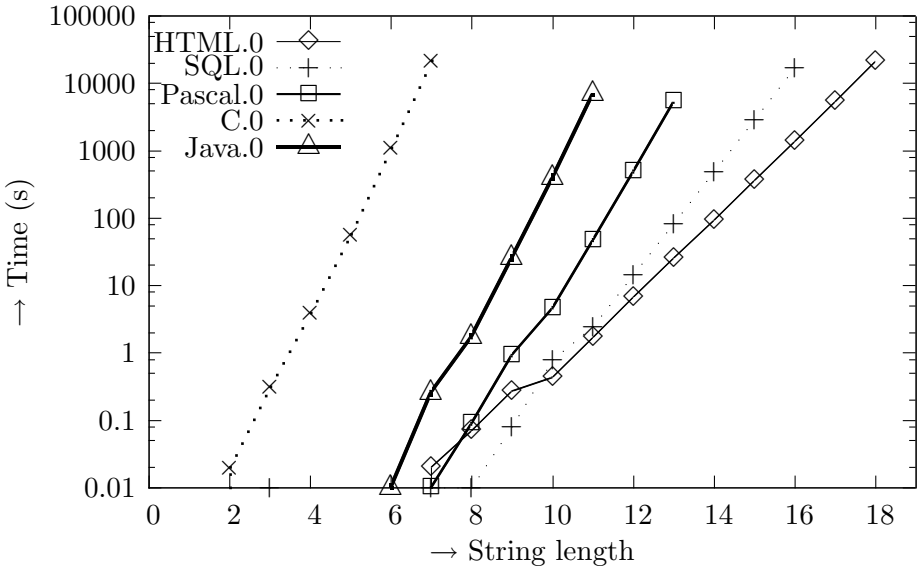

Figure A.1: Computation time of AMBER default on unambiguous medium and large grammars

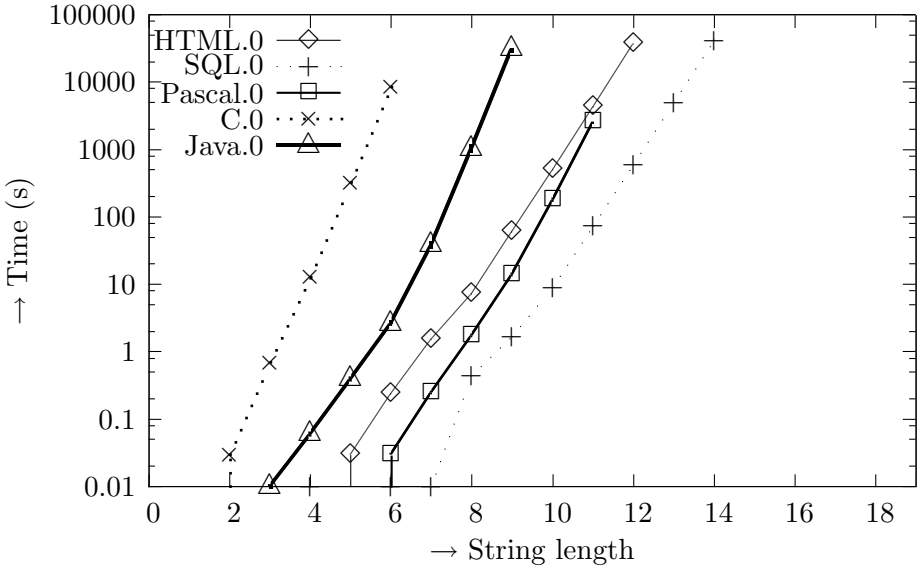

Figure A.2: Computation time of AMBER ellipsis on unambiguous medium and large grammars

Grammar	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
HTML.0	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.07	0.27	0.44	1.69	6.73	25.46	94.38	359.86	1389.58	5349.68	20942.39	*aborted*
SQL.0	0.00	0.00	0.01	0.00	0.00	0.00	0.01	0.01	0.08	0.79	2.47	14.69	82.91	487.39	2905.44	17170.48	*aborted*		
SQL.1	0.01	0.01	0.01	0.01	0.01	0.02	0.01	0.03	0.15	0.87	3.42	14.44	82.02	486.57	2890.60				
SQL.2	0.00	0.00	0.00	0.00	0.00	0.00	0.00												
SQL.3	0.00	0.01	0.00	0.01	0.00	0.00													
SQL.4	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.03	0.15										
SQL.5	0.00	0.00	0.01	0.00	0.00	0.01	0.01	0.03	0.15	0.75	3.32								
Pascal.0	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.09	0.93	4.66	47.34	497.30	5341.73	*aborted*					
Pascal.1	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.11	0.51										
Pascal.2	0.00	0.00	0.00	0.00	0.00	0.01	0.02												
Pascal.3	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.09	0.91	4.36	47.73								
Pascal.4	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.09											
Pascal.5	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.10											
C.0	0.00	0.02	0.32	3.94	57.98	1116.63	21728.39	*aborted*											
C.1	0.00	0.02	0.32	2.88	57.07														
C.2	0.00	0.02	0.33	4.39	57.67	1123.14	21761.22	*aborted*											
C.3	0.00	0.02	0.33																
C.4	0.00	0.02	0.33	3.12	57.10														
C.5	0.00	0.02	0.39	4.27	75.20	1542.41													
Java.0	0.00	0.00	0.00	0.00	0.00	0.01	0.26	1.76	25.73	407.68	7121.46	*aborted*							
Java.1	0.00	0.00	0.00	0.00	0.01	0.03	0.25	2.46	25.28	410.42	7156.42	*aborted*							
Java.2	0.00																		
Java.3	0.00	0.01	0.00	0.01	0.01	0.03	0.26	2.36	24.40	407.71	7127.93								
Java.4	0.00	0.00	0.00	0.01	0.01	0.03	0.26	1.64	24.41										
Java.5	0.00	0.00	0.00	0.01	0.01	0.03	0.27												

