Ambiguity Detection for Programming Language Grammars

Bas Basten

Bas Basten

Detection for Programming Language Grammars

Ambiguity

Detection for Programming Language Grammars

Bas Basten

Ambiguity Detection
for
Programming Language Grammars

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor promoties ingestelde
commissie, in het openbaar te verdedigen in de Agnietenkapel
op donderdag 15 december 2011, te 14:00 uur
door

Hendrikus Joseph Sebastiaan Basten

geboren te Boxmeer

CWI Centrum Wiskunde & Informatica

INSTITUUT VOOR PROGRAMMATUURKUNDE EN ALGORITMIEK ipa

The work in this thesis has been carried out at Centrum Wiskunde & Informatica (CWI) in Amsterdam, under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Contents

Acknowledgements

This thesis is the result of my four-year PhD project at the Centrum Wiskunde & Informatica in Amsterdam. During this time I met a lot of friendly people. Many of them deserve my thanks for making my PhD project a productive and enjoyable one.

First of all, I would like to thank my promotor Paul Klint for offering me a PhD position at CWI, and for supporting and guiding me throughout the project. I have learned a lot from all his comments and advice. Furthermore, Paul's liberal way of supervising creates a very positive, productive and friendly atmosphere to work in.

Second, I wish to thank my co-promotor Jurgen Vinju. He was always enthusiastic about my ideas, giving me new energy when I felt I was stuck. I think we were a good paper-writing team. Furthermore, it was very enjoyable to visit various conferences together, where Jurgen always made an effort to introduce me to many interesting people.

I also would like to specially thank Tijs van der Storm, my officemate for the last two years, close neighbour, and fellow guitar student. Tijs was always very sociable, both inside and outside of CWI. I can heartly recommend him as a city guide to Amsterdam, for finding the best places to eat and drink, at every hour of the day. It's a pity we didn't get the chance to do more research together.

Many thanks as well to all my other colleagues from SEN1: Anya Helene Bagge, for many great dinners and a nice visit to Bergen, Jeroen van den Bos, with whom it is always very enjoyable to converse and philosophize, Rob Economopoulos, for sharing an office and drinking beers[1], Jan van Eijck, Paul Griffioen, Jan Heering, Mark Hills, Anastasia Izmaylova, Davy Landman, Arnold Lankamp, Bert Lisser, Pieter Olivier, Atze van der Ploeg, Riemer van Rozen, Floor Sietsma for a nice holiday on Cyprus, Sunil Simon, Yanjing Wang and Vadim Zaytsev. My time at CWI wouldn't have been so much fun without you. For all of you who are still working on your PhDs, I wish you the best of luck.

To this list I should also add the following people from SEN3, with whom I shared some nice experiences as well: Behnaz Changizi, Stephanie Kemper, Young-Joo Moon, José Proença and Alexandra Silva.

There are also some people from outside of CWI who I wish to thank. Tiago Alves was

[1]no ambiguity intended

(and still is) a good friend and fellow PhD student. During dinner or beers we shared many thoughts about doing a PhD and other things that are important in life. I have very good memories about our holidays in Portugal. Thanks to Andreia Esteves too for all the good times.

I thank Dennis Dams for offering me a three-month visit to Alcatel-Lucent Bell Labs in Murray Hill, NJ. This was a very exciting and enriching time in my life, during which Dennis was a very friendly and open host. He helped me to find my way around in the local culture and surroundings, and let me share Thanksgiving and Christmas with his family.

Finally, I thank my parents, my sisters, and my friends for always being there for me. But most of all, I am sincerely grateful to my love Elise, for all her patience and support. She deserves more thanks than I can express in words.

Chapter 1

Introduction

"Thorns and roses grow on the same tree."
Turkish proverb

Context-free grammars are widely used in software engineering to define a broad range of languages. However, they have the undesirable property that they can be ambiguous, which, if unknown, can seriously hamper their use. Automated ambiguity detection is therefore a very welcome tool in grammar development environments or language workbenches. The goal of this thesis is to advance ambiguity detection to a level that is practical for use on grammars of real programming languages. We first investigate which criteria define the practical usability of an ambiguity detection method, and use them to evaluate existing methods. After that we present new approaches that improve upon the state of the art in terms of accuracy, performance, and report quality. We formally prove the correctness of our approaches and experimentally validate them on grammars of real programming languages. This chapter describes the motivation for this work and gives an overview of the thesis.

1.1 Introduction

Before presenting the research questions and contributions of this thesis, this section first gives a quick introduction into the field of ambiguity detection for context-free grammars.

1.1.1 Context-Free Grammars and Parsing

Context-free grammars form the basis of many language processing systems. They were originally designed by Chomsky [Cho56] for describing natural languages, but were soon

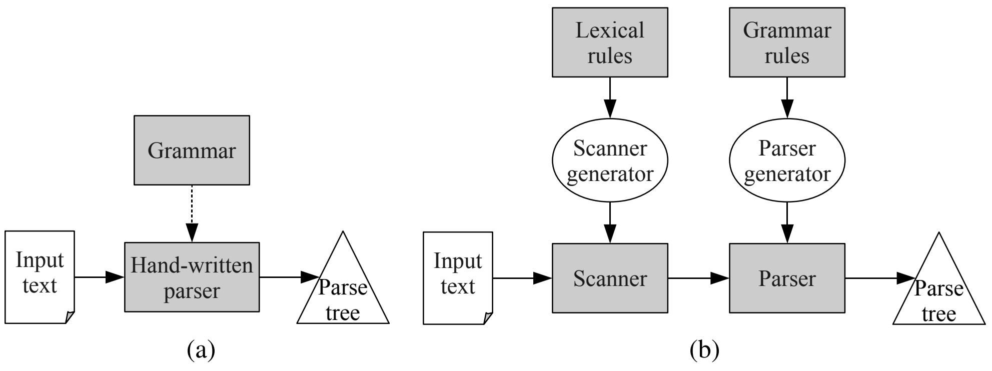


Figure 1.1: Two different parser architectures: hand-written parsers (a) are loosely based on a grammar, while traditionally generated parsers (b) are directly generated from a formal grammar definition. The generated parsers require a scanner that first subdivides the input text into a stream of tokens.

adopted in the area of computer languages. In 1959 John Backus described the syntax of the ALGOL 58 programming language in a context-free grammar formalism that would later be named BNF: Backus-Naur Form. At first BNF grammars were only used for documentation purposes, but shortly after people began to use them for the automatic generation of compilers. This was already the case for the ALGOL 60 language and its variants.

Nowadays, context-free grammars are used in much more software engineering activities, ranging from software construction to reverse engineering and reengineering. They play a central role in the development of compilers, interpreters, interactive development environments (IDEs), tools for source code analysis, source-to-source transformation, refactorings, etcetera, all for different types of languages like programming languages, data definition languages, specification languages, and domain specific languages [KLV05].

In most cases, language applications deal with source code that is formatted as plain text. However, they often also require the semantics of the code, which can be derived from its syntactic or grammatical structure. The process of recovering the grammatical structure of plain text code into a parse tree or abstract syntax tree is called *parsing*, and is executed by a *parser*. Parsers can be crafted by hand, but can also be automatically generated from a context-free grammar. The main advantage of the latter approach is that the language definition can be separated from the parsing algorithm. Context-free grammars are a much more declarative way of describing languages than the program code of a parser. They allow for easier development and maintenance of the generated parsers. Furthermore, a parser generator can be reused for different grammars, and individual grammars can be used for other purposes as well [KLV05].

Figure 1.1 shows the differences in architecture between a hand-written parser and a generated parser. Traditionally, generated parsers require a *scanner* to first subdivide their input text into a stream of tokens. These scanners are typically also generated, using another definition of the lexical syntax of a language. More recently developed *scannerless* parsers [SC89, vdBSVV02] do not require this separate pre-processing step, as is shown in

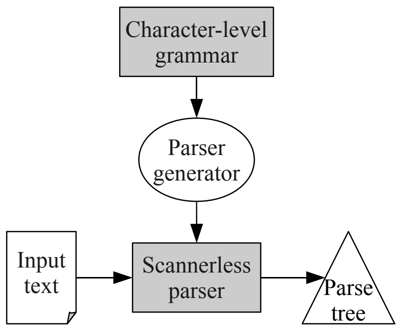


Figure 1.2: Architecture of a scannerless generalized parser. The parser requires no scanner, because it directly parses the characters of its input text from a character-level grammar.

Figure 1.2. Instead, they directly parse their input text using a *character-level grammar*, which includes both lexical and context-free definitions. This has the advantage that a language can be completely defined in a single formalism.

1.1.2 Ambiguity

Context-free grammars generate languages through recursion and embedding, which makes them a very suitable formalism for describing programming languages. Furthermore, there are many efficient parsing techniques available for them. However, context-free grammars have the undesirable property that they can be ambiguous.

A grammar is ambiguous if one or more sentences in its language have multiple distinct parse trees. The structure of a parse tree is often the basis for the semantic analysis of its sentence, so multiple parse trees might indicate multiple meanings. This is usually not intended, and can therefore be seen as a *grammar bug*. As an example, figure 1.3 shows an often occuring grammar bug: the ‘dangling-else’ ambiguity.

Using ambiguous grammars can lead to different kinds of problems during semantic analysis. For instance, a compiler could give a different interpretation to a piece of source code than the programmer that wrote it. This can lead to unintended behaviour of the compiled code. In other cases, a compiler might be unable to compile a piece of source code that perfectly fits the language specification, because it simply does not expect the ambiguity or its parser reports the ambiguity as a parse error. To avoid situations like this, it is important to find all sources of ambiguity in a grammar before it is being used.

1.1.3 Ambiguity Detection

Searching for ambiguities in a grammar by hand is a very complex and cumbersome job. Therefore, automated ambiguity detection is a very welcome tool in any grammar development environment. However, detecting the ambiguity of a grammar is undecidable in general [Can62, Flo62, CS63]. This implies that it is uncertain whether the ambiguity of a given grammar can

```
if (i==0) then
  if (j==0) then
    ...
  else
    ...
```

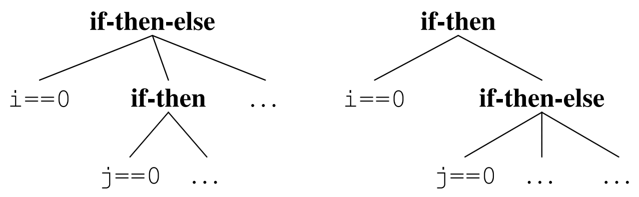


Figure 1.3: The 'dangling-else' ambiguity. This figure shows two possible parse trees of the code snippet on the left. In the left parse tree the else-statement is paired with the first if statement, in the right tree the else statement is paired with the second if statement. Depending on the parse tree that is chosen by the compiler, this piece of code will be executed differently.

be determined in a finite amount of time. Fortunately, this does not necessarily have to be a problem in practice.

Several *ambiguity detection methods* (ADMs) exist that approach the problem from different angles, all with their own strengths and weaknesses. Because of the undecidability of the problem there are general tradeoffs between accuracy and termination, and between accuracy and performance. The challenge for every ADM is to give the most accurate and understandable answer in the time available.

Existing ADMs can roughly be divided into two categories: *exhaustive* methods and *approximative* ones. Methods in the first category exhaustively search the language of a grammar for ambiguous sentences. There are various techniques available [Gor63, CU95, Sch01, AHL08, Jam05] for this so-called *sentence generation*. These methods are 100% accurate, but a problem is that they never terminate if the grammar's language is of infinite size, which usually is the case in practice. They do produce the most accurate and useful ambiguity reports, namely ambiguous sentences and their parse trees.

Methods in the approximative category sacrifice accuracy to be able to always finish in finite time. They search an approximation of the grammar or its language for possible ambiguity. The currently existing methods [Sch07b, BGM10] all apply conservative approximation to never miss ambiguities. However, they have the disadvantage that their ambiguity reports can contain false positives.

1.1.4 Goal of the Thesis

The current state of the art in ambiguity detection is unfortunately not yet sufficiently practical on grammars of realistic programming languages, especially those used for scannerless parsers. The goal of this thesis is therefore to advance ambiguity detection to a more practical level. We first investigate which criteria define the practical usability of ADMs, and use them to evaluate existing methods. After that we present new approaches that improve upon the state of the art in terms of accuracy, performance, and report quality. We formally prove the correctness of our approaches or experimentally validate them on grammars of real programming languages.

1.2 Motivation

This work is motivated by the fact that ambiguity poses problems with many different types of parsers. Depending on the parsing technique, these problems present themselves in different ways. In this section we discuss the problems of ambiguity for the most commonly used techniques, with a special focus on scannerless generalized parsing.

1.2.1 Deterministic Parsers

Although it might not seem so at first sight, ambiguity is a real problem for deterministic parsers like LL [Knu71], LR [Knu65] or LALR [DeR69]. The reason for this is that many parser implementations do not require their input grammars to exactly fit their required grammar class. Instead, they allow for *conflicts* in their internal representation. These conflicts mark points during parsing where the next parse action cannot be chosen deterministically. This can be a result from non-determinism in the grammar, but also from ambiguity.

The way these conflicts are solved during parsing is by choosing an action based on certain heuristics. This way the parser can always continue with constructing a valid parse tree. However, if the grammar is ambiguous there might be other parse trees for the same input text as well, and it is unclear whether or not the produced tree is the right one. This could for instance lead to the problem where a compiled program executes in a different way than its programmer intended. Since ambiguities remain hidden by the parser, it can take very long for these kinds of bugs to be noticed.

1.2.2 Backtracking Parsers

Backtracking parsers, like for instance Packrat parsing [For04] or LL(*) [PF11], suffer from the same problem. In case of a parse error, these parsers backtrack to the last encountered conflict, and continue with the next possible action. This process is repeated until they find the first parse tree that fits their input. Again, if the used grammar is ambiguous, it is hard to tell whether or not this is the parse tree that the grammar developer intended.

1.2.3 Generalized Parsers

Generalized parsing techniques, like Earley [Ear70], GLR [Tom85, NF91] and GLL [SJ10], but also parser combinators [Hut92], have the advantage that they do report ambiguity, because they always return all the valid parse trees of their input text. This set of parse trees is commonly refered to as a *parse forest*. However, if the parse forest returned by the parser contains an ambiguity that was not expected by the following analyses or transformations, the forest cannot be processed further. A compiler will thus not produce incorrect binaries, for example, but is simply unable to compile its input program.

1.2.4 Scannerless Generalized Parsers

A special type of generalized parsers that are particularly sensitive to ambiguity are scannerless generalized parsers. Instead of using a scanner, these parsers directly parse their input characters using a character-level grammar. Examples of character-level grammar formalisms are SDF2 [HHKR89, Vis97] and RASCAL [KvdSV11].

Among other advantages, character-level grammars can be used to describe languages that originally were not designed to be parsed using a generated scanner. This makes them very suitable for the reverse engineering of legacy languages, for instance those that do not apply keyword reservation (PL/I) or longest match (Pascal). However, to also be able to describe languages that do require these scanner heuristics, character-level grammars may be annotated with so-called disambiguation filters [KV94]. These filters allow for the removal of unwanted parse trees of ambiguous strings, without modifying the structure of the grammar. If applied right, they filter all but one parse tree of an ambiguous string, thus solving the ambiguity. Many of the typical scanner heuristics can be mimicked with disambiguation filters, for instance longest match using *follow restrictions*, or keyword reservation using *rejects*. This makes scannerless parsers suitable for parsing traditional deterministic languages, as well as non-deterministic (legacy) languages.

Unfortunately, the downside of not using a scanner is that character-level grammars are more “ambiguity-prone”. In cases where scanner-like heuristics are needed, lexical determinism needs to be declared manually with disambiguation filters. Especially for large languages it can be hard to check whether all lexical ambiguities are solved. Therefore, automated ambiguity detection would be especially helpful when developing character-level grammars.

1.3 Research Questions

In this thesis we investigate ambiguity detection methods for context-free grammars. We focus on traditional token-based grammars as well as character-level grammars. Our goal is to advance ambiguity detection to a level that is practical for use on grammars of real programming languages. The following sections describe the ideas we explore and the research questions that guide our investigation.

1.3.1 Measuring the Practical Usability of Ambiguity Detection

Despite the fact that ambiguity detection is undecidable in general, we would like to know whether approaches exist that can be useful in practice. For this we need a set of criteria to evaluate the practical usability of a given ambiguity detection method. These criteria enable us to compare different detection methods and evaluate the benefits of new or improved methods. This leads us to the first research question:

Research Question 1

How to assess the practical usability of ambiguity detection methods?

After we find such criteria, the next logical step is to evaluate existing methods. This will give insight into whether or not useable methods already exist, and may point at directions for improvements. The second question therefore is:

Research Question 2

What is the usability of the state-of-the-art in ambiguity detection?

These two questions are both addressed in Chapter 2. It starts by discussing a list of criteria for practical usability. Then the implementations of three ambiguity detection methods are tested on a set of benchmark grammars. Their scores on each of the criteria are measured and analyzed, and the methods are compared to each other.

1.3.2 Improving the Practical Usability of Ambiguity Detection

The criteria that arise from the first research question direct the areas in which we can improve the practical usefulness of ambiguity detection. From the five criteria that we defined in Chapter 2, we will mainly focus on the following three: performance, accuracy, and report quality. This leads us to the next research questions:

Research Question 3

How to improve the performance of ambiguity detection?

The goal of this question is to find faster searching techniques that require less memory.

Research Question 4

How to improve the accuracy of approximative ambiguity detection?

Approximative ADMs trade in accuracy so they can always terminate. The goal of this question is to find methods that produce less false positives or false negatives.

Research Question 5

How to improve the usefulness of ambiguity detection reports?

With a useful report we mean any information that helps the grammar developer with understanding and solving his ambiguities. The goal of this question is to find more useful types of reporting, and ways of producing them.

Since the ambiguity problem for CFGs is undecidable in general, there will never be a final answer to these questions. However, the goal of this thesis is to sufficiently improve ambiguity detection to a level that is practical for context-free grammars of programming languages. The advances we have made towards this goal are summarized in the next section.

1.4 Overview of the Chapters and Contributions

The contributions of this thesis are distributed over the following chapters:

Chapter 2. The Usability of Ambiguity Detection Methods for Context-Free Grammars

This chapter presents a set of criteria to assess the practical usability of a given ambiguity detection method, which answers research question 1. After that, it contributes to research question 2 by evaluating the usability of three exisiting methods on a set of benchmark grammars. The methods under investigation are the sentence generator AMBER by Schröer [Sch01], the Noncanonical Unambiguity Test by Schmitz [Sch07b], and the LR(k) test by Knuth [Knu65].

Chapter 3. Faster Ambiguity Detection by Grammar Filtering

This chapter presents AMBIDEXTER, a new approach to ambiguity detection that combines both approximative and exhaustive searching. We extend the Noncanonical Unambiguity Test by Schmitz [Sch07b] to enable it to filter harmless production rules from a grammar. Harmless production rules are rules that certainly do not contribute to the ambiguity of a grammar. A filtered grammar contains the same ambiguities as the original, but can be much smaller. Because of this smaller search space, exhaustive methods like sentence generators will be able to find ambiguities faster. We experimentally validate an implementation of our grammar filtering technique on a series of grammars of real world programming languages. The results show that sentence generation times can be reduced with several orders of magnitude, which is a contribution to research question 3.

Chapter 4. Tracking Down the Origins of Ambiguity in Context-Free Grammars

This chapter contains the theoretical foundation of the grammar filtering technique presented in Chapter 3. We show how to extend both the Regular Unambiguity (RU) Test and the more accurate Noncanonical Unambiguity (NU) Test to find harmless production rules. With the RU Test our approach is able to find production rules that can only be used to derive unambiguous strings. With the NU Test it can also find productions that can only be used to derive unambiguous substrings of ambiguous strings. The approach is presented in a formal way and is proven correct.

We show that the accuracy of the grammar filtering can be increased by applying it in an iterative fashion. Since this also increases the accuracy of the RU Test and NU Test in general, this is a contribution to research question 4. Furthermore, harmless production rules are helpful information for the grammar developer, which contributes to research question 5.

Chapter 5. Scaling to Scannerless

This chapter presents a set of extensions to the grammar filtering technique to make it suitable for character-level grammars. Character-level grammars are used for generating scannerless parsers, and are typically more complex than token-based grammars. We present several character-level specific extensions that take disambiguation filters into account, as well as a general precision improving technique called grammar unfolding. We test an implementation of our approach on a series of real world programming languages and measure the improvements in sentence generation times. Although the gains are not as large as in Chapter 3, our technique proves to be very useful on most grammars.

This is again an advancement related to research question 3. Furthermore, the presented extensions also improve the accuracy of filtering harmless production rules from character-level grammars, which is a contribution to research question 4.

Chapter 6. Implementing AMBIDEXTER

This chapter presents our tool implementation of AMBIDEXTER. The tool was developed to experimentally validate our grammar filtering techniques, but also to be used in real grammar development. The tool consists of two parts: a harmless production rule filter, and a parallel sentence generator. We discuss the architecture as well as implementation details of both of these parts, and finish with advise for their usage. Special attention is given to the performance of our tool, which forms a contribution to research question 3.

Chapter 7. Parse Forest Diagnostics with DR. AMBIGUITY

Once an ambiguity detection method finds an unwanted ambiguity, it should be removed from the grammar. However, it is not always straightforward for the grammar developer to see which modifications solve his ambiguity and in which way. This chapter presents an expert system called DR. AMBIGUITY, that can automatically propose applicable cures for removing an ambiguity from a grammar.

After giving an overview of different causes of ambiguity and ambiguity resolutions, the internals of DR. AMBIGUITY are described. The chapter ends with a small experimental validation of the usefulness of the expert system, in which it is applied on a realistic character-level grammar for Java. By proposing a technique that helps in understanding and resolving found ambiguities, this chapter addresses research question 5.

1.5 Origins of the Chapters

- Chapter 2 is published in the Proceedings of the Eighth Workshop on Language Descriptions, Tools and Applications (LDTA 2008) [Bas09].
- Chapter 3 is published in the Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications (LDTA 2010) [BV10]. It was co-authored by Jurgen Vinju.
- Chapter 4 is a revised version of a paper published in the proceedings of the Seventh International Colloquium on Theoretical Aspects of Computing (ICTAC 2010) [Bas10].
- Chapter 5 is published in the proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011) [BKV11]. It was co-authored by Paul Klint and Jurgen Vinju.
- Chapter 6 is based on a tool demonstation paper that is published in the proceedings of the Tenth IEEE Working Conference on Source Code Analysis and Manipulation (SCAM 2010) [BvdS10]. This paper was written together with Tijs van der Storm.
- Chapter 7 is published in the proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011) [BV11]. It was co-authored by Jurgen Vinju.