Table A.4: Computation time (s) of AMBER (default) on medium and large grammars

Grammar	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HTML.0	0.00	0.00	0.00	0.00	0.03	0.24	1.53	7.26	60.48	506.88	4315.23	37321.08	*aborted*		
SQL.0	0.00	0.00	0.00	0.01	0.00	0.01	0.01	0.44	1.66	8.91	74.00	599.28	5002.53	41467.48	*aborted*
SQL.1	0.00	0.00	0.00	0.00	0.01	0.01	0.04	0.28	1.97						
SQL.2	0.00	0.01	0.00	0.00	0.00	0.00	0.04								
SQL.3	0.00	0.00	0.00	0.01	0.01	0.01									
SQL.4	0.00	0.00	0.00	0.01	0.01	0.01	0.04	0.26	1.20						
SQL.5	0.00	0.00	0.01	0.00	0.01	0.01	0.04	0.28	1.72						
Pascal.0	0.00	0.00	0.00	0.00	0.00	0.03	0.25	1.73	13.97	179.32	2585.31	38923.78	*aborted*		
Pascal.1	0.00	0.00	0.01	0.00	0.00	0.03	0.25	1.28	14.46						
Pascal.2	0.00	0.00	0.00	0.00	0.01	0.03	0.25								
Pascal.3	0.00	0.00	0.00	0.00	0.00	0.04	0.25	1.77	13.51	180.80	2590.55				
Pascal.4	0.00	0.00	0.00	0.00	0.00	0.03	0.25	1.15							
Pascal.5	0.00	0.00	0.00	0.00	0.00	0.03	0.25	1.27							
C.0	0.00	0.03	0.70	12.75	321.62	8505.67	*aborted*								
C.1	0.00	0.03	0.55	12.38	321.74										
C.2	0.00	0.04	0.96	12.91	322.60	8448.39	*aborted*								
C.3	0.00	0.04	0.52												
C.4	0.00	0.04	0.44	11.99	320.83										
C.5	0.00	0.04	0.54	13.98	388.67	10622.85									
Java.0	0.00	0.00	0.01	0.06	0.39	2.64	38.01	1026.78	31127.91	*aborted*					
Java.1	0.00	0.01	0.02	0.06	0.98	3.11	38.82	1033.68	30928.09	*aborted*					
Java.2	0.10														
Java.3	0.00	0.00	0.37	0.06	0.38	3.09	37.57	1040.76	31002.34	*aborted*					
Java.4	0.01	0.01	0.02	0.06	0.17	2.10	38.18	1037.37	31194.74						
Java.5	0.01	0.01	0.02	0.06	0.39										

Table A.5: Computation time (s) of AMBER (ellipsis) on medium and large grammars

Length	HTML.0	SQL.0	Pascal.0	C.0	Java.0
1	-	-	-	-	-
2	-	-	-	-	-
3	-	-	-	-	-
4	-	-	-	6252	-
5	-	-	-	6252	-
6	-	-	-	6252	-
7	-	-	-	6252	-
8	-	-	-		6520
9	-	-	-		6524
10	-	-	6192		6524
11	5992	6060	6192		6520
12	5992	6064	6192		
13	5996	6064	6192		
14	5996	6060	6204		
15	5996	6060			
16	5992	6064			
17	5996				
18	5996				

Table A.6: Memory usage (KB) of AMBER (default) on unambiguous medium and large grammars

Length	HTML.0	SQL.0	Pascal.0	C.0	Java.0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	6252	0
5	0	0	0	6256	0
6	0	0	0	6256	6524
7	5996	0	0		6520
8	5992	0	6192		6520
9	5992	6064	6196		6524
10	5996	6064	6196		
11	5996	6060	6192		
12	5996	6064	6204		
13		6060			
14		6072			

Table A.7: Memory usage (KB) of AMBER (ellipsis) on unambiguous medium and large grammars