Chapter 2

The Usability of Ambiguity Detection Methods for Context-Free Grammars

"In theory there is no difference between theory and practice. In practice there is."
Jan L. A. van de Snepscheut / Yogi Berra

Despite the fact that the ambiguity problem for context-free grammars is undecidable in general, various ambiguity detection methods (ADMs) exist that aim at finding ambiguities in real grammars. In this chapter we present a set of criteria to test whether a given ambiguity detection method is useful in practice. We test implementations of three ADMs on a set of benchmark grammars, and assess their practical usability.

2.1 Introduction

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

Some grammars are designed to be completely unambiguous, while others are intended to contain a certain degree of ambiguity (for instance programming languages that will be type

This chapter was published in the Proceedings of the Eighth Workshop on Language Descriptions, Tools and Applications (LDTA 2008) [Bas09].

checked after parsing). In both cases it is important to know the sources of ambiguity in the developed grammar, so they can be resolved or verified.

Unfortunately, detecting the (un)ambiguity of a grammar is undecidable in the general case [Can62, Flo62, CS63]. Still, several Ambiguity Detection Methods (ADMs) exist that approach the problem from different angles, all with their own strengths and weaknesses. A straightforward one is to start generating all sentences of the grammar's language and checking them for ambiguity. The results of this method are 100% correct, but its problem is that it might never terminate. Other methods test for inclusion in unambiguous grammar classes (LALR, LR(k), LR Regular, etc.), but these do not cover the entire set of unambiguous grammars. More recent methods (Noncanonical Unambiguity [Sch07b], Ambiguity Checking with Language Approximation [BGM10]) search conservative approximations of a grammar or its language, leaving the original ambiguities intact. They are able to terminate in finite time, but at the expense of accuracy.

All these different characteristics result in differences in the practical usability of the ADMs. Whether a method is useful in a certain situation also depends on other factors like the tested grammar, the parameters of the method, the computation power of the used machine, the experience of the grammar developer, etc. In this chapter we investigate the practical usability of three ADM implementations in a series of use cases and compare them to each other. The investigated implementations are: AMBER [Sch01] (a derivation generator), MSTA [Mak95] (a parse table generator used as LR(k) test [Knu65]) and a modified version of BISON that implements the Noncanonical Unambiguity test [Sch10].

Overview In Section 2.2 we describe the criteria for practical usability and how they were measured. In Sections 2.3 to 2.5 we discuss the measurement results of the three investigated methods and analyze their practical usability. The methods are compared to each other in Section 2.6. Section 2.7 contains a short evaluation and we conclude in Section 2.8.

2.2 Comparison Framework

In order to measure and compare the practical usability of the investigated methods, we will first need to define it. This section describes the criteria for practical usability that we distinguish, how we measured them and how we analyzed the results.

2.2.1 Criteria for Practical Usability

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

Accuracy After a method has terminated on a grammar it needs to correctly identify the ambiguity of the tested grammar. Not all methods are always able to produce the right answer, for instance those that use approximation techniques. Reports of such methods need to be verified by the user, which influences the usability.

We define the accuracy of an ADM on a set of grammars as its percentage of correct reports. This implies that a method first has to terminate before its report can be tested. Executions that do not terminate within a set time limit are not used in our accuracy calculations.

Performance Knowing the worst case complexity of an algorithm tells something about the relation between its input and its number of steps, but this does not tell much about its runtime behavior on a certain grammar. How well a method performs on an average desktop PC also influences its usability.

Usefulness of reports After a method has successfully terminated and correctly identified the ambiguity of a grammar, it becomes even more useful if it indicates the sources of ambiguity in the grammar. That way they can be verified or resolved. An ambiguity report should be grammar oriented, localizing and succinct. A very useful one would be an ambiguous example string, preferably as short as possible, together with its multiple derivations.

Scalability Some ADMs can be executed with various levels of accuracy. There is usually a trade-off between their accuracy and performance. Accurately inspecting every single possible solution is more time consuming than a more superficial check. The finer the scale with which the accuracy of an ADM can be exchanged for performance, the more usable the method becomes.

2.2.2 Measurements

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

The medium and large collections were formed of grammars from the real life languages HTML[1], SQL[2], Pascal[3], C[3] and Java[4]. Their respective sizes were 29, 79, 176, 212 and 349 productions rules, with the line between medium and large drawn at 200 productions. Of each grammar (except HTML) five ambiguous versions were created through minor modifications. Again, some of the introduced[5] ambiguities were common ambiguous grammar constructs. Complete references of the grammars and modifications made can be found in [Bas07].

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

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

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

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

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

[5]See Section 3.3 for an overview.

performance. We measured the accuracy of the methods on the collection of small grammars to minimize computation time and memory usage.

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

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

2.2.3 Analysis

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

We finally analyzed the practical usability of the ADMs in each of the stated use cases. For each ADM we determined their critical properties that were most decisive for this, and tested if their values were within the constraints of the use cases. Our main focus will be on the ambiguous grammars. We will assume that a grammar developed for a generalized parser has a high chance of being ambiguous the first time it is tested.

2.3 AMBER

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

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

2.3.1 Measurements and Analysis

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

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

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

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

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

2.4 LR(k) Test

One of the first tests for unambiguity was the LR(k) test by Knuth [Knu65]. If an LR(k) parse table without conflicts can be constructed for a grammar, every string of its language is deterministically parsable and thus unambiguous. To test for LR(k)-ness of a grammar we used the parser generator MSTA [Mak95]. We ran it iteratively on a grammar with increasing k, until the test succeeded or the set time limit was reached. This test has a fixed precision and is therefore not scalable.

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

2.4.1 Measurements and Analysis

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

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

The maximum values of k that could be tested for within the time limits of the extensive check were 6 for the medium grammars, and only 3 for the large grammars. This should not be a problem, because chances are minimal that a grammar that is not LR(1) is LR(k) for higher values of k [GJ08]. However, experience also shows that grammars written for generalized parsers are virtually never suitable for deterministic parsing. They aim to best describe the structure of their language, instead of meeting specific parser requirements [GJ08]. The LR(k) test will probably not be a very useful ambiguity detection method for them.

2.5 Noncanonical Unambiguity Test

Schmitz' Noncanonical Unambiguity (NU) test [Sch07b] is a conservative ambiguity detection method that uses approximation to limit its search space. It constructs a nondeterministic finite automaton (NFA) that describes an approximation of the language of the tested grammar. The approximated language is then checked for ambiguity by searching the automaton for different paths that describe the same string. This can be done in finite time, but at the expense of correctness. The test is conservative however, allowing only false positives. If a grammar is Noncanonical Unambiguous then it is not ambiguous.

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

The accuracy of the test depends on the used approximation technique. Stricter approximations are usually more accurate, but result in larger NFAs. We tested an implementation [Sch10] of the NU test, which offered LR(0), SLR(1) and LR(1) precision. Their NFAs resemble the LR(0), SLR(1) or LR(1) parsing automata of a grammar, but without the use of a stack. Those of LR(0) and SLR(1) have the same amount of nodes, but the latter is more deterministic because the transitions are constrained by lookahead. The LR(1) automata are fairly bigger.

2.5.1 Measurements and Analysis

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

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

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

[1]Scores range from – – to ++

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

Table 2.1: Summary of measurement results

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

Scores range from – – – to +++

Table 2.2: Usability of investigated ADMs on ambiguous grammars of use cases

2.6 Comparison

In the previous three sections we have discussed the measurement results of the three methods. They are summarized in Table 2.1. From these results we have analyzed the practical usability of the methods in each of the stated use cases. In this chapter we will compare the methods to each other. Table 2.2 presents a (subjective) summary of this comparison.

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

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

The LR(k) test was the least usable of the three. It is actually only helpful for grammars that are LR(k) the first time they are tested. In all other cases it will never terminate and its intermediate reports are hard to trace to (possible) sources of ambiguity. Also, in general the LR(k) precision of the NU test is guaranteed to be stronger than the LR(k) test, for every value of k [Sch07b]. The LR(1) precision of the NU test did indeed find no ambiguities in grammars that MSTA identified as LR(1).

2.7 Evaluation

As a consequence of the different input formats of the investigated implementations we used only grammars in BNF notation. All three ADMs support the use of priority and associativity declarations, but it is wrong to assume they all adopt the same semantics [BBV07]. It was hard finding grammars for generalized parsers that do not use any form of disambiguation, so we gathered only YACC grammars. To create unambiguous base lines for the medium and large grammars we removed all their conflicts, effectively making them LALR(1) and thus also LR(1). The ambiguous versions we created are of course not LR(1), but might still be close. This should not be a problem for the analysis of AMBER and the LR(k) test (since we focus on ambiguous grammars), but it might result in the NU test reporting less conflicts. However, it will not influence its accuracy measurements because the NU test gives conservative answers.

We have investigated implementations of three ADMs, but it would also be interesting to compare them to other existing methods. Unfortunately, the investigated implementations were the only ones readily available at the start of this project, with the exception of the derivation generator of Jampana [Jam05]. However, we choose not to include it because it closely resembles AMBER, and it is very likely to produce incorrect results. It only generates derivations in which a production rule is never used more than once, and assumes all ambiguities of a grammar will show up.

A method that is particulary interesting is the recent "Ambiguity Checking with Language Approximation" framework (ACLA) by Brabrand et al. [BGM10]. After our own project had ended, Schmitz compared the accuracy of his ADM to that of ACLA on an extension of our collection of small unambiguous grammars [Sch07a]. His LR(0), SLR(1) and LR(1) precisions achieved accuracies of 65%, 75% and 87%, compared to 69% of ACLA. However, he did not apply any grammar unfolding, which might improve this latter score. 69% of the grammars were LR(k).

Another interesting method, which was not available at the time of this research, is CFG ANALYZER by Axelsson et al. [AHL08]. This method uses an incremental SAT-solver to exhaustively search all sentences of a grammar with increasing length. In Chapter 3 we do use a tool implementation of CFG ANALYZER. Although the goal of the described experiments is different, the results do allow for a comparison with AMBER. They show that on all tested grammars, CFG ANALYZER is equally accurate as AMBER, but much faster.

2.8 Conclusions

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

2.8.1 Discussion

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

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

Chapter 3

Faster Ambiguity Detection by Grammar Filtering

The previous chapter showed two relatively useful, but quite opposite, ambiguity detection methods: the approximative Noncanonical Unambiguity Test and the exhaustive sentence generator AMBER. *In this chapter we present* AMBIDEXTER, *a new approach to ambiguity detection that combines both approximative and exhaustive searching. We extend the Noncanonical Unambiguity Test to enable it to filter harmless production rules from a grammar. Harmless production rules are rules that certainly do not contribute to the ambiguity of a grammar. A filtered grammar contains the same ambiguities as the original, but can be much smaller. Because of this smaller search space, sentence generators like* AMBER *will be able to find ambiguities faster. We experimentally validate an implementation of our grammar filtering technique on a series of grammars of real world programming languages. The results show that sentence generation times can be reduced with several orders of magnitude.*

3.1 Introduction

Real programming languages are often defined using ambiguous context-free grammars. Some ambiguities are intentional, while others are accidental. It is therefore important to know all of them, but this can be a very cumbersome job if done by hand. Automated ambiguity

This chapter was published in the Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications (LDTA 2010) [BV10]. It was co-authored by Jurgen Vinju. Compared to the published paper, Section 3.2.6 contains an updated grammar reconstruction algorithm. Furthermore, Section 3.5 is a small appendix with updated measurement results.

checkers are therefore very valuable tools in the grammar development process, even though the ambiguity problem is undecidable in general.

In Chapter 2 we compared the practical usability of several ambiguity detection methods on a series of grammars.The exhaustive derivation generator AMBER [Sch01] was the most practical in finding ambiguities for real programming languages, despite its possible nontermination. The main reasons for this are its accurate reports (Figure 3.1) that contain examples of ambiguous strings, and its impressive efficiency. It took about 7 minutes to generate all the strings of length 10 for Java. Nevertheless, this method does not terminate in case of unambiguity and has exponential performance. For example, we were not able to analyze Java beyond a sentence length of 12 within 15 hours.

Another good competitor was Schmitz's Noncanonical Unambiguity Test [Sch07b] (NU TEST). This approximative method always terminates and can provide relatively accurate results in little time. The method can be tuned to trade accuracy for performance. Its memory usage grows to impractical levels much faster than its running time. For example, with the best available accuracy, it took more than 3Gb to fully analyze Java. A downside is that its reports can be hard to understand due to their abstractness (Figure 3.2).

In this chapter we propose to combine these two methods. We show how the NU TEST can be extended to identify parts of a grammar that do not contribute to any ambiguity. This information can be used to limit a grammar to only the part that is potentially ambiguous. The smaller grammar is then fed to the exhaustive AMBER and CFG ANALYZER [AHL08] methods to finally obtain a precise ambiguity report.

The goal of our approach is ambiguity detection that scales to real grammars and real sentence lengths, providing accurate ambiguity reports. Our new filtering method leads to significant decreases in running time for AMBER and CFG ANALYZER, which is a good step towards this goal.

Related Work Another approximative ambiguity detection method is the "Ambiguity Checking with Language Approximation" framework [BGM10] by Brabrand, Giegerich and Møller. The framework makes use of a characterization of ambiguity into horizontal and vertical ambiguity to test whether a certain production rule can derive ambiguous strings. This method might be extended in a comparable fashion as we propose to extend the NU TEST here.

Other exhaustive ambiguity detection methods are [CU95] and [Gor63]. These can benefit from our grammar filtering similarly to AMBER and CFG ANALYZER.

Outline In Section 3.2 we explain the NU TEST, how to extend it to identify harmless productions, and how to construct a filtered grammar. Section 3.3 contains an experimental validation of our method. We summarize our results in Section 3.4.

3.2 Filtering Unambiguous Productions

In this section we explain how to filter productions from a grammar that do not contribute to any ambiguity. We first briefly recall the basic NU TEST algorithm before we explain how to extend it to identify harmless productions. This section ends by explaining how to construct a

```
GRAMMAR DEBUG INFORMATION
Grammar ambiguity detected. (disjunctive)
Two different ``type_literals'' derivation trees for the same phrase.

TREE 1
------
type_literals alternative at line 787, col 9 of grammar {
  VOID_TK
  DOT_TK
  CLASS_TK
}

TREE 2
------
type_literals alternative at line 785, col 16 of grammar {
  primitive_type alternative at line 31, col 9 of grammar {
    VOID_TK
  }
  DOT_TK
  CLASS_TK
}
```

Figure 3.1: Excerpt from an ambiguity report by AMBER on a Java grammar.

```
5 potential ambiguities with LR(1) precision detected:
    (method_header -> modifiers type method_declarator throws . ,
     method_header -> modifiers VOID_TK method_declarator throws . )
    (method_header -> type method_declarator throws . ,
     method_header -> VOID_TK method_declarator throws . )
    (method_header -> type method_declarator throws . ,
     method_header -> modifiers VOID_TK method_declarator throws . )
    (method_header -> VOID_TK method_declarator throws . ,
     method_header -> modifiers type method_declarator throws . )
    (type_literals -> primitive_type DOT_TK CLASS_TK . ,
     type_literals -> VOID_TK DOT_TK CLASS_TK . )
```

Figure 3.2: Excerpt from an ambiguity report by NU TEST on a Java grammar.

valid filtered grammar that can be fed to any exhaustive ambiguity checker. A more detailed description of our method, together with proofs of correctness, can be found in Chapter 4.

3.2.1 Preliminaries

A *grammar* G is a four-tuple (N, T, P, S) where N is the set of non-terminals, T the set of terminals, P the set of productions over $N \times (N \cup T)^*$, and S is the start symbol. V is defined as $N \cup T$. We use $A, B, C, \ldots$ to denote non-terminals, $u, v, w, \ldots$ for strings of T^*, and $\alpha, \beta, \gamma, \ldots$ for sentential forms: strings over V^*. The relation $\Longrightarrow$ denotes derivation. We say $\alpha B\gamma$ directly derives $\alpha\beta\gamma$, written as $\alpha B\gamma \Longrightarrow \alpha\beta\gamma$ if a production rule $B \to \beta$ exists in P.

The symbol $\Longrightarrow^*$ means "derives in zero or more steps". An *item* indicates a position in a production rule using a dot, for instance as $S \to A \bullet BC$.

3.2.2 The Noncanonical Unambiguity Test

The Noncanonical Unambiguity test [Sch07b] by Schmitz is an approximated search for two different parse trees of the same string. It uses a *bracketed grammar*, which is obtained from an input grammar by adding a unique terminal symbol to the beginning and end of each production. The language of a bracketed grammar represents all parse trees of the original grammar.

From the bracketed grammar a *position graph* is constructed, in which the nodes are positions in strings generated by this grammar. The edges represent evaluation steps of the bracketed grammar: there are *derivation*, *reduction*, and *shift* edges. Derivations and reductions correspond to entries and exits of a production rule, while shifts correspond to steps inside a single production rule over terminal and non-terminal symbols.

This position graph describes the same language as the bracketed grammar. Every path through the graph describes a parse tree of the original grammar. Therefore, the existence of two different paths of which the labels of shift edges form the same string indicates the ambiguity of the grammar. So, position graphs help to point out ambiguity in a straightforward manner, but they are usually infinitely large. To obtain analyzable graphs Schmitz describes the use of equivalence relations on the nodes. These should induce conservative approximations of the unambiguity property of the grammar. If they report ambiguity we know that the input grammar is *potentially ambiguous*, otherwise we know for sure that it is unambiguous.

3.2.3 LR(0) Approximation

An equivalence relation that normally yields an approximated graph of analyzable size is the "item_0" relation [Sch07b]. We use item_0 here to explain the NU TEST for simplicity's sake, ignoring the intricacies of other equivalence relations.

The item_0 position graph of a grammar closely resembles its LR(0) parse automaton [Knu65]. The nodes are labeled with the LR(0) items of the grammar and the edges correspond to actions. Every node with the dot at the beginning of a production of the start symbol is a *start node*, and every item with the dot at the end of a production of the start symbol is an *end node*. There are three types of transitions:

- Shift transitions, of form $A \to \alpha \bullet X \beta \overset{X}{\longmapsto} A \to \alpha X \bullet \beta$
- Derivation transitions, of form $A \to \alpha \bullet B \gamma \overset{\langle_i}{\longmapsto} B \to \bullet \beta$, where i is the number of the production $B \to \beta$.
- Reduction transitions, of form $B \to \beta \bullet \overset{\rangle_i}{\longmapsto} A \to \alpha B \bullet \gamma$, where i is the number of the production $B \to \beta$.

The derivation and shift transitions are similar to those in an LR(0) automaton, but the reductions are different. The item_0 graph has reduction edges to every item that has the dot after the reduced non-terminal, while an LR(0) automaton jumps to a different state depending on the symbol that is at the top of the parse stack. As a result, a certain path through an item_0 graph with a $\langle_i$ transition from $A \to \alpha{\bullet}B\gamma$ does not necessarily match an $\rangle_i$ transition to $A \to \alpha B{\bullet}\gamma$. The language characterized by an item_0 position graph is thus a superset of the language of parse trees of the original grammar.

3.2.4 Finding Ambiguity in an item_0 Position Graph

To find possible ambiguities, we can traverse the item_0 graph using two cursors simultaneously. If we can traverse the graph while the two cursors use different paths, but construct the same string of shifted tokens, we have identified a possible ambiguity.

An efficient representation of all such simultaneous traversals is a *pair graph* (PG). The nodes of this graph represent the pair of cursors into the original item_0 graph. The edges represent steps made by the two cursors, but not all transitions are allowed. An edge exists for either an individual derivation or reduction transitions by one of the cursors, or for a simultaneous shift transition of the exact same symbol by both cursors.

A path in a PG thus describes two potential parse trees of the same string. We call such a path an *ambiguous path pair*, if the two paths it represents are not identical. The existence of ambiguous path pairs is indicated by a *join point*: a reduce transition from a pair with different items to a pair with identical items. Ergo, in the item_0 case we can efficiently detect (possible) ambiguity by constructing a PG and looking for join points.

To optimize the process of generating PGs we can omit certain nodes and edges. In particular, if two paths derive the exact same substring for a certain non-terminal this substring can safely be replaced by a shift over the non-terminal. We call this process *terminalization* of a non-terminal. Such optimizations avoid the traversal of spurious ambiguities.

3.2.5 Filtering Harmless Production Rules

The NU TEST stops after a PG is constructed and the ambiguous path pairs are reported to the user. In our approach we also use the PG to identify production rules that certainly do not contribute to the ambiguity of the grammar. We call these *harmless* production rules.

The main idea is that a production rule is harmless if its items are not used in any ambiguous path pair. The set of ambiguous path pairs describes an over-approximation of the set of all parse trees of ambiguous strings. So, if a production is not used by this set it is certainly not used by any real parse tree of an ambiguous string.

Note that a production like that may still be used in a parse tree of an ambiguous sentence, but then it does not cause ambiguity in itself. In this case the ambiguity already exists in a sentential form in which the non-terminal of the production is not derived yet.

We use knowledge about harmless rules to filter the PG and to eventually produce a filtered grammar containing only rules that potentially contribute to ambiguity. This is an outline of our algorithm:

1. Remove pairs not used on any ambiguous path pair.

2. Remove noticeably invalid (over-approximated) paths, until a fixed-point:

a) Remove incompletely used productions.
b) Remove unmatched derivation and reduction steps.
c) Prune dead ends and unreachable sub-graphs.

3. Collect the potentially harmful production rules that are left over.

Step 1 and Step 3 are the essential steps, but there is room for optimization. Because the item_0 graph is an over-approximation, collecting the harmful productions also takes parse trees into account that are invalid for the original grammar. There are at least two situations in which these can be easily identified and removed.

Incompletely Used Productions

Consider that any path in the item_0 graph that describes a valid parse tree of the original grammar must exercise all items of a production. So, if any item for a production is not used by any ambiguous path pair, then the entire production never causes ambiguous parse trees for a sentence for the original grammar.

Note that due to over-approximation, other items of the identified production may still be used in other valid paths in the item_0 graph, but these paths will not be possible in the unapproximated position graph since they would combine items from different productions.

Once an incompletely used production is identified, all pairs that contain one of its items can be safely removed from the pair graph and new dead ends and unreachable sub-graphs can be pruned. This removes some over-approximated invalid paths from the graph.

Unmatched Derivations and Reductions

Furthermore, next to nodes we can also remove certain derivation and reduction edges from the PG. Consider that any path in the item_0 graph that describes a valid parse tree of the original grammar must both derive and reduce every production that it uses. More specifically, if a $\langle_i$ transition is followed from $A \to \alpha \bullet B\gamma$ to $B \to \bullet\beta$, the matching $\rangle_i$ transition from $B \to \beta\bullet$ to $A \to \alpha B \bullet \gamma$ must also be used, and vice versa. Therefore, if one of the two is used in the PG, but the other is not, it can be safely removed, and the PG can be pruned again.

The process of removing items and transitions can be repeated until no more invalid paths can be found this way. After that the remaining PG uses only potentially harmful productions. We can gather them by simply collecting the productions from all items used in the graph. Note that the item_0 graph remains an over-approximation, so we might collect productions that are actually harmless. In Section 3.3 we investigate whether the reduction of the grammar will actually result in performance gains for exhaustive methods.