A.3 Termination

Grammar	Ambig.	Default		Ellipsis		Fastest
		Time (s)	Max length	Time (s)	Max length	
HTML.0	no	> 15 h.	18	> 15 h.	12	-
SQL.0	no	> 15 h.	16	> 15 h.	13	-
SQL.1	yes	3478.18	15	2.31	9	Ellipsis
SQL.2	yes	0.00	7	0.05	7	Default
SQL.3	yes	0.02	6	0.03	6	Default
SQL.4	yes	0.19	9	1.53	9	Default
SQL.5	yes	4.28	11	2.07	9	Ellipsis
Pascal.0	no	> 15 h.	13	> 15 h.	11	-
Pascal.1	yes	0.63	9	16.03	9	Default
Pascal.2	yes	0.03	7	0.29	7	Default
Pascal.3	yes	53.11	11	2786.92	11	Default
Pascal.4	yes	0.11	8	1.43	8	Default
Pascal.5	yes	0.12	8	1.55	8	Default
C.0	no	> 15 h.	7	> 15 h.	6	-
C.1	yes	60.29	5	334.70	5	Default
C.2	yes	> 15 h.	7	> 15 h.	6	-
C.3	yes	0.35	3	0.56	3	Default
C.4	yes	60.57	5	333.30	5	Default
C.5	yes	1622.29	6	11026.08	6	Default
Java.0	no	> 15 h.	11	> 15 h.	9	-
Java.1	yes	> 15 h.	11	> 15 h.	9	-
Java.2	yes	0.00	1	0.00	1	-
Java.3	yes	7562.72	11	> 15 h.	9	Default
Java.4	yes	26.36	9	32272.65	9	Default
Java.5	yes	0.32	7	0.48	5	Default

Table A.8: Termination time (s) of AMBER on medium and large grammars

Appendix B

Results: MSTA

B.1 Accuracy

Grammar	Report
1	not LR(k) for $k < 12$
2	not LR(k) for $k < 12$
3	not LR(k) for $k < 15$
4	not LR(k) for $k < 51$
5	not LR(k) for $k < 51$
6	not LR(k) for $k < 32$
10	not LR(k) for $k < 13$
11	not LR(k) for $k < 10$
12	not LR(k) for $k < 9$
13	not LR(k) for $k < 10$
14	not LR(k) for $k < 11$
18	not LR(k) for $k < 27$
21	not LR(k) for $k < 10$
22	not LR(k) for $k < 12$
23	not LR(k) for $k < 25$
25	not LR(k) for $k < 12$
27	not LR(k) for $k < 16$
29	not LR(k) for $k < 9$
30	not LR(k) for $k < 13$
31	not LR(k) for $k < 10$
32	not LR(k) for $k < 9$
33	not LR(k) for $k < 51$
34	not LR(k) for $k < 18$
38	not LR(k) for $k < 12$

Grammar	Report
39	not LR(k) for $k < 14$
41	not LR(k) for $k < 51$
42	not LR(k) for $k < 39$
43	not LR(k) for $k < 11$
44	not LR(k) for $k < 17$
46	not LR(k) for $k < 51$
48	not LR(k) for $k < 15$
49	not LR(k) for $k < 11$
50	not LR(k) for $k < 17$
51	not LR(k) for $k < 51$
52	not LR(k) for $k < 51$
53	not LR(k) for $k < 51$
55	not LR(k) for $k < 11$
56	not LR(k) for $k < 51$
59	not LR(k) for $k < 51$
60	not LR(k) for $k < 32$
61	not LR(k) for $k < 51$
62	not LR(k) for $k < 49$
63	not LR(k) for $k < 13$
64	not LR(k) for $k < 10$
66	not LR(k) for $k < 51$
67	not LR(k) for $k < 10$
68	not LR(k) for $k < 9$
84	not LR(k) for $k < 51$

Table B.1: Results of MSTA on ambiguous small grammars

Grammar	Report
7	LR(1)
8	LR(1)
9	LR(1)
15	not LR(k) for $k < 14$
16	LR(1)
17	LR(1)
19	LR(1)
20	LR(1)
24	LR(1)
26	LR(1)
28	LR(1)
35	LR(1)
36	LR(1)
37	LR(1)
40	LR(1)
45	LR(3)
47	not LR(k) for $k < 24$
54	LR(1)

Grammar	Report
57	not LR(k) for $k < 51$
58	LR(1)
65	LR(1)
69	not LR(k) for $k < 11$
70	LR(1)
71	LR(1)
72	LR(1)
73	not LR(k) for $k < 13$
74	LR(1)
75	LR(1)
76	not LR(k) for $k < 7$
77	not LR(k) for $k < 11$
78	LR(1)
79	LR(1)
80	not LR(k) for $k < 51$
81	not LR(k) for $k < 14$
82	LR(1)
83	LR(1)