3.2.6 Grammar Reconstruction

From applying the previous filtering process we are left with a set of productions that potentially lead to ambiguity. We want to use this set of productions as input to an exhaustive ambiguity detection method such as CFG ANALYZER or AMBER in order to get precise reports and clear example sentences. Note that the set of potentially ambiguous productions may be empty, in which case this step can be omitted completely.

Unfortunately, the filtered set of productions can represent an incomplete grammar. There might be non-terminals of which all productions are filtered, while they still occur in productions of other non-terminals (they have been terminalized). In this case we need to restore the productivity of these non-terminals. Furthermore, certain non-terminals might not be reachable from the start symbol anymore. This means that, in the original grammar, these non-terminals can only be used after the application of a harmless production rule. By definition they are therefore harmless as well, and we can safely discard them.

To restore the productivity property of the grammar, new production rules and terminals will have to be introduced. Naturally, we must prevent introducing new ambiguities in this process. Let us use P_{h} to denote the set of potentially harmful productions of a grammar. From P_{h} we can create a new grammar G' by constructing[1]:

1. The set of defined non-terminals of P_{h}:
   $N_{\mathsf{def}} = \{A | A \to \alpha \in P_{\mathsf{h}}\}$.
2. The used but undefined non-terminals of P_{h}:
   $N_{\mathsf{undef}} = \{B | A \to \alpha B \beta \in P_{\mathsf{h}}\} \backslash N_{\mathsf{def}}$.
3. The unproductive non-terminals:
   $N_{\mathsf{unpr}} = \{A | A \in N_{\mathsf{def}}, \neg\exists u : A \Longrightarrow^* u \text{ using only productions in } P_{\mathsf{h}}\}$.
4. New terminals t_A for each non-terminal $A \in N_{\mathsf{undef}} \cup N_{\mathsf{unpr}}$.
5. Productions to complete the unproductive and undefined non-terminals:
   $P' = P_{\mathsf{h}} \cup \{A \to (t_A)^k \mid A \in N_{\mathsf{undef}} \cup N_{\mathsf{unpr}}, k = \mathsf{minlength}(A)\}$.
6. The new set of terminal symbols:
   $T' = \{a | (A \to \beta a \gamma) \in P'\}$.
7. Finally, the new grammar:
   $G' = (N_{\mathsf{def}} \cup N_{\mathsf{undef}}, T', P', S')$.

At step 5 we reconstruct the productivity of unproductive non-terminals. For each non-terminal, we introduce a production that produces a terminal-only string with the same length as the shortest possible derivation of the non-terminal in the original grammar. This way every derivation of the original grammar corresponds to a derivation of equal or shorter length in the filtered grammar. The number of derivations of the filtered grammar up to a certain length is then always less or equal to that of the original grammar, and certainly not greater. This way, filtering a grammar can never lead to an increase in sentence generation time. Furthermore, the introduced productions make use of new unique terminals to not introduce new ambiguities.

[1] Where $\mathsf{minlength}(A) = \mathsf{min}(\{k | \exists u,\ A \Longrightarrow^* u : k = |u|\})$ using the original grammar.

3.3 Experimental Validation

After constructing a new, much smaller, grammar we can apply exhaustive algorithms like AMBER or CFG ANALYZER on it to search for the exact sources of ambiguity. The search space for these algorithms is exponential in the size of the grammar. Therefore our experimental hypothesis is:

> *By filtering the input grammar we can gain an order of magnitude improvement in run-time when running* AMBER *or* CFG ANALYZER *as compared to running them on the original grammar.*

Since building an LR(0) PG and filtering it is polynomial we also hypothesize:

> *For many real-world grammars the time invested to filter them does not exceed the time that is gained when running* AMBER *and* CFG ANALYZER *on the filtered grammar.*

We will also experiment with other approximations, such as SLR(1), LALR(1) and LR(1) to be able to reason about the return of investment for these more precise approximations.

3.3.1 Experiment Setup

To evaluate the effectiveness of our approach we must run it on realistic cases. We focus on grammars for reverse engineering projects. Grammars in this area target many different versions and dialects of programming languages. They are subject to a lengthy engineering process that includes bug fixing and specialization for specific purposes. Our realistic grammars are therefore "standard" grammars for mainstream programming languages, augmented with small variations that reflect typical intentional and accidental deviations.

We have selected standard grammars for Java [GrJ], C [GrC], Pascal [GrP] and SQL [GrS] which are initially not ambiguous. We labeled them Java.0, C.0, Pascal.0 and SQL.0. Then, we seeded each of these grammars with different kinds of ambiguous extensions[2]. Examples of ambiguity introduced by us are:

- Dangling-else constructs: Pascal.3, C.2, Java.3
- Missing operator precedence: SQL.1, SQL.5, Pascal.2, C.1, Java.4
- Syntactic overloading:[3] SQL.2, SQL.3, SQL.4, Pascal.1, Pascal.4, Pascal.5, C.4, C.5, Java.1, Java.5
- Non-terminals nullable in multiple ways: C.3, Java.2

For each of these grammars we measure:[4]

[2]A complete overview of the applied modifications can be found in [Bas07].

[3]Syntactic overloading happens when reusing terminal symbols. E.g. the use of commas as list separator and binary operator, forgetting to reserve a keyword, or reuse of juxtapositioning.

[4]Measurements done on an Intel Core2 Quad Q6600 2.40GHz PC with 8Gb DDR2 memory.

1. AMBER/CFG ANALYZER run-time and memory usage,
2. Filtering run-time with precisions LR(0), SLR(1), LALR(1) or LR(1),
3. AMBER/CFG ANALYZER run-time and memory usage after filtering.

Observing only a marginal difference between measures 1 and 3 would invalidate our experimental hypothesis. Observing the combined run-times of measure 2 and 3 being longer than measure 1 would invalidate our second hypothesis.

To help explaining our results we also track the size of the grammar (**number of production rules**), the number of harmless productions found with each precision (**rules filtered**), and the number of tokens explored to identify the first ambiguity (**length**).

We have used AMBER version 30/03/2006[5] and CFG ANALYZER version 03/12/2007[6]. To experiment with the NU TEST algorithm and our extensions we have implemented a prototype in the Java programming language. We measured CPU user time with the GNU `time` utility and measured memory usage by polling a process with `pid` every 0.1 seconds.

[5]downloaded from `http://accent.compilertools.net/`

[6]downloaded from `http://www2.tcs.ifi.lmu.de/~mlange/cfganalyzer/`

3.3.2 Experimental Results

Results of Filtering Prototype All measurement results of running our filtering prototype on the benchmark grammars are shown in Table 3.1. As expected, every precision filtered a higher or equal number of rules than the one before. Columns 3 to 6 show how much production rules could be filtered with each of the implemented precisions. We see that LR(0) on average filtered respectively 76%, 12%, 19% and 16% of the productions of the SQL, Pascal, C and Java grammars. SLR(1) filtered the same or slightly more, with the largest improvement for the Java grammars: 19%. Remarkably, LALR(1) never found more harmless rules in the ambiguous grammars than SLR(1)[7]. LR(1) improved over SLR(1) for 12 out of 20 ambiguous grammars. On average it filtered 78% for SQL, a remarkable 64% for Pascal, and 21% for Java.

Columns 7 to 10 show the run-time of the filtering tool, and columns 11 to 14 show its memory usage. We see that the LR(0) and SLR(1) precisions always ran under 9 seconds and used at most 168Mb of memory. SLR(1) was slightly more efficient than LR(0), which can be explained by the fact that an SLR(1) position graph is generally more deterministic than its LR(0) counterpart. They both have the same number of nodes and edges, but the SLR(1) reductions are constrained by lookahead, which results in a smaller pair graph.

An LALR(1) position automaton is generally several factors larger than an LR(0) one, which shows itself in longer run-time and more memory usage. The memory usage of the LR(1) precision became problematic for the C and Java grammars. For all variations of both grammars it needed more than 4Gb. Therefore we ran it on the C and Java grammars that we filtered first with the SLR(1) precision, and then it only needed around 3Gb. Here we see that filtering with a lesser precision first can be beneficial for the performance of more expensive filters.

On average the tool uses its memory almost completely for storing the pair graph, which it usually builds in two thirds of its run-time. The other one third is used to filter the graph. If we project this onto the run-times of Schmitz' C tool [Sch10], it should filter all our grammars with LR(0) or SLR(1) in under 4 seconds, if extended.

[7]In Section 3.5 we repeat these measurements with an improved implementation, which does show an advantage of LALR(1) over SLR(1).

		Rules filtered				Time				Memory (Mb)			
Grammar	**Rules**	**LR0**	**SLR1**	**LALR1**	**LR1**	**LR0**	**SLR1**	**LALR1**	**LR1**	**LR0**	**SLR1**	**LALR1**	**LR1**
SQL.0	79	79	79	79	79	0.4s	0.4s	1.0s	3.1s	16	16	49	54
SQL.1	79	65	65	65	65	0.5s	0.4s	1.4s	3.9s	17	16	51	56
SQL.2	80	47	47	47	47	1.1s	1.1s	1.9s	6.0s	34	32	58	74
SQL.3	80	54	54	54	54	0.6s	0.5s	1.3s	3.9s	18	17	50	56
SQL.4	80	71	71	71	74	0.4s	0.4s	1.1s	3.1s	17	17	51	45
SQL.5	80	68	68	68	72	0.5s	0.4s	1.2s	3.9s	17	16	53	54
Pascal.0	176	21	30	176	176	2.4s	2.2s	2.4s	15.1s	50	42	160	181
Pascal.1	177	21	25	25	104	2.4s	2.4s	5.9s	40.3s	48	49	162	297
Pascal.2	177	21	25	25	104	2.3s	2.4s	5.8s	46.9s	52	51	159	325
Pascal.3	177	21	30	30	144	2.5s	2.2s	5.0s	20.7s	52	47	160	248
Pascal.4	177	20	24	24	103	2.4s	2.3s	5.9s	42.5s	50	50	163	294
Pascal.5	177	21	25	25	103	2.4s	2.3s	5.8s	32.8s	52	49	159	326
C.0	212	41	44	212	212	4.2s	3.9s	15.8s	9m40s	88	83	427	1397
C.1	213	41	44	44	44	4.3s	3.7s	2m03s	1h45m	100	80	615	2898
C.2	213	41	44	44	44	4.3s	3.9s	1m45s	41m58s	101	80	611	2940
C.3	213	40	43	43	43	4.2s	4.1s	1m59s	42m57s	87	81	615	2885
C.4	213	41	44	44	44	4.2s	3.9s	2m06s	1h30m	87	80	607	2894
C.5	213	40	43	43	43	4.3s	3.9s	1m55s	47m15s	91	80	631	3107
Java.0	349	56	70	349	349	8.2s	6.9s	54.0s	37m47s	153	116	556	1362
Java.1	350	56	70	70	74	8.2s	6.9s	10m24s	3h55m	144	118	1088	2908
Java.2	350	53	66	66	70	8.8s	7.8s	29m57s	8h48m	168	124	1427	3209
Java.3	350	56	70	70	74	8.3s	6.9s	10m38s	3h27m	146	120	1123	3014
Java.4	350	55	69	69	73	8.2s	6.6s	10m57s	4h11m	156	119	1117	3073
Java.5	350	53	66	66	70	8.3s	6.9s	10m40s	8h01m	153	121	1117	3126

Table 3.1: Results of Filtering (LR1 was run on C and Java after filtering first with SLR1, due to excessive memory usage). These are the results as published in [BV10]. For newer results measured with an improved version of the AMBIDEXTER tool, see Table 3.4.

	Time				
Grammar	**Unfiltered**	**LR0**	**SLR1**	**LR1**	**Length**
SQL.1	28m26s	**0.1s**	0.1s	-	15
SQL.2	0.0s	0.0s	0.0s	-	7
SQL.3	0.0s	0.0s	0.0s	-	6
SQL.4	0.0s	0.0s	0.0s	0.0s	9
SQL.5	1.3s	**0.0s**	0.0s	0.0s	11
Pascal.1	0.3s	0.1s	0.1s	0.0s	9
Pascal.2	0.0s	0.0s	0.0s	0.0s	7
Pascal.3	31.8s	2.9s	1.9s	**0.0s**	11
Pascal.4	0.0s	0.0s	0.0s	0.0s	8
Pascal.5	0.0s	0.0s	0.0s	0.0s	8
C.1	42.1s	0.1s	**0.0s**	-	5
C.2	>4.50h[1]	>18.8h	>15.3h	-	>11
C.3	0.1s	0.0s	0.0s	-	3
C.4	42.0s	0.5s	**0.4s**	-	5
C.5	19m09s	0.7s	**0.5s**	-	6
Java.1	>25.0h[2]	12.2h	3.9h	**3.7h**	13
Java.2	0.0s	0.0s	0.0s	0.0s	1
Java.3	1h25m	5m35s	2m28s	**2m21s**	11
Java.4	17.0s	2.9s	1.8s	**1.7s**	9
Java.5	0.1s	0.0s	0.0s	0.0s	7

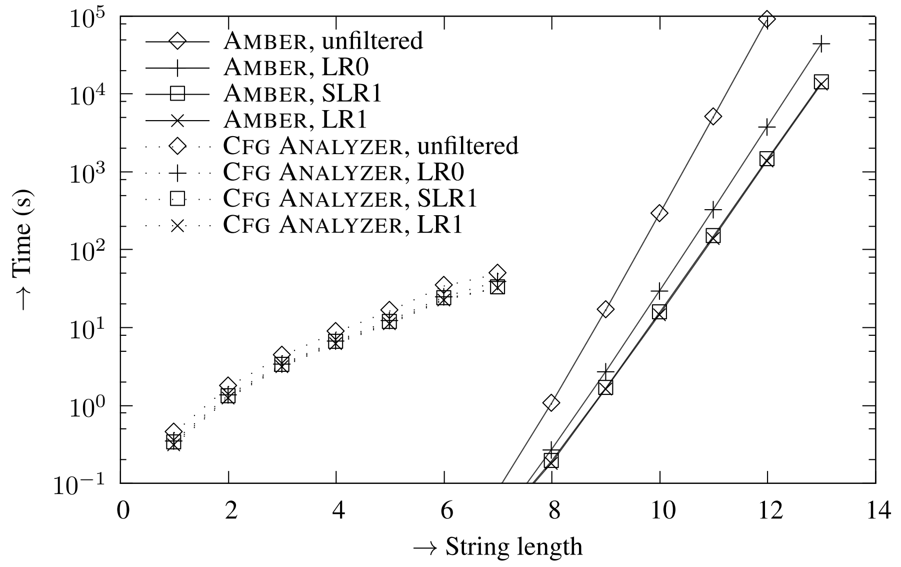
[1]only reached string length of 7.
[2]only reached string length of 12.

Table 3.2: Running AMBER on filtered and non-filtered grammars.

Results of AMBER Table 3.2 shows the effects of grammar filtering on the behavior of AMBER. Columns 2 to 5 show the time AMBER needed to find the ambiguity in the original grammars and the ones filtered with various precisions. There is no column for the LALR(1) precision, because it always filtered the same number of rules as SLR(1). For LR(1) we only mention the cases in which it filtered more than SLR(1). AMBER's memory usage was always less than 1 Mb of memory.

In all cases we see a decrease in run-time if more rules were filtered, sometimes quite drastically. For instance the unfiltered Java.1 grammar was impossible to check in under 25 hours, while filtered with SLR(1) or LR(1) it only needed less than 4 hours. The C.2 grammar still remains uncheckable within 15 hours, but the LR(0) and SLR(1) filtering extended the maximum string length possible to search within this time from 7 to 11. The decreases in run-time per string length for this grammar are shown in Figure 3.3.

This confirms our first hypothesis. To test our second hypothesis, we also need to take the run-time of our filtering tool into account. Figure 3.4 shows the combined computation times of filtering and running AMBER, compared to only running AMBER on the unfiltered grammars. Not all SQL grammars are mentioned because both filtering and AMBER took under 1 second in all cases. Also, timings of filtering with LR(1) are not mentioned because they are obviously too high and would reduce the readability of the graph. Apart from that, we see that the short filtering time of LR(0) and SLR(1) do not cancel out the decrease in run-time for grammars SQL.1, SQL.5, Pascal.3, C.1, C.4, C.5, Java.3 and Java.4. Add to that the effects on grammars C.2 and Java.1 and we get a significant improvement for 10 out of 20 ambiguous



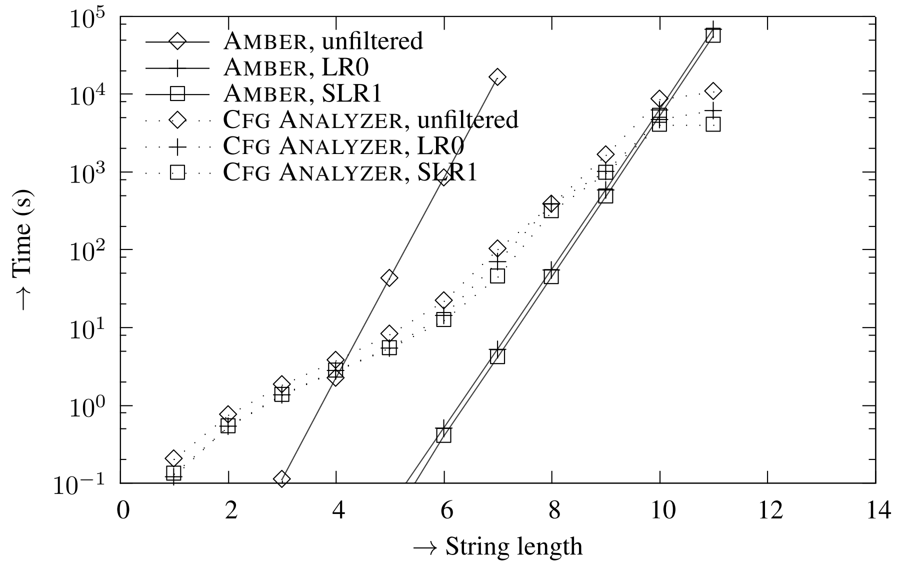


Figure 3.3: Run-time of AMBER and CFG ANALYZER on grammars Java.1 (syntax overloading) above and C.2 (dangling-else) below.

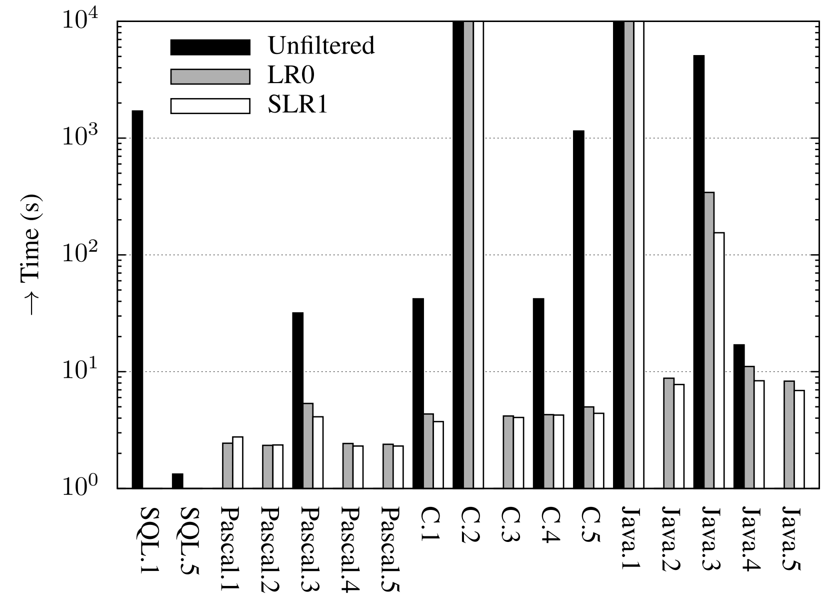


Figure 3.4: Added run-time of grammar filtering and ambiguity checking with AMBER.

grammars. For the other 10 grammars we don't see improvements because AMBER already took less time than it took to filter them.

Column 6 shows the string lengths that AMBER had to search to find the ambiguity in each grammar. All filtered grammars required the same string length as their original versions, as could be expected from our grammar reconstruction algorithm.

Results of CFG ANALYZER Table 3.3 shows the same results as Table 3.2 but then for CFG ANALYZER. Again we see a decrease in run-time in almost all cases, as the number of filtered rules increases, but less significant than in the case of AMBER. We also see that CFG ANALYZER is much faster than AMBER. It was even able to check the SLR(1) filtered C.2 grammar in 1 hour and 7 minutes. CFG ANALYZER's memory usage always stayed under 70Mb, except for C.2: it used 1.21Gb for the unfiltered grammar, 1.31Gb for the LR(0) filtered one, and 742Mb in the SLR(1) case.

We see that CFG ANALYZER always needed smaller lengths than AMBER. This is because CFG ANALYZER searches all parse trees of all non-terminals simultaneously, whereas AMBER only checks those of the start symbol.

Figure 3.5 shows the combined run-times of our filtering tool and CFG ANALYZER. Here we see only significant improvements for grammars SQL.1, SQL.5, C.2, Java.1 and Java.3. In all other cases CFG ANALYZER took less time to find the first ambiguity than it took our tool to filter a grammar.

Grammar	Unfiltered	Time LR0	SLR1	LR1	Length
SQL.1	17.6s	**1.8s**	1.8s	-	11
SQL.2	0.4s	0.1s	0.1s	-	3
SQL.3	0.4s	0.0s	0.1s	-	3
SQL.4	1.4s	**0.0s**	0.0s	0.0s	5
SQL.5	14.4s	0.8s	0.8s	**0.4s**	11
Pascal.1	1.1s	0.9s	0.9s	**0.3s**	3
Pascal.2	0.5s	0.4s	0.4s	0.1s	2
Pascal.3	9.6s	8.1s	7.5s	**1.2s**	7
Pascal.4	1.1s	0.9s	0.9s	**0.3s**	3
Pascal.5	3.5s	0.9s	0.9s	**0.3s**	3
C.1	1.7s	1.3s	**1.3s**	-	3
C.2	3.00h	1.77h	**1.11h**	-	11
C.3	0.7s	0.5s	0.5s	-	2
C.4	1.7s	1.3s	1.3s	-	3
C.5	6.6s	5.1s	**4.9s**	-	5
Java.1	48.9s	39.2s	32.5s	**32.4s**	7
Java.2	0.5s	0.4s	0.4s	0.4s	1
Java.3	47.2s	40.0s	35.2s	**35.1s**	7
Java.4	8.4s	6.7s	**6.5s**	6.5s	4
Java.5	4.3s	3.4s	3.3s	**3.3s**	3

Table 3.3: Running CFG ANALYZER on filtered and non-filtered grammars.