Table B.2: Results of MSTA on unambiguous small grammars

B.2 Performance

Grammar	1	2	3	4	5	6	7
HTML.0	0.02						
SQL.0	0.06						
SQL.1	0.07	0.08	2.38	23.13	1223.62	25817.44	*aborted*
SQL.2	0.12	0.08	1.59	25.09	1323.08	28553.91	*aborted*
SQL.3	0.04	0.08	2.08	38.05	2346.33	*aborted*	
SQL.4	0.03	0.09	1.77	23.00	1204.38	26046.56	*aborted*
SQL.5	0.05	0.09	3.23	82.04	7313.19	*aborted*	
Pascal.0	0.13						
Pascal.1	0.12	8.45	9335.29	*aborted*			
Pascal.2	0.14	7.63	9276.70	*aborted*			
Pascal.3	0.14	8.22	12465.26	*aborted*			
Pascal.4	0.13	6.47	10005.75	*aborted*			
Pascal.5	0.14	6.28	10356.06	*aborted*			
C.0	0.38						
C.1	0.37	18.25	9107.87	*aborted*			
C.2	0.45	20.59	14518.07	*aborted*			
C.3	0.55	17.95	9197.41	*aborted*			
C.4	0.39	17.92	9207.10	*aborted*			
C.5	0.44	18.17	9633.65	*aborted*			
Java.0	1.14						
Java.1	1.31	173.11	*aborted*				
Java.2	1.20	107.98	*aborted*				
Java.3	1.18	91.91	*aborted*				
Java.4	1.06	109.80	*aborted*				
Java.5	1.19	91.95	*aborted*				

Table B.3: Computation time (s) of MSTA on medium and large grammars

Grammar	1	2	3	4	5	6
HTML.0	-					
SQL.0	-					
SQL.1	-	-	2540	3132	6304	29356
SQL.2	-	-	2536	3140	6440	32024
SQL.3	-	-	2404	3240	8004	
SQL.4	-	-	2540	3128	6308	29004
SQL.5	-	-	2800	4180	11752	
Pascal.0	-					
Pascal.1	-	7460	112888			
Pascal.2	-	7752	117056			
Pascal.3	-	8760	154608			
Pascal.4	-	7328	112668			
Pascal.5	-	7328	113784			
C.0	-					
C.1	-	6108	56712			
C.2	-	6384	59936			
C.3	-	6116	56600			
C.4	-	6124	56716			
C.5	-	6196	59096			
Java.0	4304					
Java.1	3980	29740				
Java.2	3708	17176				
Java.3	4104	15312				
Java.4	4472	24412				
Java.5	3976	15172				

Table B.4: Memory consumption (KB) of MSTA on medium and large grammars

B.3 Termination

Grammar	Ambig.	Default	
		Time (s)	Max k
HTML.0	no	0.02	1
SQL.0	no	0.06	1
SQL.1	yes	$>$ 15 h.	6
SQL.2	yes	$>$ 15 h.	6
SQL.3	yes	$>$ 15 h.	6
SQL.4	yes	$>$ 15 h.	6
SQL.5	yes	$>$ 15 h.	6
Pascal.0	no	0.13	1
Pascal.1	yes	$>$ 15 h.	3
Pascal.2	yes	$>$ 15 h.	3
Pascal.3	yes	$>$ 15 h.	3
Pascal.4	yes	$>$ 15 h.	3
Pascal.5	yes	$>$ 15 h.	3
C.0	no	0.38	1
C.1	yes	$>$ 15 h.	3
C.2	yes	$>$ 15 h.	3
C.3	yes	$>$ 15 h.	3
C.4	yes	$>$ 15 h.	3
C.5	yes	$>$ 15 h.	3
Java.0	no	1.14	1
Java.1	yes	$>$ 15 h.	2
Java.2	yes	$>$ 15 h.	2
Java.3	yes	$>$ 15 h.	2
Java.4	yes	$>$ 15 h.	2
Java.5	yes	$>$ 15 h.	2

Table B.5: Termination time (s) of MSTA on medium and large grammars

Appendix C

Results: Schmitz' method

C.1 Accuracy

Grammar	# A	LR(0)	SLR(1)	LR(1)
1	3	4	4	4
2	3	4	4	4
3	2	1	1	1
4	1	1	1	1
5	3	3	3	3
6	2	1	1	1
10	2	6	6	5
11	5	5	5	2
12	4	9	9	9
13	2	2	2	2
14	6	6	6	5
18	1	1	1	1
21	6	20	12	10
22	5	8	8	8
23	1	2	2	2
25	2	3	3	3
27	2	2	2	2
29	1	1	1	1
30	2	6	6	5
31	5	5	5	2
32	4	9	9	9
33	2	1	1	1
34	8	6	6	6
38	6	9	9	9

Grammar	# A	LR(0)	SLR(1)	LR(1)
39	1	2	2	2
41	1	1	1	1
42	4	4	4	4
43	1	3	3	3
44	3	6	6	6
46	3	2	2	2
48	1	1	1	1
49	5	9	9	9
50	1	4	4	3
51	1	1	1	1
52	1	1	1	1
53	1	2	2	2
55	1	1	1	1
56	1	1	1	1
59	1	8	1	1
60	3	3	3	3
61	1	5	1	1
62	1	5	5	5
63	3	3	3	3
64	2	14	14	14
66	1	1	1	1
67	5	15	14	14
68	6	18	15	15
84	1	1	1	1

Table C.1: Number of potential ambiguities reported by Schmitz method on ambiguous small grammars

Grammar	LR(0)	SLR(1)	LR(1)
7	-	-	-
8	-	-	-
9	-	-	-
15	1	1	-
16	-	-	-
17	-	-	-
19	-	-	-
20	-	-	-
24	-	-	-
26	-	-	-
28	-	-	-
35	-	-	-
36	-	-	-
37	-	-	-
40	-	-	-
45	-	-	-
47	-	-	-
54	-	-	-

Grammar	LR(0)	SLR(1)	LR(1)
57	1	1	1
58	6	-	-
65	-	-	-
69	4	4	2
70	5	-	-
71	5	-	-
72	1	1	-
73	8	8	3
74	8	3	-
75	-	-	-
76	14	13	12
77	1	1	-
78	-	-	-
79	1	1	-
80	-	-	-
81	10	9	8
82	1	1	-
83	-	-	-

Table C.2: Number of potential ambiguities reported by Schmitz method on unambiguous small grammars

Grammar	Amb.	LR(0)	SLR(1)	LR(1)
HTML.0	no	9	3	0
SQL.0	no	0	0	0
SQL.1	yes	1	1	1
SQL.2	yes	3	3	3
SQL.3	yes	4	4	4
SQL.4	yes	1	1	1
SQL.5	yes	3	3	3
Pascal.0	no	228	36	0
Pascal.1	yes	236	38	2
Pascal.2	yes	232	38	2
Pascal.3	yes	235	37	1
Pascal.4	yes	229	37	1
Pascal.5	yes	229	37	3
C.0	no	3	3	0
C.1	yes	4	4	1
C.2	yes	4	4	1
C.3	yes	62	59	58
C.4	yes	56	56	55
C.5	yes	54	54	*Mem exc.*
Java.0	no	160	129	0
Java.1	yes	163	132	2
Java.2	yes	273	236	*Mem exc.*
Java.3	yes	163	132	2
Java.4	yes	169	138	*Mem exc.*
Java.5	yes	165	134	5

Table C.3: Number of potential ambiguities reported by Schmitz method on medium and large grammars

C.2 Performance

	LR(0)		SLR(1)		LR(1)	
Grammar	Time (s)	Mem (KB)	Time (s)	Mem (KB)	Time (s)	Mem (KB)
HTML.0	0.02	-	0.00	-	0.04	-
SQL.0	0.05	-	0.01	-	0.28	-
SQL.1	0.02	-	0.02	-	0.31	-
SQL.2	0.09	-	0.01	-	0.32	-
SQL.3	0.02	-	0.01	-	0.32	-
SQL.4	0.05	-	0.01	-	0.29	-
SQL.5	0.03	-	0.01	-	0.30	-
Pascal.0	0.25	-	0.14	-	2.97	56748
Pascal.1	0.25	-	0.15	-	3.35	76948
Pascal.2	0.28	-	0.17	-	3.43	81056
Pascal.3	0.28	-	0.16	-	2.52	64028
Pascal.4	0.28	-	0.16	-	3.29	91872
Pascal.5	0.28	-	0.16	-	2.74	65868
C.0	0.09	-	0.20	-	177.66	1564392
C.1	0.15	-	0.13	-	182.84	1582620
C.2	0.17	-	0.19	-	163.57	1603228
C.3	0.54	-	0.45	-	2165.17	3083368
C.4	0.55	-	0.21	-	2461.39	3092424
C.5	0.56	-	0.45	-	*Mem exc.*	>3GB
Java.0	1.47	64560	1.19	59664	2166.30	2579848
Java.1	2.19	4072	1.03	-	4796.19	2609176
Java.2	0.99	-	0.49	-	*Mem exc.*	>3GB
Java.3	1.25	68180	0.90	-	1849.14	2645232
Java.4	1.22	67372	0.92	-	*Mem exc.*	>3GB
Java.5	1.23	65904	0.90	-	2153.50	2585276

Table C.4: Performance of Schmitz method on medium and large grammars

Appendix D

Grammar collections

D.1 Small grammars

The following list contains the collection of small grammars that was used for the accuracy measurements. Grammar 1 - 50 were taken from Jampana's Master thesis [17]. Of all other grammars the author is referenced if it was taken from literature.