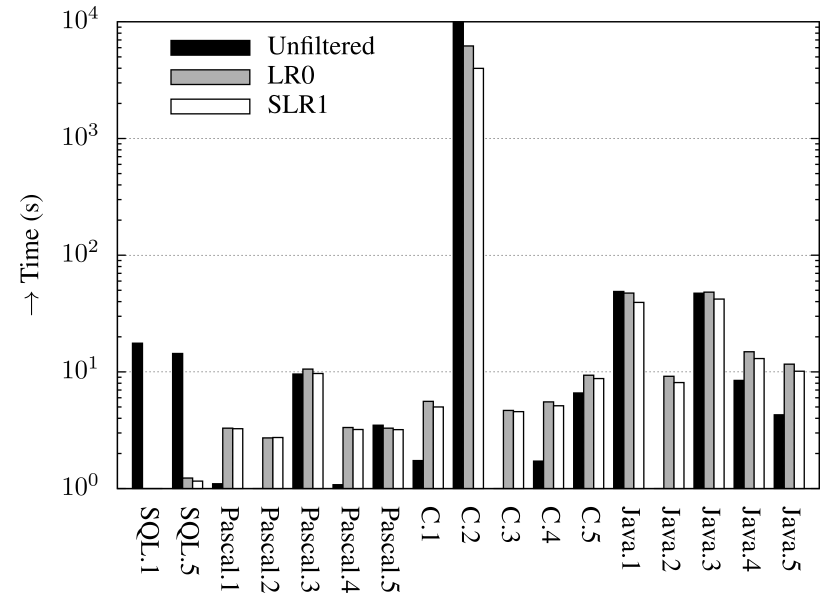


Figure 3.5: Added run-time of grammar filtering and ambiguity checking with CFG ANALYZER.

3.3.3 Analysis and Conclusions

We saw that filtering more rules resulted in shorter run-times for both AMBER and CFG ANALYZER. Especially AMBER profited enormously for certain grammars. The reductions in run-time of CFG ANALYZER were smaller but still significant. This largely confirms our first hypothesis.

We conclude that the SLR(1) precision was the most beneficial for reducing the run-time of AMBER and CFG ANALYZER, while requiring only a small filtering overhead. In some cases LR(1) provided slightly larger reductions, but these did not match up against its own long run-time. Filtering with SLR(1) resulted in significant decreases in run-time for AMBER on 10 of the 20 ambiguous grammars, and for CFG ANALYZER on 5 grammars. In all other cases the filtering did not contribute to an overall reduction, because it took longer than the time the tools initially needed to check the unfiltered grammars. Nevertheless, this was never more than 9 seconds. Therefore our second hypothesis is confirmed for the situations that really matter.

3.3.4 Threats to validity

Internally a bug in our implementation would invalidate our conclusions. This is unlikely since we tested and compared our results with other independently constructed tools (NU TEST [Sch10], CFG ANALYZER and AMBER) for a large number of grammars and we obtained the same results. Our source code is available for your inspection at `http://homepages.cwi.nl/~basten/ambiguity/`. Also note that our Java version is slower than Schmitz' original implementation in C. An optimized version would eliminate some of the overhead we observed while analyzing small grammars[8].

As for external validity, it is entirely possible that our method does not lead to significant decreases in run-time for any specific grammar that we did not include in our experiment. However, we did select representative grammars and the ambiguities we seeded are typical extensions or try-outs made by language engineers.

3.4 Conclusions

We proposed to adapt the approximative NU TEST to a grammar filtering tool and to combine that with the exhaustive AMBER and CFG ANALYZER ambiguity detection methods. Using our grammar filters we can conservatively identify production rules that do not contribute to the ambiguity of a grammar. Filtering these productions from the grammar lead to significant reductions in run-time, sometimes orders of magnitude, for running AMBER and CFG ANALYZER. The result is that we could produce precise ambiguity reports in a much shorter time for real world grammars.

[8]We are thankful to Arnold Lankamp for his help fixing efficiency issues in our Java version.

3.5 Appendix: Updated Measurement Results

The results show in Table 3.1 were measured with the first prototype implementation of our grammar filtering technique. Unfortunately, this first prototype turned out to be too inefficient for running the character-level experiments of Chapter 5. We therefore improved our tool for this type of grammars, which also had a positive effect on checking the token-based grammars of this chapter.

3.5.1 Improved Implementation

In our first implementation we represented all item pairs and pair transitions of the pair graph in memory, and performed the filtering and pruning on these data structures. In the next design we chose to filter the NFA instead, and rebuild the pair graph after each iteration. This had the advantage that we did not have to store all pair graph transitions, which consumed the largest part of the memory. Another advantage was that the filtering and pruning of the NFA required much less time. This enabled us to implement a more thourough NFA pruning algorithm, which consists of a full reachability analysis of the NFA. For a complete discription of the latest design and implementation details see Chapter 6.

Because of this new design, our new implementation became efficient enough for filtering character-level grammars. However, it also performed much better on the token-based grammars checked in this chapter, both in terms of performance and filtering accuracy. To show these improvements we repeated the measurements of Table 3.1 with our latest implementation of AMBIDEXTER. The updated results are shown in Table 3.4. We see various things:

3.5.2 Analysis

Harmless Production Rules First, the better NFA pruning leads to the detection of more harmless production rules. For almost all grammars, LR(0) found more harmless rules than before. SLR(1) and LALR(1) also improved substantially on all ambiguous Pascal grammars. On the Java grammars, SLR(1) found on average 30 more harmless rules. In only four cases, LR(1) finds more rules than LALR(1). However, the increases of 66 and 68 for respectively C.1 and C.5 are quite remarkable.

Computation Time The new timing figures for LR(0) and SLR(1) show no substantial differences. Checking with LALR(1) on the other hand, has become much faster. Its average running time on the C grammars decreased from around two minutes to twelve seconds. For LR(1) we see similar improvements. It can check all the Java grammars in under nine minutes, where our previous implementation required at least three and a half hours.

Memory Usage In the figures of the memory usage of the precisions we only see big differences for LR(1). The memory required for the C and Java grammars looks higher, but this is because we did not need to pre-filter these grammars first. The new implementation was able to test them all within 5Gb.

		Rules filtered				Time				Memory (Mb)			
Grammar	**Rules**	**LR0**	**SLR1**	**LALR1**	**LR1**	**LR0**	**SLR1**	**LALR1**	**LR1**	**LR0**	**SLR1**	**LALR1**	**LR1**
SQL.0	79	79	79	79	79	0.6s	0.5s	0.8s	1.5s	22	21	27	59
SQL.1	79	65	65	65	65	0.6s	0.7s	1.0s	1.8s	21	31	47	64
SQL.2	80	47	47	47	47	0.9s	0.9s	1.4s	3.6s	33	45	61	100
SQL.3	80	**68**	**68**	**68**	**68**	0.6s	0.6s	0.9s	1.7s	36	32	44	65
SQL.4	80	71	71	71	74	0.7s	0.7s	0.9s	1.7s	38	39	34	58
SQL.5	80	68	68	68	72	0.6s	0.6s	0.8s	1.6s	35	33	27	60
Pascal.0	176	**30**	**176**	176	176	3.4s	2.0s	1.9s	3.7s	90	66	67	150
Pascal.1	177	**30**	**53**	**104**	104	3.2s	3.4s	4.5s	8.4s	88	96	109	270
Pascal.2	177	**30**	**53**	**104**	104	3.3s	3.1s	4.3s	9.3s	89	90	107	297
Pascal.3	177	**30**	**144**	**144**	144	3.3s	2.4s	3.1s	5.6s	88	80	85	170
Pascal.4	177	**29**	**52**	**103**	103	3.3s	3.4s	4.5s	9.1s	86	93	113	289
Pascal.5	177	**30**	**53**	**103**	103	3.4s	3.6s	4.0s	8.8s	90	94	110	286
C.0	212	**44**	44	212	212	4.4s	4.3s	5.2s	1m3s	128	117	162	1488
C.1	213	**44**	44	44	44	4.1s	4.0s	12.8s	4m41s	117	118	527	4242
C.2	213	**44**	44	**114**	**180**	4.3s	4.2s	10.0s	3m02s	127	115	362	3339
C.3	213	**43**	43	43	43	4.0s	3.8s	12.1s	4m15s	119	116	496	4150
C.4	213	**44**	44	44	44	4.1s	4.0s	12.2s	4m28s	125	119	474	4411
C.5	213	**43**	43	**100**	**168**	4.1s	4.2s	10.6s	3m33s	130	117	353	3198
Java.0	349	**86**	**101**	349	349	6.4s	6.2s	7.6s	1m13s	207	194	294	2291
Java.1	350	**86**	**101**	**101**	**101**	6.7s	6.3s	28.1s	7m03s	204	199	1008	4735
Java.2	350	**83**	**96**	**96**	**96**	6.8s	6.5s	39.6s	8m45s	239	201	1298	5105
Java.3	350	**86**	**101**	**101**	**101**	6.4s	6.7s	27.3s	7m18s	206	203	1154	4704
Java.4	350	**83**	**98**	**98**	**98**	7.2s	6.0s	29.3s	7m27s	212	191	947	4702
Java.5	350	**82**	**93**	**93**	**93**	6.8s	5.8s	30.4s	7m10s	187	199	1001	4685

Table 3.4: Updated results of grammar filtering experiments shown in Table 3.1, measured with optimized version of the AMBIDEXTER tool. Improvements in number of harmless production rules found are highlighted in boldface.

	Time AMBER				
Grammar	**Previous best**	**SLR1**	**LALR1**	**LR1**	**Length**
C.2	>15.3h[1] (SLR1)	-	19.1h	8.4s	13
Java.1	3.7h (LR1)	3.1h	-	-	13
Java.3	2m21s (LR1)	1m54s	-	-	11

[1]only reached string length of 11.

	Time CFG ANALYZER				
Grammar	**Previous best**	**SLR1**	**LAR1**	**LR1**	**Length**
C.2	1.11h (SLR1)	-	31m43s	5.4s	11
Java.1	32.4s (LR1)	30.4s	-	-	7
Java.3	35.1s (LR1)	28.9s	-	-	7

Table 3.5: Updated results of Tables 3.2 and 3.3: running AMBER and CFG ANALYZER on a selection of grammars, before and after filtering with the improved version of AMBIDEXTER. Only measurements relevant due to an increase of detected harmless productions are shown.

Concluding, we see that the new figures show the anticipated superiority of LALR(1) over SLR(1). On four Pascal grammars and two C grammars it found around twice as much harmless production rules. Furthermore, LALR(1) equaled LR(1) on all but four grammars, while requiring much less time and memory. On the bigger C and Java grammars, LALR(1) finished in 10 to 40 seconds, while LR(1) required between 3 to 9 minutes.

3.5.3 Effects on Sentence Generation Times

If we extrapolate our new filtering results to the sentence generation experiments of this chapter, the number of grammars for which AMBER and CFG ANALYZER will benefit from filtering do not change. The reason for this is that for the majority of grammars, both tools were already very fast either before or after filtering. The only results that might change significantly are the sentence generation times for grammars C.2, Java.1 and Java.3.

Table 3.5 show the results of checking the new filtered versions of these three grammars with AMBER and CFG ANALYZER. We see that the newly filtered productions indeed result in even faster sentence generation times. Both tools perform a little better on the versions of Java.1 and Java.3 filtered with SLR1. However, for the C.2 grammar, which is the most complex grammar in our set, we see spectacular speedups. Both AMBER and CFG ANALYZER are now able to find the ambiguity in the LR1 filtered grammar in under 10 seconds, were they first took over 15 hours, and 1.11 hours respectively. Therefore, the new implementation of our grammar filter shows that both LALR(1) and LR(1) are now viable approximation precisions as well.

Chapter 4

Tracking Down the Origins of Ambiguity in Context-Free Grammars

"All difficult things have their origin in that which is easy, . . . "
Lao Tzu

This chapter contains the theoretical foundation of the grammar filtering technique described in the previous chapter. We show how to extend both the Regular Unambiguity (RU) Test and the more accurate Noncanonical Unambiguity (NU) Test to find harmless production rules. With the RU Test our approach is able to find production rules that can only be used to derive unambiguous strings. With the NU Test it can also find productions that can only be used to derive unambiguous substrings of ambiguous strings. The approach is presented in a formal way and is proven correct.

4.1 Introduction

Context-free grammars (CFGs) are widely used in various fields, such as programming language development, natural language processing, and bioinformatics. They are suitable for the definition of a wide range of languages, but their possible ambiguity can hinder their use. Designed ambiguities are not uncommon, but accidentally introduced ambiguities are unwanted. Ambiguities are very hard to detect by hand, so automated ambiguity checkers are welcome tools.

This chapter is a revised version of a paper published in the proceedings of the Seventh International Colloquium on Theoretical Aspects of Computing (ICTAC 2010) [Bas10]. Compared to the published article, this chapter includes all proofs. Furthermore, Sections 4.4.2, 4.5.2 and 4.5.3 are improved.

Despite the fact that the CFG ambiguity problem is undecidable in general [Can62, Flo62, CS63], various detection schemes exist. They can roughly be divided into two categories: exhaustive methods and approximative ones. Methods in the first category exhaustively search the usually infinite set of derivations of a grammar, while the latter ones apply approximation to limit their search space. This enables them to always terminate, but at the expense of potentially incorrect reports. Exhaustive methods do produce precise reports, but only if they find ambiguity before they are halted. The latter is essential since otherwise the searching could potentially run forever.

Because of the undecidability it is impossible to always terminate with a correct and detailed report. The challenge is to develop a method that gives the most precise answer in the time available. In this chapter we propose to combine exhaustive and approximative methods as a step towards this goal. We show how to extend the Regular Unambiguity Test and Noncanonical Unambiguity Test [Sch07a, Sch07b] of Schmitz to improve the precision of their approximation and that of their ambiguity reports. The extension enables the detection of *harmless production rules*, which are rules that do not contribute to the ambiguity of a grammar. These are already helpful reports for the grammar developer, but can also be used to narrow the search space of other detection methods. In Chapter 3 we witnessed significant reductions in the run-time of exhaustive methods due to our grammar filtering.

4.1.1 Related Work

The original Noncanonical Unambiguity Test by Schmitz is an approximative test for the unambiguity of a grammar. The approximation it applies is always conservative, so it can only find a grammar to be *unambiguous* or *potentially ambiguous*. Its answers always concern the grammar as a whole, but the reports of a prototype implementation [Sch10] by the author also contain clues about the production rules involved in the potential ambiguity. However, these are very abstract and hard to understand. The extensions that we present do result in precise reports, while remaining conservative.

Another approximative ambiguity detection scheme is the “Ambiguity Checking with Language Approximation” framework [BGM10] by Brabrand, Giegerich and Møller. The framework makes use of a characterization of ambiguity into horizontal and vertical ambiguity — which is equal to [AL90] — to test whether a certain production rule can derive ambiguous strings. The difference with our approach is that we test whether a production rule is vital for the existence of parse trees of ambiguous strings.

4.1.2 Overview

We start with background information about grammars and languages in Section 4.2. Then we repeat the definition of the Regular Unambiguity (RU) Test in Section 4.3. In Section 4.4 we explain how the RU Test can be extended to identify sets of parse trees of unambiguous strings. From these parse trees we can identify harmless production rules as explained in Section 4.5 . Section 4.6 explains the Noncanonical Unambiguity (NU) Test, an improvement over the RU Test, and also shows how it improves the effect of our parse tree and production rule filtering.

In Section 4.7 we describe how our approach can be used iteratively to increase its accuracy. Finally, Section 4.8 contains the conclusion.

4.2 Preliminaries

This section gives a quick overview of the theory of grammars and languages, and introduces the notational convention used throughout this document. For more background information we refer to [HU79, SSS88].

4.2.1 Context-Free Grammars

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

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

We use V to denote the set $N \cup T$, and V' for $V \cup \{\varepsilon\}$. The following characters are used to represent different symbols and strings: $a, b, c, \ldots$ represent terminals, $A, B, C, \ldots$ represent nonterminals, X, Y, Z represent either nonterminals or terminals, $\alpha, \beta, \gamma, \ldots$ represent strings in V^*, $u, v, w, \ldots$ represent strings in T^*, ε represents the empty string.

A production (A, α) in P is written as $A \rightarrow \alpha$. We use the function $\mathsf{pid} : P \rightarrow \mathbb{N}$ to relate each production to a unique identifier. An *item* [Knu65] indicates a position in the right hand side of a production using a dot. Items are written like $A \rightarrow \alpha \bullet \beta$. We use I to denote the set of items of a grammar.

The relation $\Longrightarrow$ denotes direct derivation, or derivation in one step. Given the string $\alpha B \gamma$ and a production rule $B \rightarrow \beta$, we can write $\alpha B \gamma \Longrightarrow \alpha \beta \gamma$ (read $\alpha B \gamma$ directly derives $\alpha \beta \gamma$). The symbol $\Longrightarrow^*$ means "derives in zero or more steps". A sequence of derivation steps is simply called a *derivation*. Strings in V^* are called *sentential forms*. We call the set of sentential forms that can be derived from S of a grammar G, the *sentential language* of G, denoted $\mathcal{S}(G)$. A sentential form in T^* is called a *sentence*. The set of all sentences that can be derived from S of a grammar G is called the *language* of G, denoted $\mathcal{L}(G)$.

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

The *parse tree* of a sentential form α describes how α is derived from S, but disregards the order of the derivation steps. To represent parse trees we use bracketed strings (See Section 4.2.3). A grammar G is ambiguous iff there is at least one string in $\mathcal{L}(G)$ for which multiple parse trees exist.

4.2.2 Bracketed Grammars

From a grammar $G = (N, T, P, S)$ a *bracketed grammar* G_b can be constructed, by adding unique terminals to the beginning and end of every production rule [GH67]. The bracketed grammar G_b is defined as the 4-tuple (N, T_b, P_b, S), where:

- $T_b = T \cup T_\langle \cup T_\rangle$,
- $T_\langle = \{ \langle_i \mid \exists p \in P : i = \mathsf{pid}(p)\}$,
- $T_\rangle = \{ \rangle_i \mid \exists p \in P : i = \mathsf{pid}(p)\}$,
- $P_b = \{A \to \langle_i \alpha \rangle_i \mid A \to \alpha \in P, i = \mathsf{pid}(A \to \alpha)\}$.

V_b is defined as $T_b \cup N$, and V_b' as $V_b \cup \{\varepsilon\}$. We use $a_b, b_b, \ldots$ and X_b, Y_b, Z_b to represent symbols in respectively T_b and V_b. Similarly, $u_b, v_b, \ldots$ and $\alpha_b, \beta_b, \ldots$ represent strings in respectively T_b^* and V_b^*, The relation $\Longrightarrow_b$ denotes direct derivation using productions in P_b. The homomorphism h from V_b^* to V^* maps each string in $\mathcal{S}(G_b)$ to $\mathcal{S}(G)$. It is defined by $h(\langle_i) = \varepsilon$, $h(\rangle_i) = \varepsilon$, and $h(X) = X$.

4.2.3 Parse Trees

$\mathcal{L}(G_b)$ describes exactly all parse trees of all strings in $\mathcal{L}(G)$. $\mathcal{S}(G_b)$ describes exactly all parse trees of all strings in $\mathcal{S}(G)$. We divide it into two disjoint sets:

Definition 1. *The set of* parse trees of ambiguous strings *of G is* $\mathcal{P}^a(G) = \{\alpha_b \mid \alpha_b \in \mathcal{S}(G_b), \exists \beta_b \in \mathcal{S}(G_b) : \alpha_b \neq \beta_b, h(\alpha_b) = h(\beta_b)\}$. *The set of parse trees of unambiguous strings of G is* $\mathcal{P}^u(G) = \mathcal{S}(G_b) \setminus \mathcal{P}^a(G)$.

Example 4.1. Below is an example grammar (4.1) together with its bracketed version (4.2). The string aaa has two parse trees, $\langle_1\langle_2\langle_2\langle_3 a\rangle_3\langle_3 a\rangle_3\rangle_2\langle_3 a\rangle_3\rangle_2\rangle_1$ and $\langle_1\langle_2\langle_3 a\rangle_3\langle_2\langle_3 a\rangle_3\langle_3 a\rangle_3\rangle_2\rangle_2\rangle_1$, and is therefore ambiguous.

$$1 : S \to A, \quad 2 : A \to AA, \quad 3 : A \to a \tag{4.1}$$

$$1 : S \to \langle_1 A\rangle_1,\ 2 : A \to \langle_2 AA\rangle_2,\ 3 : A \to \langle_3 a\rangle_3 \tag{4.2}$$

4.2.4 Ambiguous Core

We call the set of the smallest possible ambiguous sentential forms of G the *ambiguous core* of G. These are the ambiguous sentential forms that cannot be derived from other sentential forms that are already ambiguous. Their parse trees are the smallest indicators of the ambiguities in G.

Definition 2. *The set of parse trees of the* ambiguous core *of a grammar G is* $\mathcal{C}^a(G) = \{\alpha_b \mid \alpha_b \in \mathcal{P}^a(G), \neg\exists \beta_b \in \mathcal{P}^a(G) : \beta_b \Longrightarrow_b \alpha_b\}$

From $\mathcal{C}^a(G)$ we can obtain $\mathcal{P}^a(G)$ by adding all sentential forms reachable with $\Longrightarrow_b$. And since $\mathcal{C}^a(G) \subseteq \mathcal{P}^a(G)$ we get the following Lemma:

Lemma 1 (ambiguous core)**.** *A grammar G is ambiguous iff $\mathcal{C}^a(G)$ is non-empty.*

Similar to $\mathcal{P}^u(G)$, we define the complement of $\mathcal{C}^a(G)$ as $\mathcal{C}^u(G) = \mathcal{S}(G_b) \setminus \mathcal{C}^a(G)$, for which holds that $\mathcal{P}^u(G) \subseteq \mathcal{C}^u(G)$.

Example 4.2. The two parse trees $\langle_1\langle_2\langle_2 AA\rangle_2 A\rangle_2\rangle_1$ and $\langle_1\langle_2 A\langle_2 AA\rangle_2\rangle_2\rangle_1$, of the ambiguous sentential form AAA, are in the ambiguous core of Grammar (4.1).

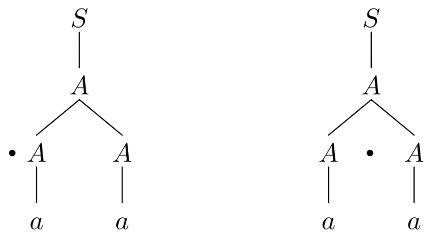
4.2.5 Positions

A *position* in a sentential form is an element in $V_b^* \times V_b^*$. The position (α_b, β_b) is written as $\alpha_b \bullet \beta_b$. We use $\mathsf{pos}(G_b)$ to denote the set of all positions in strings of $\mathcal{S}(G_b)$. It is defined as $\{\alpha_b \bullet \beta_b \mid \alpha_b\beta_b \in \mathcal{S}(G_b)\}$.

Every position in $\mathsf{pos}(G_b)$ is a position in a parse tree, and corresponds to an item of G. The item of a position can be identified by the closest enclosing $\langle_i$ and $\rangle_i$ pair around the dot, considering balancing. For positions with the dot at the beginning or the end we introduce two special items $\bullet S$ and $S\bullet$.

We use the function item to map a position to its item. It is defined by $\mathsf{item}(\gamma_b \bullet \delta_b) = A \to \alpha' \bullet \beta'$ iff $\gamma_b \bullet \delta_b = \eta_b \langle_i \alpha_b \bullet \beta_b \rangle_i \theta_b$, $A \to \langle_i \alpha' \beta' \rangle_i \in P_b$, $\alpha' \Longrightarrow_b^* \alpha_b$ and $\beta' \Longrightarrow_b^* \beta_b$, $\mathsf{item}(\bullet \alpha_b) = \bullet S$, and $\mathsf{item}(\alpha_b \bullet) = S\bullet$. Another function items returns the set of items used at all positions in a parse tree. It is defined as $\mathsf{items}(\alpha_b) = \{A \to \alpha \bullet \beta \mid \exists \gamma_b \bullet \delta_b : \gamma_b\delta_b = \alpha_b,\ A \to \alpha\bullet\beta = \mathsf{item}(\gamma_b \bullet \delta_b)\}$.

Example 4.3. The following shows the parse tree representations of the positions $\langle_1\langle_2 \bullet \langle_3 a\rangle_3 \langle_3 a\rangle_3\rangle_2\rangle_1$ and $\langle_1\langle_2\langle_3 a\rangle_3 \bullet \langle_3 a\rangle_3\rangle_2\rangle_1$. We see that the first position is at item $A \to \bullet AA$ and the second is at $A \to A \bullet A$.



The function proditems maps a production rule to the set of all its items. It is defined as $\mathsf{proditems}(A \to \alpha) = \{A \to \beta \bullet \gamma \mid \beta\gamma = \alpha\}$. If a production rule is used to construct a parse tree, then all its items occur at one or more positions in the tree.

Lemma 2 (items in parse trees)**.** *$\forall \alpha_b \langle_i \beta_b \rangle_i \gamma_b \in \mathcal{S}(G_b) : \exists A \to \delta \in P : \mathsf{pid}(A \to \delta) = i$, $\mathsf{proditems}(A \to \delta) \subseteq \mathsf{items}(\alpha_b \langle_i \beta_b \rangle_i \gamma_b)$.*

4.2.6 Automata

An *automaton* A is a 5-tuple (Q, Σ, R, Q_s, Q_f) where Q is the set of *states*, Σ is the input alphabet, R in $Q \times \Sigma \times Q$ is the set of *rules* or *transitions*, $Q_s \subseteq Q$ is the set of *start states*, and $Q_f \subseteq Q$ is the set of *final states*. A transition (q_0, a, q_1) is written as $q_0 \overset{a}{\longmapsto} q_1$. The language of an automaton is the set of strings read on all paths from a start state to an end state. Formally, $\mathcal{L}(A) = \{\alpha \mid \exists q_s \in Q_s,\, q_f \in Q_f : q_s \overset{\alpha}{\longmapsto}{}^* q_f\}$.

4.3 Regular Unambiguity Test

This section introduces the Regular Unambiguity (RU) Test [Sch07b] by Schmitz. The RU Test is an approximative test for the existence of two parse trees for the same string, allowing only false positives.

4.3.1 Position Automaton

The basis of the Regular Unambiguity Test is a *position automaton*, which describes all strings in $\mathcal{S}(G_b)$. The states of this automaton are the positions in $\mathsf{pos}(G_b)$. The transitions are labeled with elements in V_b.

Definition 3. *The* position automaton[1] $\Gamma(G)$ *of a grammar G is the tuple* (Q, V_b, R, Q_s, Q_f), *where*

- $Q = \mathsf{pos}(G_b)$,
- $R = \{\alpha_b \bullet X_b \beta_b \overset{X_b}{\longmapsto} \alpha_b X_b \bullet \beta_b \mid \alpha_b X_b \beta_b \in \mathcal{S}(G_b)\}$,
- $Q_s = \{\bullet \alpha_b \mid \alpha_b \in \mathcal{S}(G_b)\}$,
- $Q_f = \{\alpha_b \bullet \mid \alpha_b \in \mathcal{S}(G_b)\}$.

There are three types of transitions: *derives* with labels in $T_\langle$, *reduces* with labels in $T_\rangle$, and *shifts* of terminals and nonterminals in V. The symbols read on a path through $\Gamma(G)$ describe a parse tree of G. Thus, $\mathcal{L}(\Gamma(G)) = \mathcal{S}(G_b)$.

A grammar G is ambiguous iff two paths exist through $\Gamma(G)$ that describe different parse trees in $\mathcal{P}^a(G)$ — strings in $\mathcal{S}(G_b)$ — of the same string in $\mathcal{S}(G)$. We call such two paths an *ambiguous path pair*.

Example 4.4. Figure 4.1 shows the first part of the position automaton of the grammar from Example 4.1. It shows paths for parse trees S, $\langle_1 A \rangle_1$ and $\langle_1 \langle_3 a \rangle_3 \rangle_1$.

4.3.2 Approximated Position Automaton

If G has an infinite number of parse trees, the position automaton is also of infinite size. Checking it for ambiguous path pairs would take forever. Therefore the position automaton is approximated using equivalence relations on the positions, also known as *quotienting*. The approximated position automaton has equivalence classes of positions for its states. For every transition between two positions in the original automaton a new transition with the same

[1] We modified the original definition of the position automaton to be able to explain our extensions more clearly. This does not essentially change the RU Test and NU Test however, since their only requirement on $\Gamma(G)$ is that it defines $\mathcal{S}(G_b)$.

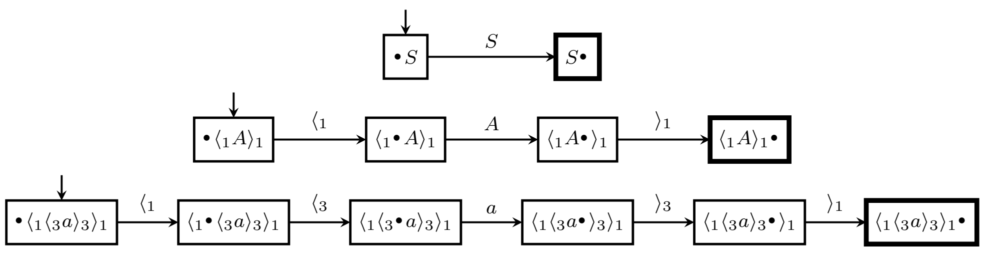

$\bullet S$
S
$S\bullet$
$\bullet\langle_1 A\rangle_1$
$\langle_1$
$\langle_1\bullet A\rangle_1$
A
$\langle_1 A\bullet\rangle_1$
$\rangle_1$
$\langle_1 A\rangle_1\bullet$
$\bullet\langle_1\langle_3 a\rangle_3\rangle_1$
$\langle_1$
$\langle_1\bullet\langle_3 a\rangle_3\rangle_1$
$\langle_3$
$\langle_1\langle_3\bullet a\rangle_3\rangle_1$
a
$\langle_1\langle_3 a\bullet\rangle_3\rangle_1$
$\rangle_3$
$\langle_1\langle_3 a\rangle_3\bullet\rangle_1$
$\rangle_1$
$\langle_1\langle_3 a\rangle_3\rangle_1\bullet$

Figure 4.1: Part of the position automaton of the grammar of Example 4.1.

label then exists between the equivalence classes that the positions are in. If an equivalence relation is used that yields a finite set of equivalence classes, the approximated automaton can be checked for ambiguous path pairs in finite time.

Definition 4. *Given an equivalence relation* $\equiv$ *on positions, the* approximated position automaton $\Gamma_\equiv(G)$ *of the automaton* $\Gamma(G) = (Q, V_b, R, Q_s, Q_f)$*, is the tuple* $(Q_\equiv, V_b', R_\equiv, \{q_s\}, \{q_f\})$ *where*

- $Q_\equiv = Q/{\equiv} \cup \{q_s, q_f\}$*, where* $Q/{\equiv}$ *is the set of non-empty equivalence classes over* Q *modulo* $\equiv$*, defined as* $\{[\alpha_b\bullet\beta_b]_\equiv \mid \alpha_b\bullet\beta_b \in Q\}$*,*
- $R_\equiv = \{[q_0]_\equiv \overset{X_b}{\longmapsto} [q_1]_\equiv \mid q_0 \overset{X_b}{\longmapsto} q_1 \in R\} \cup \{q_s \overset{\varepsilon}{\longmapsto} [q]_\equiv \mid q \in Q_s\} \cup \{[q]_\equiv \overset{\varepsilon}{\longmapsto} q_f \mid q \in Q_f\}$*,*
- q_s *and* q_f *are respectively a new start state and a new final state.*

The paths through $\Gamma_\equiv(G)$ describe an overapproximation of the set of parse trees of G, thus $\mathcal{L}(\Gamma(G)) \subseteq \mathcal{L}(\Gamma_\equiv(G))$. So if no ambiguous path pair exists in $\Gamma_\equiv(G)$, grammar G is unambiguous. But if there is an ambiguous path pair, it is unknown if its paths describe real parse trees of G or approximated ones. In this case we say G is *potentially ambiguous*.

4.3.3 The item_0 Equivalence Relation

Checking for ambiguous paths in finite time also requires an equivalence relation with which $\Gamma_\equiv(G)$ can be built in finite time. A relation like that should enable the construction of the equivalence classes without enumerating all positions in $\mathsf{pos}(G_b)$. A simple but useful equivalence relation with this property is the item_0 relation [Sch07b]. Two positions are equal modulo item_0 if they are both at the same item.

Definition 5. $\alpha_b\bullet\beta_b\ \mathsf{item}_0\ \gamma_b\bullet\delta_b$ *iff* $\mathsf{item}(\alpha_b\bullet\beta_b) = \mathsf{item}(\gamma_b\bullet\delta_b)$.

Intuitively the item_0 position automaton $\Gamma_{\mathsf{item}_0}(G)$ of a grammar resembles that grammar's LR(0) parse automaton [Knu65]. The nodes are the LR(0) items of the grammar and the X and $\rangle$ edges correspond to the shift and reduce actions in the LR(0) automaton. The $\langle$ edges correspond to the LR(0) item closure operation.

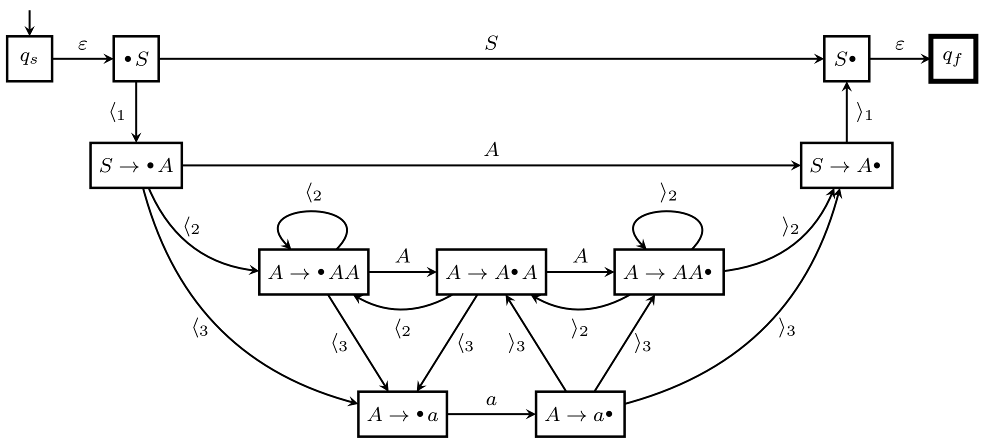


Figure 4.2: The item_0 position automaton of the grammar of Example 4.1.

The difference between an LR(0) automaton and an item_0 position automaton is in the reductions. The position automaton $\Gamma_{\mathsf{item}_0}(G)$ has reduction edges to every item that has the dot after the reduced nonterminal, while an LR(0) automaton jumps to a different state depending on the symbol that is at the top of the parse stack. As a result, a certain path through $\Gamma_{\mathsf{item}_0}(G)$ with a $\langle_i$ transition from $A \to \alpha\bullet B\gamma$ does not necessarily need to have a matching $\rangle_i$ transition to $A \to \alpha B\bullet\gamma$.

Example 4.5. Figure 4.2 shows the item_0 position automaton of the grammar of Example 4.1. Strings $\langle_1\langle_2\langle_3 a\rangle_3\rangle_1$ and $\langle_1\langle_3 a\rangle_3\rangle_1$ form an ambiguous path pair.

4.3.4 Position Pair Automaton

The existence of ambiguous path pairs in a position automaton can be checked with a *position pair automaton*, in which every state is a pair of states from the position automaton. Transitions between pairs are described using the *mutual accessibility relation* ma.

Definition 6. *The* regular position pair automaton $\Pi^R_{\equiv}(G)$ *of* $\Gamma_{\equiv}(G)$ *is the tuple* $(Q^2_{\equiv}, V_b'^2, \mathsf{ma}, q_s^2, q_f^2)$*, where* ma *over* $Q^2_{\equiv} \times V_b'^2 \times Q^2_{\equiv}$*, denoted by* $\rightrightarrows$*, is the union of the following subrelations:*

$$\begin{aligned}
\mathsf{maDl} &= \{(q_0,q_1) \xrightarrow{(\langle_i,\varepsilon)} (q_2,q_1) \mid q_0 \overset{\langle_i}{\longmapsto} q_2\},\\
\mathsf{maDr} &= \{(q_0,q_1) \xrightarrow{(\varepsilon,\langle_i)} (q_0,q_3) \mid q_1 \overset{\langle_i}{\longmapsto} q_3\},\\
\mathsf{maS} &= \{(q_0,q_1) \xrightarrow{(X,X)} (q_2,q_3) \mid q_0 \overset{X}{\longmapsto} q_2 \wedge q_1 \overset{X}{\longmapsto} q_3,\ X \in V'\},\\
\mathsf{maRl} &= \{(q_0,q_1) \xrightarrow{(\rangle_i,\varepsilon)} (q_2,q_1) \mid q_0 \overset{\rangle_i}{\longmapsto} q_2\},\\
\mathsf{maRr} &= \{(q_0,q_1) \xrightarrow{(\varepsilon,\rangle_i)} (q_0,q_3) \mid q_1 \overset{\rangle_i}{\longmapsto} q_3\}.
\end{aligned}$$

Automaton	**Symbol**	**States**	**Alphabet**
Position automaton	$\Gamma(G)$	$Q = \mathsf{pos}(G_b)$	V_b
Approximated position automaton	$\Gamma_\equiv(G)$	$Q_\equiv = Q/{\equiv} \cup \{q_s, q_f\}$	V_b
Position pair automaton	$\Pi^R_\equiv(G)$	$Q^2_\equiv$	V'^2_b

Table 4.1: Overview of the different automata used in the Regular Unambiguity Test.

Every path through this automaton from q^2_s to q^2_f describes two paths through $\Gamma_\equiv(G)$ that shift the same symbols. The language of $\Pi^R_\equiv(G)$ is thus a set of pairs of strings. A path indicates an ambiguous path pair if its two bracketed strings are different, but equal under the homomorphism h. Because $\mathcal{L}(\Gamma_\equiv(G))$ is an over-approximation of $\mathcal{S}(G_b)$, $\mathcal{L}(\Pi^R_\equiv(G))$ contains at least all ambiguous path pairs through $\Gamma(G)$.

Lemma 3 (ambiguous path pairs)**.** $\forall \alpha_b, \beta_b \in \mathcal{P}^a(G) : \alpha_b \neq \beta_b \wedge h(\alpha_b) = h(\beta_b) \Rightarrow (\alpha_b, \beta_b) \in \mathcal{L}(\Pi^R_\equiv(G))$.

From this we can conclude that the absence of ambiguous path pairs through $\Pi^R_\equiv(G)$ proves the unambiguity of G. However, in the other case it remains uncertain whether or not G is really ambiguous. The RU Test is therefore an approximative test, giving only conservative answers. This concludes the explanation of the RU Test. For an overview of the different automata used see Table 4.1.

4.4 Finding Parse Trees of Unambiguous Strings

The Regular Unambiguity Test described in the previous section can conservatively detect the unambiguity of a given grammar. If it finds no ambiguity we are done, but if it finds potential ambiguity this report is not detailed enough to be useful. In this section we show how the RU Test can be extended to identify parse trees of unambiguous strings. These will form the basis of more detailed ambiguity reports, as we will see in Section 4.5.

4.4.1 Unused Positions

From the states of $\Gamma_\equiv(G)$ that are not used on ambiguous path pairs, we can identify parse trees of unambiguous strings. For this we use the fact that every bracketed string that represents a parse tree of G must pass all its positions on its path through $\Gamma(G)$. Therefore, all positions in states of $\Gamma_\equiv(G)$ that are not used by any ambiguous path pair through $\Pi^R_\equiv(G)$ are positions in parse trees of unambiguous strings.

To find these parse trees we first gather all states covered by ambiguous path pairs.

Definition 7. *The set of states of* $\Gamma_\equiv(G)$ *that are used on ambiguous path pairs through* $\Pi^R_\equiv(G)$ *is* $Q^a_\equiv = \{q_0, q_1 \,|\, \exists \alpha_b, \beta_b, \alpha'_b, \beta'_b : \alpha_b\beta_b \neq \alpha'_b\beta'_b,\ q^2_s \xrightarrow{(\alpha_b,\alpha'_b)}{}^* (q_0, q_1) \xrightarrow{(\beta_b,\beta'_b)}{}^* q^2_f\}$. *The set of states not used on ambiguous path pairs is* $Q^u_\equiv = Q_\equiv \setminus Q^a_\equiv$.

The following lemma says that the states in $Q^a_\equiv$ together contain at least all positions in the parse trees of all ambiguous strings.

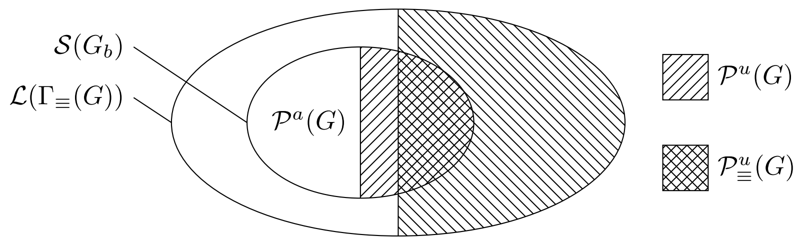


Figure 4.3: Euler diagram showing the relationship between $\mathcal{S}(G_b)$ and $\mathcal{L}(\Gamma_\equiv(G))$. The vertical lines divide both sets in two: their parse trees of ambiguous strings (left) and parse trees of unambiguous strings (right).

Lemma 4 (coverage of ambiguous path pairs)**.** $\forall \alpha_b\beta_b \in \mathcal{P}^a(G) : [\alpha_b\bullet\beta_b]_\equiv \in Q^a_\equiv$.

Proof. We take an arbitrary string $\alpha_b \in \mathcal{P}^a(G)$, and show that the equivalence classes of all its positions are states in $Q^a_\equiv$.

Because $\alpha_b \in \mathcal{P}^a(G)$ there exists at least one $\beta_b \in \mathcal{P}^a(G)$, such that $\alpha_b \neq \beta_b$ and $h(\alpha_b) = h(\beta_b)$. From Lemma 3 we know that $(\alpha_b, \beta_b) \in \mathcal{L}(\Pi^R_\equiv(G))$.

From Definitions 3 and 4 it follows that the path α_b through $\Gamma_\equiv(G)$ visits all states that are an equivalence class of a position in α_b. Let us call this set of states $Q^\alpha_\equiv$. From Definition 6 we can see that all these states occur as first elements in the pairs on the path (α_b, β_b) through $\Pi^R_\equiv(G)$. By Definition 7 it then holds that $Q^\alpha_\equiv \subseteq Q^a_\equiv$. □

The states that are not in $Q^a_\equiv$ therefore contain positions in parse trees of strings that are unambiguous.

Definition 8. *The set of parse trees of unambiguous strings of G that are identifiable with* $\equiv$*, is* $\mathcal{P}^u_\equiv(G) = \{\alpha_b\beta_b \mid [\alpha_b\bullet\beta_b]_\equiv \in Q^u_\equiv\}$.

This set is always a subset of $\mathcal{P}^u(G)$, as stated by the following Theorem, and illustrated by Figure 4.3.

Theorem 1 (underapproximation of $\mathcal{P}^u(G)$)**.** *For all equivalence relations* $\equiv$, $\mathcal{P}^u_\equiv(G) \subseteq \mathcal{P}^u(G)$.

Proof. We take an arbitrary string $\alpha_b\beta_b \in \mathcal{P}^u_\equiv(G)$ and prove $\alpha_b\beta_b \in \mathcal{P}^u(G)$.

By definition $[\alpha_b\bullet\beta_b]_\equiv \in Q^u_\equiv$. From $Q^u_\equiv = Q_\equiv \setminus Q^a_\equiv$ it follows that $[\alpha_b\bullet\beta_b]_\equiv \in Q_\equiv$ and $[\alpha_b\bullet\beta_b]_\equiv \notin Q^a_\equiv$. From the first we know that $\alpha_b\beta_b \in \mathcal{S}(G_b)$, and from the latter and Lemma 4 it follows that $\alpha_b\beta_b \notin \mathcal{P}^a(G)$. Therefore, because $\mathcal{P}^u(G) = \mathcal{S}(G_b) \setminus \mathcal{P}^a(G)$, it must be that $\alpha_b\beta_b \in \mathcal{P}^u(G)$. □

The positions in the states in $Q^a_\equiv$ and $Q^u_\equiv$ thus identify parse trees of respectively potentially ambiguous strings and certainly unambiguous strings. However, iterating over all positions in $\mathsf{pos}(G)$ is infeasible if this set is infinite. The used equivalence relation should therefore allow the direct identification of parse trees from the states of $\Gamma_\equiv(G)$.

For instance, a state in $\Gamma_{\mathsf{item}_0}(G)$ represents all parse trees in which a particular item appears. With this information we can identify production rules that only appear in parse trees in $\mathcal{P}^u_{\equiv}(G)$, as we will show in the next section.