The following modifications were made to the grammars of Jampana:

- Grammars 3, 13, 23, 33, 41 and 43 are ambiguous, but were originally marked unambiguous.
- Grammar 20: removed duplicate production rule `C->c`
- Grammar 25: removed duplicate production rule `A->AS`
- Grammar 25: changed `S->b` to `B->b`

Grammar 1:
Ambiguous
```
S->AB
S->a
A->SB
A->b
B->BA
B->a
```

Grammar 2:
Ambiguous
```
E->E+E
E->E*E
E->(E)
E->a
```

Grammar 3:
Ambiguous
```
E->E*T
E->T
T->T*F
T->F
F->(E)
F->a
```

Grammar 4:
Ambiguous (inherently)
```
S->AB
S->C
A->aAb
A->ab
B->cBd
B->cd
C->aCd
C->aDd
D->bDc
D->bc
```

Grammar 5:
Ambiguous
```
S->AB
S->CD
S->EF
A->a
B->b
C->a
D->b
E->a
F->b
```

Grammar 6:

Ambiguous dangling else
S->ictS
S->ictSeS
S->a

Grammar 7:
Unambiguous if-else
S->M
S->U
M->ictMeM
M->a
U->ictS
U->ictMeU

Grammar 8:
Unambiguous
S->aSa
S->bSb
S->c

Grammar 9:
Unambiguous
S->AB
S->ASB
A->a
B->b

Grammar 10:
Ambiguous
S->AB
S->CA
A->a
B->BC
B->AB
C->aB
C->b
B->b

Grammar 11:
Ambiguous
S->AB
S->BC
A->BAD
A->a
B->CC
B->bD
C->AB
C->c
D->d

Grammar 12:
Ambiguous
S->bA
S->aB
A->bAA
A->aS
A->a
B->bBB
B->b
B->SB

Grammar 13:
Ambiguous
S->AA
S->a
A->SS
A->b

Grammar 14:
Ambiguous
S->AB
S->BC
A->BA
A->a
B->CC
B->b
C->AB
C->a

Grammar 15:
Unambiguous
S->ABCD
A->CS
B->bD
D->SB
A->a
B->b
C->c
D->d

Grammar 16:
Unambiguous
S->N1V1
S->N2V2
N1->DN3S1
N2->DN4
V1->V3N2
V2->V4S1
V2->stink
S1->CS
D->the
N3->fact
N4->cats
N4->dogs
C->that
V4->think
V3->amazes
V3->bothers

Grammar 17:
Unambiguous (same as 16 but with single letter terminals)
S->N1V1
S->N2V2
N1->DN3S1
N2->DN4
V1->V3N2
V2->V4S1
V2->s
S1->CS
D->t
N3->f
N4->c
N4->d
C->h
V4->i
V3->a
V3->b

Grammar 18:
Ambiguous
S->PQ
P->ROT
P->a
R->MP
O->a
O->ab
T->b
T->bb
M->a
Q->CeD

C->a
C->ab
D->d
D->ed

Grammar 19:
Unambiguous
S->ABD
S->ABC
A->AE
A->a
B->SE
B->b
D->d
C->c
E->dc

Grammar 20:
Unambiguous
S->BC
S->AB
A->CF
C->c
F->ged
B->ab
B->aC

Grammar 21:
Ambiguous
S->U
S->V
U->TaU
U->TaT
V->TbV
V->TbT
T->aTbT
T->bTaT
T->

Grammar 22:
Ambiguous
S->AA
A->AAA
A->bA
A->Ab
A->a

Grammar 23:
Ambiguous
S->ACA
A->aAa
A->B
A->C
B->bB
B->b
C->cC
C->c

Grammar 24:
Unambiguous
T->ABC
T->ABD
A->AD
A->AC
A->a
B->AB
B->b
C->c
D->d

Grammar 25:
Ambiguous
A->AS
S->AB
S->BB
B->b
A->bA
A->a

Grammar 26:
Unambiguous
S->AB
B->bb
B->bB
A->a
A->aAb

Grammar 27:
Ambiguous
A->a
A->B
A->CA
B->bD
B->b
D->d
D->dD
D->ad
C->bc
C->c
C->CC

Grammar 28:
Unambiguous
Z->aXY
Z->bXZ
Z->z
X->aY
X->az
X->y
Y->y

Grammar 29:
Ambiguous
S->NVNDJ
N->a
N->h
N->PJN
N->PN
V->f
V->e
P->f
P->p
D->r
D->v
J->b
J->g
J->PJ

Grammar 30:
Ambiguous
S->AB
S->CA
A->a
B->BC
B->AB
C->aB
C->b
B->b

Grammar 31:
Ambiguous

```
S->AB
S->BC
A->BAD
A->a
B->CC
B->bD
C->AB
C->c
D->d
```

Grammar 32: Ambiguous

```
S->bA
S->aB
A->bAA
A->aS
A->a
A->bBB
B->b
B->SB
```

Grammar 33: Ambiguous

```
S->T
S->a
T->A
T->b
T->T
A->ab
A->aab
```

Grammar 34: Ambiguous

```
S->t
S->e
S->he
S->S
S->eH
S->eHS
S->H
H->hH
H->h
H->ht
```

Grammar 35: Unambiguous

```
S->WCT
W->while
C->bconditionb
T->bRb
R->statement
R->statementR
```