4.4.2 Computation

The above definition of $Q^a_{\equiv}$ is not yet suitable for computation, because it requires the iteration over all pairs of different strings $\alpha_b\beta_b$ and $\alpha'_b\beta'_b$ in $\mathcal{L}(\Pi^R_{\equiv}(G))$, of which there can be infinitely many. We therefore need a definition that can be calculated in finite time. For this we make use of the fact that all pairs $\alpha_b\beta_b$ and $\alpha'_b\beta'_b$ have common pre- and postfixes, with different substrings in the middle. These middle substrings can differ in the following ways: they either each start with different brackets, or one of the two starts with a bracket and the other with a terminal or non-terminal. By describing these cases we get an alternative definition of the part of $\Pi^R_{\equiv}(G)$ that is covered by ambiguous path pairs.[2]

First, we need the following additional mutual accessibility relations:

$$\begin{aligned} \mathsf{maEq} \quad &= \{(q_0, q_0) \xrightarrow{(X_b, X_b)}{}^{*} (q_1, q_1) \mid q_0 \overset{X_b}{\longmapsto} q_1,\ X_b \in V'_b\}, \\ \mathsf{maDiff}_{\langle} &= \{(q_0, q_0) \xrightarrow{(\langle_i, \langle_j)}{}^{*} (q_1, q_2) \mid q_0 \overset{\langle_i}{\longmapsto} q_1 \wedge q_0 \overset{\langle_j}{\longmapsto} q_2,\ i \neq j\}, \end{aligned}$$

Then we can define all path pairs of the above mentioned forms with the following relations:

$$\mathsf{maEq}^* \cdot \mathsf{maDiff}_{\langle} \cdot \mathsf{nma}^+ \tag{4.3}$$

$$\mathsf{maEq}^* \cdot (\mathsf{nmaDl} \cup \mathsf{nmaCl})^+ \cdot \mathsf{nmaS} \cdot \mathsf{nma}^+ \tag{4.4}$$

$$\mathsf{maEq}^* \cdot (\mathsf{nmaDr} \cup \mathsf{nmaCr})^+ \cdot \mathsf{nmaS} \cdot \mathsf{nma}^+ \tag{4.5}$$

The union of these three relations forms a computable description of all ambiguous path pairs through $\Pi^R_{\equiv}(G)$. Note that it suffices to only specify the different beginnings of the middle substrings. We could also explicitely specify the common postfixes, which would result in nine different relations.

With this alternative definition we can gather $Q^a_{\equiv}$ by traversing the above relations from q^2_s and collect all states in encountered state pairs. This process is linear in the number of edges in $\Pi^R_{\equiv}(G)$ (see [SSS88] Chapter 2), which is worst case $\mathcal{O}(|R_{\equiv}|^2)$.

4.5 Harmless Production Rules

In this section we show how we can use $Q^a_{\equiv}$, the part of $\Gamma_{\equiv}(G)$ that is covered by ambiguous path pairs, to identify production rules that do not contribute to the ambiguity of G. These are the production rules that can never occur in parse trees of ambiguous strings. We call them *harmless production rules*.

[2]We thank the anonymous reviewers of ICTAC 2010 for this idea.

4.5.1 Finding Harmless Production Rules

A production rule is certainly harmless if it is only used in parse trees in $\mathcal{P}^u_{\equiv}(G)$. We should therefore search for productions that are never used on ambiguous path pairs of $\Pi^R_{\equiv}(G)$ that describe valid parse trees in G. We can find them by looking at the items of the positions in the states of $Q^a_{\equiv}$. If not all items of a production rule are used then the rule cannot be used in a valid string in $\mathcal{P}^a(G)$ (Lemma 2), and we know it is harmless.

Definition 9. *The set of items used on the ambiguous path pairs through* $\Pi^R_{\equiv}(G)$ *is* $I^a_{\equiv} = \{A \to \alpha\bullet\beta \mid \exists q \in Q^a_{\equiv} : \exists \gamma_b\bullet\delta_b \in q : A \to \alpha\bullet\beta = \mathsf{item}(\gamma_b\bullet\delta_b)\}$.

With it we can identify production rules of which all items are used:

Definition 10. *The set of* potentially harmful production rules *of G, identifiable from* $\Pi^R_{\equiv}(G)$*, is* $P_{\text{hf}} = \{A \to \alpha \mid \mathsf{proditems}(A \to \alpha) \subseteq I^a_{\equiv}\}$.

Because of the approximation it is uncertain whether or not they can really be used to form valid parse trees of ambiguous strings. Nevertheless, all the other productions in P will certainly not appear in parse trees of ambiguous strings.

Definition 11. *The set of* harmless production rules *of G, identifiable from* $\Pi^R_{\equiv}(G)$*, is* $P_{\text{hl}} = P \setminus P_{\text{hf}}$.

Theorem 2 (harmless production rules)**.** $\forall p \in P_{\text{hl}} : \neg\exists \alpha_b\langle_i\beta_b\rangle_i\gamma_b \in \mathcal{P}^a(G) : i = \mathsf{pid}(p)$.

Proof. We take an arbitrary production rule $p \in P_{\text{hl}}$ and an arbitrary parse tree $\delta_b = \alpha_b\langle_i\beta_b\rangle_i\gamma_b$ such that $i = \mathsf{pid}(p)$, and prove that $\delta_b \notin \mathcal{P}^a(G)$.

Because $p \notin P_{\text{hf}}$ there is (at least) one item of p that is not in $I^a_{\equiv}$, let us call this item m. According to Lemma 2 there must be a position $\eta_b\bullet\theta_b$ in δ_b such that $\mathsf{item}(\eta_b\bullet\theta_b) = m$. From $m \notin I^a_{\equiv}$ it follows that $[\eta_b\bullet\theta_b]_{\equiv} \notin Q^a_{\equiv}$. With Lemma 4 we can then conclude that $\delta_b \notin \mathcal{P}^a(G)$. □

Example 4.6 in Section 4.7 shows finding P_{hl} for a small grammar.

4.5.2 Complexity

Finding P_{hf} comes down to building $\Pi^R_{\equiv}(G)$, finding $Q^a_{\equiv}$, and enumerating all positions in all classes in $Q^a_{\equiv}$ to find $I^a_{\equiv}$. The number of these classes is finite, but the number of positions might not be. It would therefore be convenient if the definition of the chosen equivalence relation could be used to collect $I^a_{\equiv}$ in finitely many steps. With the item_0 relation this is possible, because all the positions in a class are all in the same item.

Constructing $\Pi^R_{\mathsf{item}_0}(G)$ can be done in $\mathcal{O}(|G|)$ (see [Sch07b]), where $|G|$ is the number of items of G. After that, $Q^a_{\mathsf{item}_0}$ can be gathered in $\mathcal{O}(|G|^2)$ (see Section 4.4.2), because $|R_{\mathsf{item}_0}|$ is linear with $|G|$. Then, constructing both $I^a_{\mathsf{item}_0}$ and P_{hf} is also linear with $|G|$. The worst case complexity of finding P_{hf} with item_0 is therefore $\mathcal{O}(|G|^2)$.

4.5.3 Grammar Reconstruction

Finding P_{hl} can be very helpful information for the grammar developer. Also, P_{hf} represents a smaller grammar that can be checked again more easily to find the true origins of ambiguity. However, the reachability and productivity properties of this smaller grammar might be violated because of the removed productions in P_{hl}. These must be restored first before the grammar can be checked again.

If the reachability of the grammar is broken this is not a problem. In fact, all productions in P_{hf} that are not reachable from S anymore are harmless. Since if a production p is not reachable from S, it means that p can only appear in parse trees that also contain productions from P_{hl}. These trees are therefore not in $\mathcal{P}^a(G)$, which implies that p is harmless. All unreachable productions in P_{hf} can thus safely be discarded.

The productivity of the grammar can be restored by adding new productions and terminals. We must prevent introducing new ambiguities in this process. From P_{hf} we can create a new grammar G' by constructing:[3]

1. The set of defined nonterminals of P_{hf}: $N_{\mathrm{def}} = \{A \mid A \to \alpha \in P_{\mathrm{hf}}\}$.
2. The used but undefined nonterminals of P_{hf}:
   $N_{\mathrm{undef}} = \{B \mid A \to \alpha B \beta \in P_{\mathrm{hf}}\} \backslash N_{\mathrm{def}}$.
3. The unproductive nonterminals:
   $N_{\mathrm{unpr}} = \{A \mid A \in N_{\mathrm{def}}, \neg\exists u : A \Longrightarrow^* u$ using only productions in $P_{\mathrm{hf}}\}$.
4. New terminal symbols t_A for each nonterminal $A \in N_{\mathrm{undef}} \cup N_{\mathrm{unpr}}$.
5. Productions to complete the unproductive and undefined nonterminals:
   $P' = P_{\mathrm{hf}} \cup \{A \to t_A \mid A \in N_{\mathrm{undef}} \cup N_{\mathrm{unpr}}\}$.
6. The new set of terminal symbols: $T' = \{a \mid A \to \beta a \gamma \in P'\}$.
7. Finally, the new grammar: $G' = (N_{\mathrm{def}} \cup N_{\mathrm{undef}}, T', P', S)$.

At step 5 we create the new productions that restore the productivity of the grammar, without introducing new ambiguities. Every new production contains a new unique terminal to make sure it cannot be used to form a new parse tree for an existing string in $\mathcal{L}(G)$.

4.6 Noncanonical Unambiguity Test

In this section we explain the Noncanonical Unambiguity (NU) Test [Sch07b], which is more precise than the Regular Unambiguity Test. It enables the identification of a larger set of irrelevant parse trees, namely the ones that are not in the ambiguous core of G. From these we can also identify a larger set of harmless production rules.

[3]This algorithm is essentially the same as the one presented in Section 3.2.6, with the only difference that this version does not preserve the distribution of string lengths in $\mathcal{L}(G')$.

4.6.1 Improving the Regular Unambiguity Test

The regular position pair automaton described in Section 4.3 checks all pairs of paths through a position automaton for ambiguity. However, it also checks some spurious paths that are unnecessary for identifying the ambiguity of a grammar.

These are the path pairs that derive the same unambiguous substring for a certain nonterminal. We can ignore these paths because in this situation there are also two paths in which the nonterminal was shifted instead of derived. For instance, consider paths $\langle_1\langle_2\langle_3 a\rangle_3\alpha_b\rangle_2\rangle_1$ and $\langle_1\langle_2\langle_3 a\rangle_3\beta_b\rangle_2\rangle_1$. If they form a pair in $\mathcal{L}(\Pi^R_{\equiv}(G))$ then the shorter paths $\langle_1\langle_2 A\alpha_b\rangle_2\rangle_1$ and $\langle_1\langle_2 A\beta_b\rangle_2\rangle_1$ will too (considering $A \to \langle_3 a\rangle_3 \in P_b$). In addition, if the first two paths form an ambiguous path pair, then these latter two will also, because $\langle_3 a\rangle_3$ does not contribute to the ambiguity. In this case we prefer the latter paths because they describe smaller parse trees than the first paths.

4.6.2 Noncanonical Position Pair Automaton

To avoid common unambiguous substrings we should only allow path pairs to take identical reduce transitions if they do not share the same substring since their last derives. To keep track of this property we add two extra boolean flags c_0 and c_1 to the position pairs. These flags tell for each position in a pair whether or not its path has been in conflict with the other, meaning it has taken different reduce steps as the other path since its last derive. A value of 0 means this has not occurred yet, and we are thus allowed to ignore an identical reduce transition.

All start pairs have both flags set to 0, and every derive step resets the flag of a path to 0. The flag is set to 1 if a path takes a *conflicting* reduce step, which occurs if the other path does not follow this reduce at the same time (for instance $\rangle_2$ in the parse trees $\langle_1\langle_2\langle_3 a\rangle_3\rangle_2\rangle_1$ and $\langle_1\langle_2\langle_3 a\rangle_3\rangle_1$). We use the predicate confl (called eff by Schmitz) to identify a situation like that.

$$\mathsf{confl}(q,i) = \exists u \in T^*_{\langle} : q \overset{u}{\longmapsto}{}^{*}\, q_f \vee (\exists q' \in Q_{\equiv}, X \in V \cup T_{\rangle} : X \neq \rangle_i, q \overset{uX}{\longmapsto}{}^{+}\, q') \quad (4.6)$$

It tells whether there is another shift or reduce transition other than $\rangle_i$ possible from q, ignoring $\langle$ steps, or if q is at the end of the automaton.

Definition 12. *The* noncanonical *position pair automaton* $\Pi^N_{\equiv}(G)$ *of* $\Gamma_{\equiv}(G)$ *is the tuple* $(Q^p, V_b'^2, \mathsf{nma}, (q_s,0)^2, (q_f,1)^2)$, *where* $Q^p = (Q_{\equiv}\times\mathbb{B})^2$, *and* nma *over* $Q^p \times V_b'^2 \times Q^p$ *is the* noncanonical *mutual accessibility relation, defined as the union of the following subrelations:*

$$\begin{aligned}
\mathsf{nmaDl} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(\langle_i,\varepsilon)} (q_2,q_1)0,c_1 \mid q_0 \overset{\langle_i}{\longmapsto} q_2\},\\
\mathsf{nmaDr} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(\varepsilon,\langle_i)} (q_0,q_3)c_0,0 \mid q_1 \overset{\langle_i}{\longmapsto} q_3\},\\
\mathsf{nmaS} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(X,X)} (q_2,q_3)c_0,c_1 \mid q_0 \overset{X}{\longmapsto} q_2,\ q_1 \overset{X}{\longmapsto} q_3,\ X \in V'\},\\
\mathsf{nmaCl} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(\rangle_i,\varepsilon)} (q_2,q_1)1,c_1 \mid q_0 \overset{\rangle_i}{\longmapsto} q_2,\ \mathsf{confl}(q_1,i)\},\\
\mathsf{nmaCr} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(\varepsilon,\rangle_i)} (q_0,q_3)c_0,1 \mid q_1 \overset{\rangle_i}{\longmapsto} q_3,\ \mathsf{confl}(q_0,i)\},\\
\mathsf{nmaR} &= \{(q_0,q_1)c_0,c_1 \xrightarrow{(\rangle_i,\rangle_i)} (q_2,q_3)1,1 \mid q_0 \overset{\rangle_i}{\longmapsto} q_2,\ q_1 \overset{\rangle_i}{\longmapsto} q_3,\ c_0 \vee c_1\}.
\end{aligned}$$

As with $\Pi^R_{\equiv}(G)$, the language of $\Pi^N_{\equiv}(G)$ describes ambiguous path pairs through $\Gamma_{\equiv}(G)$. The difference is that $\mathcal{L}(\Pi^N_{\equiv}(G))$ does not include path pairs without conflicting reductions. Therefore $\mathcal{L}(\Pi^N_{\equiv}(G)) \subseteq \mathcal{L}(\Pi^R_{\equiv}(G))$. Nevertheless, $\Pi^N_{\equiv}(G)$ does at least describe all the *core* parse trees in $\mathcal{C}^a(G)$:

Theorem 3 (core ambiguous path pairs). $\forall \alpha_b, \beta_b \in \mathcal{C}^a(G) : \alpha_b \neq \beta_b \wedge h(\alpha_b) = h(\beta_b) \Rightarrow (\alpha_b, \beta_b) \in \mathcal{L}(\Pi^N_{\equiv}(G))$.

Proof. We take an arbitrary string $\alpha_b \in \mathcal{C}^a(G)$. Then there is at least one $\beta_b \in \mathcal{C}^a(G)$ such that $\alpha_b \neq \beta_b$ and $h(\alpha_b) = h(\beta_b)$. We show that $(\alpha_b, \beta_b) \in \mathcal{L}(\Pi^N_{\equiv}(G))$.

Because $\mathcal{C}^a(G) \subseteq \mathcal{P}^a(G)$ we know from Lemma 3 that $\alpha_b, \beta_b \in \mathcal{L}(\Gamma_{\equiv}(G))$ and $(\alpha_b, \beta_b) \in \mathcal{L}(\Pi^R_{\equiv}(G))$. To prove that (α_b, β_b) is also in $\mathcal{L}(\Pi^N_{\equiv}(G))$ we show that the extra restrictions of nma over ma do not apply for (α_b, β_b). We distinguish the following cases:

- nmaDl, nmaDr and nmaS: These relations are similar to respectively maDl, maDr and maS, and have no additional restrictions.
- nmaR: One nmaR transition is similar to taking two consecutive maRl and maRr transitions with the same $\rangle_i$, with the extra restriction that at least one boolean flag is 1. We will show that it is not possible to reach a pair with both flags 0 if both paths need to read the same $\rangle_i$.

  If we would reach a pair like that this means we have not followed any $\langle$ or $\rangle$ steps since the two $\langle_i$ steps that match the $\rangle_i$s. Reading a $\langle_j$ step after $\langle_i$ on any path would set a flag to 0, but then we would also have read a matching $\rangle_j$ before $\rangle_i$ because $\langle$s and $\rangle$s are always balanced. However, this would have set at least one flag to 1.

  The only steps we thus could have taken since the $\langle_i$s are shifts on both paths of the same terminal or nonterminal symbols. But then we have an identical substring $\langle_i \gamma \rangle_i$ in both strings α_b and β_b that represents the same substring in $h(\alpha_b)$ and $h(\beta_b)$. This means $\alpha_b, \beta_b \notin \mathcal{C}^a(G)$, because if we "underive" $\langle_i \gamma \rangle_i$ — substituting it with the nonterminal at the right hand side of production i — in α_b an β_b we still get two parse trees of an ambiguous string. Therefore, nmaR can always be followed on path (α_b, β_b).
- nmaCl and nmaCr: These relations are similar to maRl and maRr, with the added confl restrictions. Above we saw that if both paths reach identical $\rangle_i$ symbols we can read them with nmaR. In all other cases we can read $\rangle_i$ symbols with either nmaCl and nmaCr, because then confl will be true: if we ignore $\langle$ symbols, the $\rangle_i$ symbol will eventually come into conflict with another $\rangle_j$ or X symbol of the other path, or the other path is already at its end.

All $\langle$ symbols in α_b and β_b can thus be read through $\Pi^N_{\equiv}(G)$ with nmaDl or nmaDr transitions, the $X \in V$ symbols can be read synchronously with nmaS, and the $\rangle$s with nmaCl, nmaCr or nmaR. Therefore $(\alpha_b, \beta_b) \in \mathcal{L}(\Pi^N_{\equiv}(G))$. □

Theorem 3 shows that if G is ambiguous — that is $\mathcal{C}^a(G)$ is non-empty — $\mathcal{L}(\Pi^N_{\equiv}(G))$ is also non-empty. This means that if $\mathcal{L}(\Pi^N_{\equiv}(G))$ is empty, G is unambiguous.

4.6.3 Effects on Identifying Harmless Production Rules

The new nma relation enables our parse tree identification algorithm of Section 4.4 to potentially identify a larger set of irrelevant parse trees, namely $\mathcal{C}^u(G)$. These trees might be ambiguous, but this is not a problem because we are interested in finding the trees of the smallest possible ambiguous sentential forms of G, namely the ones in $\mathcal{C}^a(G)$. The trees in $\mathcal{C}^a(G)$ are sufficient to prove the ambiguity or unambiguity of a grammar (Lemma 1).

Definition 13. *Given $Q^u_\equiv$ from $\Pi^N_\equiv(G)$, the set of parse trees not in the ambiguous core of G, identifiable with $\equiv$, is $\mathcal{C}^u_\equiv(G) = \{\alpha_b\beta_b \mid \exists q \in Q^u_\equiv, \alpha_b\bullet\beta_b \in q\}$.*

Theorem 4 (overapproximation of ambiguous core)**.** *For all equivalence relations $\equiv$, $\mathcal{C}^u_\equiv(G) \subseteq \mathcal{C}^u(G)$.*

The set of harmless production rules that can be identified with $\Pi^N_\equiv(G)$ is also potentially larger. It might include rules that can be used in parse trees of ambiguous strings, but not in parse trees in $\mathcal{C}^a(G)$. Therefore they are not vital for the ambiguity of G.

Definition 14. *Given $Q^a_\equiv$ and $I^a_\equiv$ from $\Pi^N_\equiv(G)$, the set of harmless productions of G, identifiable from $\Pi^N_\equiv(G)$, is $P'_{\mathrm{hl}} = P \setminus \{A \to \alpha \mid \mathsf{proditems}(A \to \alpha) \subseteq I^a_\equiv\}$.*

Theorem 5 (harmless production rules of $\Pi^N_\equiv(G)$)**.** $\forall p \in P'_{\mathrm{hl}} : \neg\exists \alpha_b\langle_i\beta_b\rangle_i\gamma_b \in \mathcal{C}^a(G) : i = \mathsf{pid}(p)$.

4.7 Excluding Parse Trees Iteratively

Our approach for the identification of parse trees of unambiguous strings is most useful if applied in an iterative setting. By checking the remainder of the potentially ambiguous parse trees again, there is possibly less interference of the trees during approximation. This could result in less ambiguous path pairs in the position pair automaton. We could then exclude a larger set of parse trees and production rules.

Example 4.6. The grammar below (4.7) is unambiguous but needs two iterations of the NU Test with item_0 to detect this. At first, $\Pi^N_{\mathsf{item}_0}(G)$ contains only the ambiguous path pair $\langle_1\langle_4 c\rangle_4\rangle_1$ and $\langle_2\langle_5\langle_6 c\rangle_6\rangle_3\rangle_1$. The first path describes a valid parse tree, but the second does not. From $B \to \bullet Cb$ it derives to $C \to \bullet c$, but from $C \to c\bullet$ it reduces to $A \to aC\bullet$. Therefore productions 2, 5 and 3 are only used partially, and they are thus harmless. After removing them and checking the reconstructed grammar again there are no ambiguous path pairs anymore.

$$1 : S \to A,\ 2 : S \to B,\ 3 : A \to aC,\ 4 : A \to c,\ 5 : B \to Cb,\ 6 : C \to c \tag{4.7}$$

We can gain even higher precision by choosing a new equivalence relation with each iteration, similar to the approach of counterexample-guided abstraction refinement [CGJ$^+$00] described by Clarke *et al.* If with each step $\Gamma_\equiv(G)$ better approximates $\mathcal{S}(G_b)$, we might end up with only the parse trees in $\mathcal{P}^u(G)$. Unfortunately, the ambiguity problem is undecidable, and this process does not necessarily have to terminate. There might be an infinite number of equivalence relations that yield a finite number of equivalence classes. Or at some point we might need to resort to equivalence relations that do not yield a finite graph. Therefore, the iteration has to stop at a certain moment, and we can continue with an exhaustive search of the remaining parse trees.

In the end this exhaustive searching is the most practical, because it can point out the exact parse trees of ambiguous strings. A drawback of this approach is its exponential complexity. Nevertheless, excluding sets of parse trees beforehand can reduce its search space significantly, as we have seen in the previous chapter.

4.8 Conclusions

We have shown how the Regular Unambiguity Test and Noncanonical Unambiguity Test can be extended to conservatively identify parse trees of unambiguous strings. From these trees we can identify harmless production rules. These are the production rules that do not contribute to the ambiguity of a grammar. With the RU Test our approach is able to find production rules that can only be used to derive unambiguous strings. With the NU Test it can also find productions that can only be used to derive unambiguous substrings of ambiguous strings. This information is already very useful for a grammar developer, but it can also be used to significantly reduce the search space of other ambiguity detection methods.

Chapter 5

Scaling to Scannerless

In this chapter we present a set of extensions to our grammar filtering technique to make it suitable for character-level grammars. Character-level grammars are used for generating scannerless parsers, and are typically more complex than token-based grammars. We present several character-level specific extensions that take disambiguation filters into account, but also a general precision improving extension called grammar unfolding. We test an implementation of our approach on a series of real world programming languages and measure the improvements in sentence generation times. Although the gains are not as large as in Chapter 3, our technique proves very useful on most grammars.

5.1 Introduction

5.1.1 Background

Scannerless generalized parsers [vdBSVV02], generated from character-level context-free grammars, serve two particular goals in textual language engineering: parsing legacy languages and parsing language embeddings. We want to parse legacy languages when we construct reverse engineering and reengineering tools to help mitigating cost-of-ownership of legacy source code. The syntax of legacy programming languages frequently does not fit the standard scanner-parser dichotomy. This is witnessed by languages that do not reserve keywords from identifiers (PL/I) or do not always apply "longest match" when selecting a token class (Pascal). For such languages we may generate a scannerless generalized parser that will deal with such idiosyncrasies correctly.

This chapter was published in the proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011) [BKV11]. It was co-authored by Paul Klint and Jurgen Vinju. For completeness, Section 5.4 contains two additional algorithms that were not included in the published paper.

Language embeddings need different lexical syntax for different parts of a composed language. Examples are COBOL with embedded SQL, or Aspect/J with embedded Java. The comment conventions may differ, different sets of identifiers may be reserved as keywords and indeed identifiers may be comprised of different sets of characters, depending on whether the current context is the "host language" or the embedded "guest language". Language embeddings are becoming increasingly popular, possibly due to the belief that one should select the right tool for each job. A character-level grammar can be very convenient to implement a parser for such a combined language [BV04]. The reason is that the particular nesting of non-terminals between the host language and the guest language defines where the different lexical syntaxes are applicable. The lexical ambiguity introduced by the language embedding is therefore a non-issue for a scannerless parser. There is no need to program state switches in a scanner [VS07], to use scanner non-determinism [AH01], or to use any other kind of (ad-hoc) programming solution.

Using a character-level grammar and a generated scannerless parser results in more declarative BNF grammars which may be maintained more easily than partially hand-written parsers [KVW10]. It is, however, undeniable that character-level grammars are more complex than classical grammars since all lexical aspects of a language have to be specified in full detail. The character-level grammar contains more production rules, which may contain errors or introduce ambiguity. In the absence of lexical disambiguation heuristics, such as "prefer keywords" and "longest match", a character-level grammar may contain many ambiguities that need resolving. Ergo, character-level grammars lead to more declarative grammar specifications but increase the risk of ambiguities and makes automated ambiguity detection harder.

5.1.2 Contributions and Roadmap

We introduce new techniques for scaling ambiguity detection methods to the complexity that is present in character-level grammars for real programming languages. Our point of departure is a fast ambiguity detection framework that combines a grammar approximation stage with a sentence generation stage — see Chapter 3. The approximation is used to split a grammar into a set of rules that certainly do not contribute to ambiguity and a set that might. The latter is then fed to a sentence generator to obtain a clear and precise ambiguity report. We sketch this global framework (Section 5.2) and then describe our baseline algorithm (Section 5.4). The correctness of this framework has been established in Chapter 4 and is not further discussed here.

We present several extensions to the baseline algorithm to make it suitable for character-level grammars (Section 5.5). First, we consider character classes as shiftable symbols, instead of treating every character as a separate token. This is necessary to deal with the increased lexical complexity of character-level grammars. Second, we make use of disambiguation filters [vdBSVV02] to deal with issues such as keyword reservation and longest match. These filters are used for precision improvements at the approximation stage, and also improve the run-time efficiency of the sentence generation stage by preventing spurious explorations of the grammar. Third, we use grammar unfolding as a general optimization technique (Section 5.6). This is necessary for certain character-level grammars but is also generally applicable. At

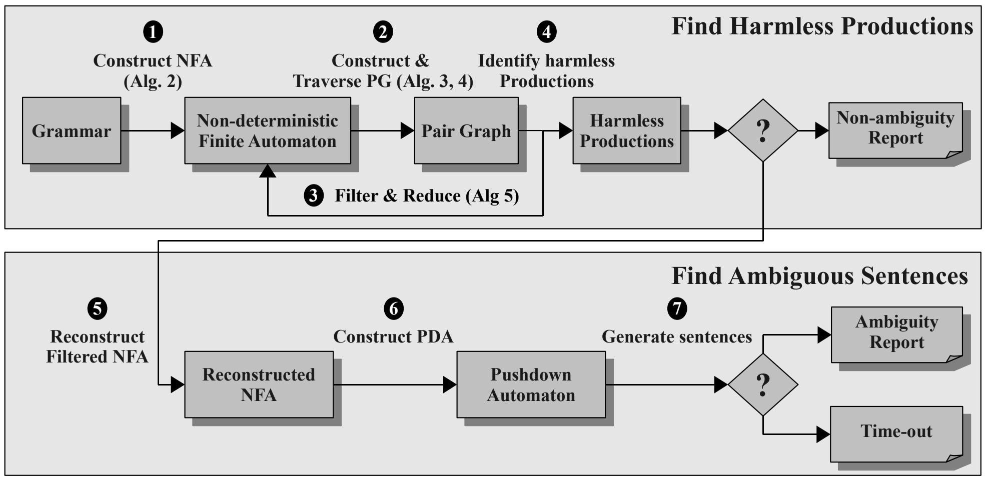


Figure 5.1: Baseline architecture for fast ambiguity detection.

a certain cost, it allows us to more effectively identify the parts of a grammar that do not contribute to ambiguity.

We have selected a set of real character-level grammars and measure the speed, footprint and accuracy of the various algorithms (Section 5.7). The result is that the total cost of ambiguity detection is dramatically reduced for these real grammars.

5.2 The Ambiguity Detection Framework

5.2.1 The Framework

Our starting point is an ambiguity detection framework called AMBIDEXTER, which combines an extension of the approximative Noncanonical Unambiguity Test [Sch07b] with an exhaustive sentence generator comparable to [Sch01]. The former is used to split a grammar into a set of harmless rules and a set of rules that may contribute to ambiguity. The latter is used to generate derivations based on the potentially ambiguous rules and produce understandable ambiguity reports.

Figure 5.1 displays the architecture of the baseline algorithm which consists of seven steps, ultimately resulting in a non-ambiguity report, an ambiguity report, or a time-out.

1. In step ❶ the grammar is bracketed, starting and ending each rule with a unique terminal. The language of the bracketed grammar represents all parse trees of the original grammar. In this same step an NFA is constructed that over-approximates the language of the bracketed grammar. This NFA allows us to find strings with multiple parse trees, by approximation, but in finite time.

2. In step ❷ a data-structure called a Pair Graph (PG) is constructed from the NFA. This PG represents all pairs of two different paths through the NFA that produce the same sentence, i.e., potentially ambiguous derivations. During construction, the PG is immediately traversed to identify the part of the NFA that is covered by the potentially ambiguous derivations.

3. In step ❸ we filter the uncovered parts from the NFA and clean up dead ends. This might filter potentially ambiguous derivations from the NFA that are actually false positives, so we reconstruct the PG again to find more uncovered parts. This process is repeated until the NFA cannot be reduced any further.

4. In step ❹ we use the filtered NFA to identify harmless productions. These are the productions that are not used anymore in the NFA. If the NFA is completely filtered then all productions are harmless and the grammar is unambiguous.

5. In step ❺ we prepare the filtered NFA to be used for sentence generation. Due to the removal of states not all paths produce terminal only sentences anymore. We therefore reconstruct the NFA by adding new terminal producing paths.

   In our original approach we generated sentences based on the remaining potentially harmful productions. However, by immediately using the filtered NFA we retain more precision, because the NFA is a more precise description of the potentially ambiguous derivations than a reconstructed grammar.

6. In step ❻ we convert the NFA into a pushdown automaton (PDA) which enables faster sentence generation in the next step.

7. The final step (❼) produces ambiguous strings, including their derivations, to report to the user. This may not terminate, since most context-free grammars generate infinite languages; we need to stop after a certain limited time. All ambiguity that was detected before the time limit is reported to the user.

It was shown in Chapter 4 that the calculations employed in this architecture are correct, and in Chapter 3 that indeed the efficiency of ambiguity detection can be improved considerably by first filtering harmless productions. However, the baseline algorithm is not suitable for character-level grammars since it is unable to handle their increased complexity and it will still find ambiguities that are already solved. It can even lead to incorrect results because it cannot deal with the non-context-free behaviour of follow restrictions. In this chapter we identify several opportunities for optimization and correction:

- We filter nodes and edges in the NFA and PG representations in order to make use of disambiguation information that is found in character-level grammars (Section 5.5).
- We “unfold” selected parts of a grammar to handle the increased lexical complexity of character-level grammars (Section 5.6).

For the sake of presentation we have separated the discussion of the baseline algorithm (Section 5.4), the filtering (Section 5.5), and the unfolding (Section 5.6), but it is important to note that these optimizations are not orthogonal.

5.2.2 Notational Preliminaries

A *context-free grammar* G is a four-tuple (N, T, P, S) where N is the set of non-terminals, T the set of terminals, P the set of productions over $N \times (N \cup T)^*$, and S is the start symbol. V is defined as $N \cup T$. We use $A, B, C, \ldots$ to denote non-terminals, $X, Y, Z, \ldots$ for either terminals or non-terminals, $u, v, w, \ldots$ for sentences: strings of T^*, and $\alpha, \beta, \gamma, \ldots$ for sentential forms: strings over V^*.

A production (A, α) in P is written as $A \rightarrow \alpha$. A grammar is augmented by adding an extra non-terminal symbol S', a terminal symbol *\$* and a production $S' \rightarrow S\$$, and making S' the start symbol. We use the function $\mathsf{pid}: P \rightarrow \mathbb{N}$ to relate each production to a unique number. An *item* indicates a position in a production rule with a dot, for instance as $S \rightarrow A \bullet BC$. We use I to denote the set of all items of G.

The relation $\Longrightarrow$ denotes derivation. We say $\alpha B \gamma$ directly derives $\alpha\beta\gamma$, written as $\alpha B \gamma \Longrightarrow \alpha\beta\gamma$ if a production rule $B \rightarrow \beta$ exists in P. The symbol $\Longrightarrow^*$ means "derives in zero or more steps". The *language* of G, denoted $\mathcal{L}(G)$, is the set of all sentences derivable from S. We use $\mathcal{S}(G)$ to denote the *sentential language* of G: the set of all sentential forms derivable from S.

From a grammar G we can create a *bracketed grammar* G_b by surrounding each production rule with unique bracket terminals [GH67]. The bracketed grammar of G is defined as $G_b = (N, T_b, P_b, S)$ where T_b is the set of terminals and brackets, defined as $T_b = T \cup T_\langle \cup T_\rangle$, $T_\langle = \{\langle_i \mid i \in \mathbb{N}\}$, $T_\rangle = \{\rangle_i \mid i \in \mathbb{N}\}$, and $P_b = \{A \rightarrow \langle_i \alpha \rangle_i \mid A \rightarrow \alpha \in P,\ i = \mathsf{pid}(A \rightarrow \alpha)\}$. V_b is defined as $N \cup T_b$. We use the function $\mathsf{bracketP}$ to map a bracket to its corresponding production, and $\mathsf{bracketN}$ to map a bracket to its production's left hand side non-terminal. They are defined as $\mathsf{bracketP}(\langle_i) = \mathsf{bracketP}(\rangle_i) = A \rightarrow \alpha$ iff $\mathsf{pid}(A \rightarrow \alpha) = i$, and $\mathsf{bracketN}(\langle_i) = \mathsf{bracketN}(\rangle_i) = A$ iff $\exists A \rightarrow \alpha \in P$, $\mathsf{pid}(A \rightarrow \alpha) = i$. A string in the language of G_b describes a parse tree of G. Therefore, if two unique strings exist in $\mathcal{L}(G_b)$ that become identical after removing their brackets, G is ambiguous.

5.3 Character-Level Grammars

Now we introduce character-level grammars as used for scannerless parsing. Character-level grammars differ from conventional grammars in various ways. They define their syntax all the way down to the character level, without separate token definitions. For convenience, sets of characters are used in the production rules, so-called *character classes*. Regular list constructs can be used to handle repetition, like in EBNF. Also, additional constructs are needed to specify the disambiguation that is normally done by the scanner, so called disambiguation filters [KV94]. Typical disambiguation filters for character-level grammars are follow restrictions and rejects [vdBSVV02]. Follow restrictions are used to enforce longest match of non-terminals such as identifiers and comments. Rejects are typically used for keyword reservation. Other commonly used disambiguation filters are declarations to specify operator priority and associativity, so these do not have to be encoded manually into the production rules.

```
Declaration ::= Specifiers Ws? Identifier Ws? ";"      (1)
Specifiers  ::= Specifiers Ws? Specifier               (2)
Specifiers  ::= Specifier                              (3)
Specifier   ::= Identifier | "int" | "float" | ...     (4)

Identifier  ::= [a-z]+                                 (5)
Identifier  ::= Keyword  { reject }                    (6)
Keyword     ::= "int" | "float" | ...                  (7)
Ws          ::= [\ \t\n]+                              (8)

Identifier  -/- [a-z]                                  (9)
"int"       -/- [a-z]                                  (10)
"float"     -/- [a-z]                                  (11)
```

Figure 5.2: Example character-level grammar for C-style declarations.

5.3.1 Example

Figure 5.2 shows an excerpt of a character-level grammar, written in SDF2 [HHKR89, Vis97]. The excerpt describes syntax for C-style variable declarations. A `Declaration` statement consists of a list of `Specifiers` followed by an `Identifier` and a semicolon, separated by whitespace (Rule 1). A `Specifier` is either a predefined type like `int` or `float`, or a user-defined type represented by an `Identifier` (Rule 4). At Rule 5 we see the use of the character class `[a-z]` to specify the syntax of `Identifier`.

The grammar contains both rejects and follow restrictions to disambiguate the lexical syntax. The `{reject}` annotation at Rule 6 declares that reserved keywords of the language cannot be recognized as an `Identifier`. The follow restriction statements at Rules 9–11 declare that any substring that is followed by a character in the range `[a-z]` cannot be recognized as an `Identifier` or keyword. This prevents the situation where a single `Specifier`, for instance an `Identifier` of two or more characters, can also be recognized as a list of multiple shorter `Specifiers`. Basically, the follow restrictions enforce that `Specifiers` should be separated by whitespace.

5.3.2 Definition

We define a *character-level context-free grammar* $G^{\mathcal{C}}$ as the eight-tuple $(N, T, \mathcal{C}, P^{\mathcal{C}}, S, R_D, R_F, R_R)$ where $\mathcal{C} \subset N$ is the set of character classes over $\mathcal{P}(T)$, $P^{\mathcal{C}}$ the set of production rules over $N \times N^*$, R_D the set of derivation restrictions, R_F the set of follow restrictions, R_R the set of rejects.

A *character class* non-terminal is a finite set of terminals in T. For each of its elements it has an implicit production with a single terminal right hand side. We can write $\alpha C \beta \Longrightarrow \alpha c \beta$ iff $C \in \mathcal{C}$ and $c \in C$.

The *derivation restrictions* R_D restrict the application of productions in the context of others. They can be used to express priority and associativity of operators. We define R_D as a relation over $I \times P^{\mathcal{C}}$. Recall that we have defined I as the set of all items of a grammar. An element $(A \to \alpha \bullet B \gamma, B \to \beta)$ in R_D means that we are not allowed to derive a B

Algorithm 1 Base algorithm for filtering the NFA and finding harmless productions.

```
function FIND-HARMLESS-PRODUCTIONS() =
    $(Q, R)$ = BUILD-NFA()
    do
        nfasize = $|Q|$
        $Q_a$ = TRAVERSE-PATH-PAIRS$(Q, R)$ // returns items used on conflicting path pairs
        $(Q, R)$ = FILTER-NFA$(Q, R, Q_a)$ // removes unused items and prunes dead ends
    while nfasize $\neq |Q|$
    return $P \setminus$ USED-PRODUCTIONS$(Q)$
```

non-terminal with production $B \to \beta$, if it originated from the B following the dot in the production $A \to \alpha B \gamma$.

The *follow restrictions* R_F restrict the derivation of substrings following a certain non-terminal. We define them as a relation over $N \times T^+$. An element (A, u) in this relation means that during the derivation of a string $\beta A \gamma$, γ can not be derived into a string of form $u\delta$.

The *rejects* R_R restrict the language of a certain non-terminal, by subtracting the language of another non-terminal from it. We define them as a relation over $N \times N$. An element (A, B) means that during the derivation of a string $\alpha A \beta$, A cannot be derived to a string that is also derivable from B.

5.4 Baseline Algorithm

In this section we explain the baseline algorithm for finding harmless production rules and ambiguous counter-examples. The presentation follows the steps shown in Figure 5.1. Algorithm 1 gives an overview of the first stage of finding harmless productions. All functions operate on a fixed input grammar $G = (N, T, P, S)$ to which they have global read access.

5.4.1 Step 1: NFA Construction

The first step of the baseline algorithm is to construct the NFA from the grammar. It is defined by the tuple (Q, R) where Q is the set of states and R is the transition relation over $Q \times V_b \times Q$. Edges in R are denoted by $Q \overset{V_b}{\longmapsto} Q$. The states of the NFA are the items of G. The start state is $S' \to \bullet S\$$ and the end state is $S' \to S\$\bullet$. There are three types of transitions:

- *Shifts* of (non-)terminal symbols to advance to a production's next item,
- *Derives* from items with the dot before a non-terminal to the first item of one of the non-terminal's productions, labeled over $T_\langle$,
- *Reduces* from items with the dot at the end, to items with the dot after the non-terminal that is at the first item's production's left hand side, labeled over $T_\rangle$.

Algorithm 2 describes the construction of the NFA from G. First, the set of states Q is composed from the items of G. Then the transitions in R are constructed, assuming only items

Algorithm 2 Computing the NFA from a grammar.

```
function BUILD-NFA() =
1  $Q = I$ // the items of $G$
2  $R = \{A \to \alpha \bullet X\beta \overset{X}{\longmapsto} A \to \alpha X \bullet \beta \mid \}$ // shifts
3     $\cup \{A \to \alpha \bullet B\gamma \overset{\langle_i}{\longmapsto} B \to \bullet\beta \mid i = \mathsf{pid}(B \to \beta)\}$ // derives
4     $\cup \{B \to \beta \bullet \overset{\rangle_i}{\longmapsto} A \to \alpha B \bullet \gamma \mid i = \mathsf{pid}(B \to \beta)\}$ // reduces
5  return $(Q, R)$
```

in Q are used. Lines 2–4 respectively build the shift, derive and reduce transitions between the items of G.

Intuitively, the NFA resembles an LR(0) parse automaton before the item closure. The major differences are that also shifts of non-terminals are allowed, and that the NFA has — by definition — no stack. The LR(0) pushdown automaton uses its stack to determine the next reduce action, but in the NFA all possible reductions are allowed. Its language is therefore an overapproximation of the set of parse trees of G. However, the shape of the NFA does allow us to turn it into a pushdown automaton that only generates valid parse trees of G. We will do this later on in the sentence generation stage.

Without a stack the NFA can be searched for ambiguity in finite time. Two paths through it that shift the same sequence of symbols in V, but different bracket symbols in T_b, represent a possible ambiguity. If no conflicting paths can be found then G is unambiguous, but otherwise it is uncertain whether or not all conflicting paths represent ambiguous strings in $\mathcal{L}(G)$. However, the conflicting paths can be used to find harmless production rules. These are the rules that are not or incompletely used on these paths. If not all items of a production are used in the overapproximated set of ambiguous parse trees of G, then the production can certainly not be used to create a real ambiguous string in $\mathcal{L}(G)$.

5.4.2 Step 2: Construct and Traverse Pair Graph

NFA Traversal

To collect the items used on all conflicting path pairs we can traverse the NFA with two cursors at the same time. The traversal starts with both cursors at the start item $S' \to \bullet S\$$. From there they can progress through the NFA either independently or synchronized, depending on the type of the followed transition. Because we are looking for conflicting paths that represent different parse trees of the same string, the cursors should shift the same symbols. To enforce this we only allow synchronized shifts of equal symbols. The derive and reduce transitions are followed asynchronously, because the number of brackets on each path may vary.

During the traversal we wish to avoid the derivation of unambiguous substrings, i.e. an identical sequence of one derive, zero or more shifts, and one reduce on both paths, and prefer non-terminal shifts instead. This enables us to filter more items and edges from the NFA. Identical reduce transitions on both paths are therefore not allowed if no conflicts have occurred yet since their corresponding derives. A path can thus only reduce if the other path can be continued with a different reduce or a shift. This puts the path in conflict with the other.

After the paths are conflicting we do allow identical reductions (synchronously), because otherwise it would be impossible to reach the end item $S' \to S\$\bullet$. To register whether a path is in conflict with the other we use boolean flags, one for each path. For a more detailed description of these flags we refer to [Sch07a] and Chapter 4.

Algorithm 3 describes the traversal of path pairs through the NFA. It contains gaps ①–④ that we will fill in later on (Algorithm 9), when extending it to handle character-level grammars. To model the state of the cursors during the traversal we use an *item pair* datatype with four fields: two items q_1 and q_2 in Q, and two conflict flags c_1 and c_2 in $\mathbb{B}$. We use Π to denote the set of all possible item pairs.

The function TRAVERSE-EDGES explores all possible continuations from a given item pair. We assume it has access to the global NFA variables Q and R. To traverse each pair graph edge it calls the function TRAVERSE-EDGE — see Algorithm 4 — which in turn calls TRAVERSE-EDGES again on the next pair. The function SHIFTABLE determines the symbol that can be shifted on both paths. In the baseline setting we can only shift if the next symbols of both paths are identical. Later on we will extend this function to handle character-level grammars. The function CONFLICT determines whether a reduce transition of a certain path leads to a conflict. This is the case if the other path can be continued with a shift or reduce that differs from the first path's reduce.

Pair Graph

There can be infinitely many path pairs through the NFA, which can not all be traversed one-by-one. We therefore model all conflicting path pairs with a finite structure, called a *pair graph*, which nodes are item pairs. The function TRAVERSE-EDGES describes the edges of this graph. An infinite amount of path pairs translates to cycles in this finite pair graph. To find the items used on all conflicting paths it suffices to do a depth first traversal of the pair graph that visits each edge pair only once. Since the pair graph might contain dead ends, the traversal also needs to keep track of which item pairs are 'alive'. This is the case if they can reach an end pair: a pair with $S' \to S\$\bullet$ for both its items.

Algorithm 4 describes how to find all items used in alive item pairs in the pair graph. It basically is the depth first traversal algorithm of Tarjan to find the strongly connected components (SCCs) of a graph [Tar72], extended to mark alive item pairs. We will not go into the details of the traversal algorithm, but will only comment on the marking of the alive pairs.