Grammar 36: Unambiguous

```
S->WCT
W->w
C->bcb
T->bRb
R->s
R->sR
```

Grammar 37: Unambiguous

```
S->wbcbbRb
R->s
R->sR
```

Grammar 38: Ambiguous

```
S->SS
S->AS
S->a
S->b
A->AA
A->AS
A->a
```

Grammar 39: Ambiguous

```
S->b
S->Tb
S->TQ
T->baT
T->caT
T->aT
T->ba
T->ca
T->a
Q->bc
Q->bcQ
Q->caQ
Q->ca
Q->a
Q->aQ
```

Grammar 40: Unambiguous

```
T->rXr
T->rXrT
X->text
X->C
C->dtextd
C->dtextdC
```

Grammar 41: Ambiguous

```
S->UVPO
S->UVCPO
U->house
U->houseflies
V->flies
V->like
C->like
P->a
P->the
O->banana
```

Grammar 42: Ambiguous

```
S->MN
S->PQ
M->aM
M->Mc
M->b
N->Nc
N->bN
N->c
P->Pd
P->cP
P->d
Q->ad
```

Grammar 43: Ambiguous

```
A->BDE
B->cA
B->c
B->a
D->cD
D->d
```

```
D->aB
E->e
E->de
E->ce
```

Grammar 44:
Ambiguous
```
S->SAB
S->ASB
S->b
A->ab
A->Ba
B->b
B->bB
```

Grammar 45:
Unambiguous
```
L->AND
L->GA
A->a
A->aA
A->ab
N->ab
D->ba
D->Da
G->bG
G->baG
G->ba
```

Grammar 46:
Ambiguous
```
S->NVN
S->NVNVingN
N->dogs
N->NVingN
V->eat
```

Grammar 47:
Unambiguous
```
A->SB
A->AS
S->ab
B->b
B->bB
B->SB
```

Grammar 48:
Ambiguous
```
P->PRQ
P->p
R->pr
R->r
R->rR
Q->q
Q->qQ
Q->rq
```

Grammar 49:
Ambiguous
```
S->ABSB
S->ASB
S->a
A->aA
A->a
B->ab
B->AB
B->b
B->bB
```

Grammar 50:
Ambiguous
```
S->AC
S->BA
A->Ba
A->aB
A->a
C->CB
C->c
B->Bc
B->b
B->bc
```

Grammar 51:
Ambiguous
```
S->E
E->E+E
E->a
```

Grammar 52:
Ambiguous
```
S->A
S->B
A->a
B->a
```

Grammar 53:
Ambiguous [13] system 1
```
A->a
A->b
A->aA
A->Ab
```

Grammar 54:
Unambiguous [13] system 2
```
A->B
A->C
A->BC
B->a
B->aB
C->b
C->bC
```

Grammar 55:
Ambiguous [2] exercise 2.6.27
```
S->aSbSc
S->aSb
S->bSc
S->d
```

Grammar 56:
Ambiguous [2] example 6.11
```
S->A
S->B
A->aA
A->a
B->Ba
B->a
```

Grammar 57:
Unambiguous but not LR
```
S->aSa
S->a
```

Grammar 58:
Unambiguous produces empty string
```
S->AB
A->aA
A->
B->bB
B->
```

Grammar 59:
Ambiguous produces empty string

```
S->A
S->B
A->aA
A->
B->bB
B->
```

Grammar 60:
Ambiguous [4] grammar C.3

```
S->N1 V
S->S P
N1->N2
N1->a N2
N1->N1 P
N2->i
N2->man
N2->home
P->at N1
V->see N1
```

Grammar 61:
Ambiguous [27] grammar 2

```
S->A
A->B
A->BC
B->b
C->c
C->
```

Grammar 62:
Ambiguous SDF layout conflict

```
S->aLALa
L-> L
L->
A->Ab
A->
```

Grammar 63:
Ambiguous [7]

```
A->Ab
A->BA
B->BA
A->a
B->b
```

Grammar 64:
Ambiguous [7]

```
S->aB
S->bA
A->a
A->aS
A->baA
A->bbAAa
B->b
B->bS
B->aBB
```

Grammar 65:
Unambiguous [3]

```
S->I
S->A
I->ifEthenS
A->D=E
E->D=D
D->if
D->then
```

Grammar 66:
Ambiguous C pointer declaration vs. multiplication expression

```
S->E
S->D
E->I*I
D->T*I
I->a
T->a
```

Grammar 67:
Ambiguous [10] G1

```
S->(S)
S->.S
S->S.
S->SS
S->
```

Grammar 68:
Ambiguous [10] G2

```
S->(P)
S->.S
S->S.
S->SS
S->
P->(P)
P->S
```

Grammar 69:
Unambiguous [10] G3

```
S->(S)
S->.L
S->R.
S->LS
L->(S)
L->.L
R->R.
R->
```

Grammar 70:
Unambiguous [10] G4

```
S->.S
S->T
S->
T->T.
T->(S)
T->T(S)
```

Grammar 71:
Unambiguous [10] G5

```
S->.S
S->(S)S
S->
```

Grammar 72:
Unambiguous [10] G6

```
S->LS
S->L
L->(F)
L->.
F->(F)
F->LS
```

Grammar 73:
Unambiguous [10] G7

```
S->(P)
S->.L
S->R.
```

```
S->LS
L->(P)
L->.L
R->R.
R->
P->(P)
P->(N)
N->.L
N->R.
N->LS
```

Grammar 74:
Unambiguous [10] G8

```
S->.S
S->T
S->
T->T.
T->(P)
T->T(P)
P->(P)
P->(N)
N->.S
N->T.
N->T(P)
```

Grammar 75:
Unambiguous [6] example 18

```
S->AA
A->xAx
A->y
```

Grammar 76:
Unambiguous [6] example 12

```
R->cRg
R->gRc
R->aRu
R->uRa
R->gRu
R->uRg
R->
```

Grammar 77:
Unambiguous [6] example 13 Voss-Light

```
P->(P)
P->(O)
O->LP
O->PR
O->SPS
O->H
L->.L
L->.
R->.R
R->.
S->.S
S->.
H->.H
H->...
```

Grammar 78:
Unambiguous [12] Grammar 2.8 SLR(1), not LR(0)

```
S->aBc
S->aDd
B->b
D->b
```

Grammar 79:
Unambiguous [12] Grammar 2.9 LR(1), not SLR(1)

```
S->aBc
S->aDd
S->Bd
B->b
D->b
```

Grammar 80:
Unambiguous [23] G5

```
S->AC
S->BCb
A->a
B->a
C->cCb
C->cb
```

Grammar 81:
Unambiguous [23] G6

```
S->aSa
S->bSb
S->a
S->b
S->
```

Grammar 82:
Unambiguous [1] grammar 4.20 NOT SLR

```
S->L=R
S->R
L->*R
L->a
R->L
```

Grammar 83:
Unambiguous (same as 80 without B and C for a)

```
S->aC
S->aCb
C->cCb
C->cb
```

Grammar 84:
Ambiguous [6] Horizontal ambiguity example

```
S->AB
A->xa
A->x
B->ay
B->y
```

D.2 Medium and large grammars

The medium and large grammars were taken from the Internet in Yacc format:

Medium grammars

- Grammar HTML
  size: 29 production rules
  source: `http://www.graphviz.org/pub/graphviz/CURRENT/graphviz-2.13.20070308.0540.tar.gz`
  modification: removed `string : string T_string`
- Grammar SQL
  size: 79 production rules
  source: `grass-6.2.0RC1/lib/db/sqlp/yac.y` from `http://grass.itc.it/grass62/source/grass-6.2.0RC1.tar.gz`
- Grammar Pascal
  size: 176 production rules
  source: `ftp://ftp.iecc.com/pub/file/pascal-grammar`
  modification: removed `statement : IF expression THEN statement`

Large grammars

- Grammar C
  size: 212 production rules
  source: `ftp://ftp.iecc.com/pub/file/c-grammar.gz`
  modification: removed `selection_statement : IF '(' expr ')' statement`
- Grammar Java
  size: 349 production rules
  source: `http://gcc.gnu.org/cgi-bin/cvsweb.cgi/gcc/gcc/java/parse.y?rev=1.475`

All conflicts were removed to disambiguate them, these versions are labeled *grammar*.0. Of each grammar 5 ambiguous versions were made, labeled *grammar*.1 to *grammar*.5. The following modifications were made:

- Grammar SQL.1: changed `y_sub_condition : y_sub_condition OR y_sub_condition2` into `y_sub_condition : y_sub_condition OR y_sub_condition`
- Grammar SQL.2: added `y_expression : '(' y_expression ')'`
- Grammar SQL.3: added `y_value : NULL_VALUE`
- Grammar SQL.4: added `y_columndef : NAME INT ',' NAME INT`
- Grammar SQL.5: added `y_sub_condition2 : y_boolean ORDER BY y_order`

- Grammar Pascal.1: added `actuals_list : actual_param ":" expression`

- Grammar Pascal.2: added `term : NOT term`
- Grammar Pascal.3: added `statement : IF expression THEN statement`
- Grammar Pascal.4: added `add_op : '<'`
- Grammar Pascal.5: added `mult_op : '.'`

- Grammar C.1: added `unary_operator : '+' '+'`
- Grammar C.2: added `selection_statement : IF '(' expr ')' statement`
- Grammar C.3: added `parameter_type_list : //empty`
- Grammar C.4: added `assignment_operator : ','`
- Grammar C.5: added `type_specifier : GOTO`

- Java.1: added `cast_expression : OP_TK primitive_type CP_TK unary_expression C_TK`
- Java.2: added `empty_statement : //empty`
- Java.3: changed `if_then_else_statement : IF_TK OP_TK expression CP_TK statement_nsi ELSE_TK statement` into `if_then_else_statement : IF_TK OP_TK expression CP_TK statement ELSE_TK statement`
- Java.4: added `unary_expression : trap_overflow_corner_case OSB_TK CSB_TK`
- Java.5: added `primitive_type : VOID_TK`