The start function for the traversal is TRAVERSE-PATH-PAIRS. It initializes *alive*, the global set of alive pairs, with the possible end pairs of the pair graph (line 3), and begins the traversal by calling TRAVERSE-PAIR with the start pair (line 4). This function traverses all edge pairs originating at a given item pair using the earlier described function TRAVERSE-EDGES — see Algorithm 3 —, which in turn calls the function TRAVERSE-EDGE for each edge pair. Line 6 of this function adds the source pair of an edge to *alive* if its target pair is alive. Lines 5–8 of TRAVERSE-PAIR add all elements of a strongly connected component to *alive* if the root of the component is alive. Thus, if a traversal of a path reaches an end pair or an already visited alive pair, all pairs on the path get marked alive as the path is unwound.

Algorithm 3 Traversing NFA edge pairs.

```
function TRAVERSE-EDGES($p \in \mathit{\Pi}$) =
for each $(p.q_1 \xmapsto{\langle_i} q_1') \in R$ do //derive of $q_1$
    $p' = p,\ p'.q_1 = q_1',\ p'.c_1 = 0$
    TRAVERSE-EDGE($p, p'$)
od
for each $(p.q_2 \xmapsto{\langle_i} q_2') \in R$ do //derive of $q_2$
    $p' = p,\ p'.q_2 = q_2',\ p'.c_2 = 0$
    TRAVERSE-EDGE($p, p'$)
od
for each $(p.q_1 \xmapsto{X} q_1'), (p.q_2 \xmapsto{Y} q_2') \in R$ do // synchronized shift
    if SHIFTABLE($X, Y$) $\neq \emptyset$ then
        $p' = p,\ p'.q_1 = q_1',\ p'.q_2 = q_2'$
        // ... ①
        TRAVERSE-EDGE($p, p'$)
    fi
for each $(p.q_1 \xmapsto{\rangle_i} q_1') \in R$ do
    if CONFLICT($p.q_2, \rangle_i$) then // conflicting reduction of $q_1$
        $p' = p,\ p'.q_1 = q_1',\ p'.c_1 = 1$
        // ... ②
        TRAVERSE-EDGE($p, p'$)
    fi
for each $(p.q_2 \xmapsto{\rangle_i} q_2') \in R$ do
    if CONFLICT($p.q_1, \rangle_i$) then // conflicting reduction of $q_2$
        $p' = p,\ p'.q_2 = q_2',\ p'.c_2 = 1$
        // ... ③
        TRAVERSE-EDGE($p, p'$)
    fi
if $p.c_1 \vee p.c_2$ then
    for each $(p.q_1 \xmapsto{\rangle_i} q_1'), (p.q_2 \xmapsto{\rangle_i} q_2') \in R$ do // synchronized reduction
        $p' = p,\ p'.q_1 = q_1',\ p'.q_2 = q_2',\ p'.c_1 = p'.c_2 = 1$
        // ... ④
        TRAVERSE-EDGE($p, p'$)
    od
```

```
function SHIFTABLE($X \in V, Y \in V$) =
// returns whether $X$ and $Y$ can be shifted together
if $X = Y$ then return $X$ else return $\emptyset$
```

```
function CONFLICT($q \in Q,\ \rangle_i \in T_\rangle$) =
// returns whether a shift or reduce transition that conflicts with $\rangle_i$ can be taken from $q$
return $\exists q' \in Q, u \in T_\langle^* : (\exists X : q \xmapsto{uX}{}^+ q') \vee (\exists \rangle_j \neq \rangle_i : q \xmapsto{u\rangle_j}{}^+ q')$
```

Algorithm 4 Traversing the pair graph.

```
function TRAVERSE-PATH-PAIRS($Q$, $R$) =
1 unnumber all pairs in $\Pi$, $i = 0$
2 initialize stack to empty
3 alive = $\{(S' \to S\$\bullet, S' \to S\$\bullet, 0, 1), (S' \to S\$\bullet, S' \to S\$\bullet, 1, 1)\}$
4 TRAVERSE-PAIR($(S' \to \bullet S\$, S' \to \bullet S\$, 0, 0)$)
5 return $\{q_1, q_2 \mid \exists(q_1, q_2, c_1, c_2) \in alive\}$

function TRAVERSE-PAIR($p \in \Pi$) =
1 $i$ = $i + 1$, lowlink($p$) = number($p$) = $i$
2 push $p$ on stack
3 TRAVERSE-EDGES($p$)
4 if lowlink($p$) = number($p$) then // p is the root of a SCC
5     do // add all pairs in SCC to alive if p is alive
6         pop $p'$ from stack
7         if $p \in$ alive then add $p'$ to alive
8     while $p' \neq p$

function TRAVERSE-EDGE($p_1 \in \Pi$, $p_2 \in \Pi$) =
1 if $p_2$ is unnumbered then
2     TRAVERSE-PAIR($p_2$)
3     lowlink($p_1$) = min(lowlink($p_1$), lowlink($p_2$))
4 else if $p_2$ is on stack then
5     lowlink($p_1$) = min(lowlink($p_1$), number($p_2$))
6 if $p_2 \in$ alive then add $p_1$ to alive
```

Algorithm 5 Filtering unused states and transitions from an NFA.

```
function FILTER-NFA($Q$, $R$, $Q_a \subseteq Q$) =
1 // gather subNFA of completely used productions
2 $Q_u = \{A \to \alpha\bullet\beta \mid A \to \alpha\beta \in$ USED-PRODUCTIONS$(Q_a)\}$
3 $R_u = \{q_0 \overset{X_b}{\longmapsto} q_1 \in R \mid q_0, q_1 \in Q_u\}$ // $R$ restricted to $Q_u$
4 // prune dead ends
5 $Q' = \{q \in Q_u \mid S' \to \bullet S\$ \ R_u^* \ q,\ q \ R_u^* \ S' \to S\$\bullet\}$ // items alive in $R_u$
6 $R' = \{q_0 \overset{X_b}{\longmapsto} q_1 \in R_u \mid q_0, q_1 \in Q'\}$ // $R_u$ restricted to $Q'$
7 return $(Q', R')$

function USED-PRODUCTIONS($Q' \subseteq Q$) =
1 return $\{A \to \alpha \mid \mathsf{proditems}(A \to \alpha) \subseteq Q'\}$
```

5.4.3 Steps 3–4: NFA Filtering and Harmless Rules Identification

After the items used on conflicting path pairs are collected in Q_a we can identify harmless production rules from them. As said, these are the productions of which not all items are used. All other productions of G are potentially harmful, because it is uncertain if they can really be used to derive ambiguous strings. They can be calculated using the function USED-PRODUCTIONS shown in Algorithm 5.

We filter the harmless production rules from the NFA by removing all their items and pruning dead ends. This is described by the function FILTER-NFA in Algorithm 5. If there are productions of which some but not all items were used, we actually remove a number of conflicting paths that do not represent valid parse trees of G. After filtering there might thus be even more unused items in the NFA. We therefore repeat the traversing and filtering process until no more items can be removed — see again Algorithm 1. Then, all productions that are not used in the NFA are harmless. This step concludes the first stage of our framework (*Find Harmless Productions* in Figure 5.1).

5.4.4 Steps 5–7: NFA Reconstruction and Sentence Generation

In the second part of our framework we use an inverted SGLR parser [vdBSVV02] as a sentence generator to find real ambiguous sentences in the remainder of the NFA. However, certain states in the NFA might not lead to the generation of terminal-only sentences anymore, due to the removal of terminal shift transitions during filtering. These are the states with outgoing non-terminal shift transitions that have no corresponding derive and reduce transitions anymore. To make such a non-terminal productive again we introduce a new terminal-only production for it that produces a shortest string from its original language. Then we add a new chain of derive, shift, and reduce transitions for this production to the states before and after the unproductive non-terminal shift.

After the NFA is reconstructed we generate an LR(0) pushdown automaton from it to generate sentences with. In contrast to the first stage, we now do need a stack because we only want to generate proper derivations of the grammar. Also, because of the item closure that is applied in LR automata, all derivations are unfolded statically, which saves generation steps at run-time.

The inverted parser generates all sentences of the grammar, together with their parse trees. If it finds a sentence with multiple trees then these are reported to the user. They are the most precise ambiguity reports possible, and are also very descriptive because they show the productions involved. Because the number of derivations of a grammar can be infinite, we continue searching strings of increasing length until a certain time limit is reached. The number of strings to generate can grow exponentially with increasing length, but filtering unambiguous derivations beforehand can also greatly reduce the time needed to reach a certain length as Section 5.7 will show.

5.5 Ambiguity Detection for Character-level Grammars

After sketching the baseline algorithm we can extend it to find ambiguities in character-level grammars. We take disambiguation filters into account during ambiguity detection, so we do not report ambiguities that are already solved by the grammar developer. Furthermore, we explain and fix the issue that the baseline harmless rules filtering is unable to properly deal with follow restrictions.

5.5.1 Application of Baseline Algorithm on Example Grammar

Before explaining our extensions we first show that the baseline algorithm can lead to incorrect results on character-level grammars. If we apply it to the example grammar of Figure 5.2, the harmless production rule filter will actually remove ambiguities from the grammar. Since the filtering is supposed to be conservative, this behaviour is incorrect.

The baseline algorithm will ignore the reject rule and follow restrictions in the grammar (Rules 6, 7, 9–11), and will therefore find the ambiguities that these filters meant to solve. Ambiguous strings are, among others, "`float f;`" (`float` can be a keyword or identifier) and "`intList l;`" (`intList` can be one or more specifiers). Rules 1–5 will therefore be recognized as potentially harmful. However, in all ambiguous strings, the substrings containing whitespace will always be unambiguous. This is detected by the PG traversal and Rule 8 (`Ws ::= [\ \t\n]+`) will therefore become harmless.

Rule 8 will be filtered from the grammar, and during reconstruction `Ws?` will be terminalized with the shortest string from its language, in this case ε. This effectively removes all whitespace from the language of the grammar. In the baseline setting the grammar would still be ambiguous after this, but in the character-level setting the language of the grammar would now be empty! The follow restriction of line 9 namely dictates that valid `Declaration` strings should contain at least one whitespace character to separate specifiers and identifiers.

This shows that our baseline grammar filtering algorithm is not suitable for character-level grammars as is, because it might remove ambiguous sentences. In addition, it might even introduce ambiguities in certain situations. This can happen when non-terminals are removed that have follow restrictions that prohibit a second derivation of a certain string. In short, follow restrictions have a non-context-free influence on sentence derivation, and the baseline algorithm assumes only context-free derivation steps. In the extensions presented in the next

Algorithm 6 SHIFTABLE function for character-level pair graph.

```
function SHIFTABLE($X \in N, Y \in N$) =
  // returns the symbol that can be shifted from $X$ and $Y$
  if $X \in \mathcal{C} \wedge Y \in \mathcal{C}$ then // $X$ and $Y$ are character classes
    return $X \cap Y$
  else if $X = Y$ then // $X$ and $Y$ are the same non-terminal
    return $X$
  else // no shift possible
    return $\emptyset$
```

Algorithm 7 Filtering derive restrictions from the NFA.

```
function FILTER-DERIVE-RESTRICTIONS($R$) =
  return $R \backslash \{A \to \alpha \bullet B\gamma \overset{\langle_i}{\longmapsto} B \to \bullet\beta \mid i = \mathsf{pid}(B \to \beta),\ (A \to \alpha \bullet B\gamma, B \to \beta) \in R_D\}$
         $\backslash \{B \to \beta \bullet \overset{\rangle_i}{\longmapsto} A \to \alpha B \bullet \gamma \mid i = \mathsf{pid}(B \to \beta),\ (A \to \alpha \bullet B\gamma, B \to \beta) \in R_D\}$
```

section we repair this flaw and make sure that the resulting algorithm does not introduce or lose ambiguous sentences.

5.5.2 Changes to the Baseline Algorithm

The differences between character-level grammars and conventional grammars result in several modifications of our baseline algorithm. These modifications deal with the definitions of both the NFA and the pair graph. We reuse the NFA construction of Algorithm 2 because it is compliant with character-level productions, and apply several modifications to the NFA afterwards to make it respect a grammar's derivation restrictions and follow restrictions. An advantage of this is that we do not have to modify the pair graph construction. To keep the test practical and conservative we have to make sure that the NFA remains finite, while its paths describe an overapproximation of $\mathcal{S}(G_b)$.

Character Classes

Because of the new shape of the productions, we now shift entire character classes at once, instead of individual terminal symbols. This avoids adding derives, shifts and reduces for the terminals in all character classes, which would bloat the NFA, and thus also the pair graph. In the PG we allow a synchronized shift of two character classes if their intersection is non-empty. To enforce this behaviour we only need to change the SHIFTABLE function as shown in Algorithm 6.

Derivation Restrictions and Follow Restrictions

After the initial NFA is constructed we remove derive and reduce edges that are disallowed by the derivation restrictions. This is described in function FILTER-DERIVE-RESTRICTIONS in Algorithm 7. Then we propagate the follow restrictions through the NFA to make it only generate strings that comply with them. This is described in function PROPAGATE-FOLLOW-RESTRICTIONS in Algorithm 8. The operation will result in a new NFA with states that are tuples containing a state of the original NFA and a set of follow restrictions over $\mathcal{P}(T^+)$. A new state cannot be followed by strings that have a prefix in the state's follow restrictions. To enforce this we constrain character class shift edges according to the follow restrictions of their source states.

The process starts at $(S' \rightarrow \bullet S\$, \emptyset)$ and creates new states while propagating a state's follow restrictions over the edges of its old NFA item. In contrast to the original NFA, which had at most one shift edge per state, states in the new NFA can have multiple. This is because non-terminal or character class edges actually represent the shift of multiple sentences, which can each result in different follow restrictions. Lines 6–9 show the reconstruction of character-class shift edges from a state $(A \rightarrow \alpha \bullet B\beta, f)$. Shift edges are added for characters in B that are allowed by f. All characters in B that will result in the same new set of follow restrictions are combined into a single shift edge, to not bloat the new NFA unneccesarily. The restrictions after a shift of a are the tails of the strings in f beginning with a, and are calculated by the function NEXT-FOLLOW.

Line 12 describes how a state's restrictions are passed on unchanged over derive edges. Lines 13–20 show how new non-terminal shift edges are added from a state $(A \rightarrow \alpha \bullet B\beta, f)$ once their corresponding reduce edges are known. This is convenient because we can let the propagation calculate the different follow restrictions that can reach $A \rightarrow \alpha B \bullet \beta$. Once the restrictions that were passed to the derive have reached a state $B \rightarrow \gamma \bullet$, we propagate them upwards again over a reduce edge to $A \rightarrow \alpha B \bullet \beta$. If B has follow restrictions — in R_F — these are added to the new state as well. Note that multiple follow restriction sets might occur at the end of a production, so we might have to reduce a production multiple times. For a given state $B \rightarrow \bullet \gamma$, the function SHIFT-ENDS returns all states that are at $B \rightarrow \gamma \bullet$ and that are reachable by shifting.

If the reduced production is of form $B \rightarrow \varepsilon$ we create a special non-terminal symbol B^ε and make it the label of the shift edge instead of B. This is a small precision improvement of the PG traversal. It prevents the situation where a specific non-terminal shift that — because of its follow restriction context — only represents the empty string, is traversed together with another instance of the same non-terminal that cannot derive ε.

The propagation ends when no new edges can be added to the new NFA. In theory the new NFA can now be exponentially larger than the original, but since follow restrictions are usually only used sparingly in the definition of lexical syntax this will hardly happen in practice. In Section 5.7 we will see average increases in NFA size of a factor 2–3.

Rejects

Instead of encoding a grammar's rejects in the NFA, we choose to handle them during the PG traversal. Consider an element (A, B) in R_R, which effectively subtracts the language of B from that of A. If the language of B is regular then we could, for instance, subtract it from the NFA part that overapproximates the language of A. This would not violate the finiteness and overapproximation requirements. However, if the language of B is context-free we have to

Algorithm 8 Propagating follow restrictions through the NFA.

```
function PROPAGATE-FOLLOW-RESTRICTIONS($Q$, $R$) =
1  // propagate follow restrictions through NFA $(Q, R)$ and return a new NFA $(Q', R')$
2  $Q' = \{(S' \to \bullet S\$, \emptyset)\}$, $R' = \emptyset$
3  repeat
4    add all states used in $R'$ to $Q'$
5    for $q_f = (A \to \alpha \bullet B\beta, f) \in Q'$ do
6      if $B \in \mathcal{C}$ then // $B$ is a character class
7        for $a \in B$, $a \notin f$ do // all shiftable characters in $B$
8          let $B' = \{b | b \in B, b \notin f$, NEXT-FOLLOW$(a, f)$ = NEXT-FOLLOW$(b, f)\}$
9          add $q_f \overset{B'}{\longmapsto} (A \to \alpha B \bullet \beta$, NEXT-FOLLOW$(a, f))$ to $R'$
10       od
11     else // $B$ is a normal non-terminal
12       for $A \to \alpha \bullet B\beta \overset{\langle_i}{\longmapsto} q' \in R$ do
13         add $q_f \overset{\langle_i}{\longmapsto} (q', f)$ to $R'$ // propagate $f$ over derivation
14         for $q_f^r = (q^r, f^r) \in$ SHIFT-ENDS$((q', f))$
15           let $q_f^s = (A \to \alpha B \bullet \beta, f^r \cup R_F(B))$ // shift target
16           add $q_f^r \overset{\rangle_i}{\longmapsto} q_f^s$ to $R'$ // reduction to shift target
17           if bracketP$(\langle_i) = B \to \varepsilon$ then
18             add $q_f \overset{B^\varepsilon}{\longmapsto} q_f^s$ to $R'$ // non-terminal shift representing empty string
19           else
20             add $q_f \overset{B}{\longmapsto} q_f^s$ to $R'$ // non-terminal shift of non-empty strings
21         od
22       od
23 until no more edges can be added to $R'$
24 return $(Q', R')$

function SHIFT-ENDS($(A \to \bullet\alpha, f) \in Q'$) =
1  // return the states at the end of $A \to \alpha$, reachable from $q$ using only shifts
2  let $\dashrightarrow = \{q \dashrightarrow q' \mid q \overset{B}{\longmapsto} q' \in R'\}$ // the shift transitions of $R'$
3  return $\{(A \to \alpha\bullet, f') \mid (A \to \bullet\alpha, f) \dashrightarrow^* (A \to \alpha\bullet, f')\}$

function NEXT-FOLLOW($a \in T$, $f \in \mathcal{P}(T^+)$)
1  return $\{\alpha \mid a\alpha \in f, \alpha \neq \varepsilon\}$ // the next follow restrictions of $f$ after a shift of $a$
```

*under*approximate it to finite form first, to keep the NFA an overapproximation and finite. A possible representation for this would be a second NFA, which we could subtract from the first NFA beforehand, or traverse alongside the first NFA in the PG.

Instead, we present a simpler approach that works well for the main use of rejects: keyword reservation. We make use of the fact that keywords are usually specified as a set of non-terminals that represent literal strings — like Rules 6 and 7 in Figure 5.2. The production

Algorithm 9 Extensions to TRAVERSE-EDGES for avoiding rejected keywords.

```
// at ① (shift) insert:
    $p'.r_1 = p'.r_2 = \emptyset$ // clear reduced sets

// at ② and ④ (conflicting and pairwise reduce) insert:
    if not CHK-REJECT($\rangle_i, p.r_2$) then continue
    $p'.r_1$ = NEXT-REJECT($\rangle_i, p.r_1$)

// similarly, insert at ③ and ④:
    if not CHK-REJECT($\rangle_i, p.r_1$) then continue
    $p'.r_2$ = NEXT-REJECT($\rangle_i, p.r_2$)

function CHK-REJECT($\rangle_i \in T_\rangle, r \in \mathcal{P}(N)$) =
    // returns whether a reduction with $\rangle_i$ is possible after reductions $r$ on other path
    let $A = \mathsf{bracketN}(\rangle_i)$
    return $\neg\exists\, B \in r : (A, B) \in R_R \vee (B, A) \in R_R$

function NEXT-REJECT($\rangle_i \in T_\rangle, r \in \mathcal{P}(N)$) =
    // adds non-terminal reduced with $\rangle_i$ to $r$ if it is involved in a reject
    let $A = \mathsf{bracketN}(\rangle_i)$
    if $\exists B \in r : (A, B) \in R_R \vee (B, A) \in R_R$ then
        return $r \cup \{A\}$
    else return $r$
```

rules for `"int"`, `"float"`, etc. are not affected by the approximation, and appear in the NFA in their original form. We can thus recognize that, during the PG traversal, a path has completely shifted a reserved keyword if it reduces `"int"`. After that, we can prevent the other path from reducing `Identifier` before the next shift. This does not restrict the language of `Identifier` in the NFA — it is kept overapproximated —, but it does prevent the ambiguous situation where “`int`” is recognized as an `Identifier` on one path and as an `"int"` on the other path.

Of course, `Identifier` could also be reduced before `"int"`, so we need to register the reductions of both non-terminals. During the PG traversal, we keep track of all reduced non-terminals that appear in R_R, in two sets r_1 and r_2, one for each path. Then, if a path reduces a non-terminal that appears in a pair in R_R, together with a previously reduced non-terminal in the other path's set, we prevent this reduction. The sets are cleared again after each pairwise shift transition. Algorithm 9 shows this PG extension.

5.5.3 NFA Reconstruction

In Section 5.5.1 we saw that follow restrictions should be handled with care when filtering and reconstructing a grammar, because of their non-context-free behaviour. By removing productions from a grammar certain follow restrictions can become unavoidable, which removes sentences from the language. On the other hand, by removing follow restrictions new sentences can be introduced that were previously restricted. When reconstructing a character-level grammar we thus need to terminalize filtered productions depending on the possible follow-restrictions they might generate or that might apply to them.

Instead, by directly reusing the filtered NFA for sentence generation, we can avoid this problem. The follow restrictions that are propagated over the states already describe the follow restriction context of each item. For each distinct restriction context of an item a separate state exists. We can just terminalize each unproductive non-terminal shift edge with an arbitrary string from the language of its previously underlying automaton.

Furthermore, the filtered NFA is a more detailed description of the potentially ambiguous derivations than a filtered grammar, and therefore describes less sentences. For instance, if derive and reduce edges of a production $B \rightarrow \beta$ are filtered out at a specific item $A \rightarrow \alpha \bullet B\gamma$, but not at other items, we know $B \rightarrow \beta$ is harmless in the context of $A \rightarrow \alpha \bullet B\gamma$. The propagated follow restrictions also provide contexts in which certain productions can be harmless. We could encode this information in a reconstructed grammar by duplicating non-terminals and productions of course, but this could really bloat the grammar. Instead, we just reuse the baseline NFA reconstruction algorithm.

5.6 Grammar Unfolding

In Section 5.7 we will see that the precision of the algorithm described above is not always sufficient for some real life grammars. The reason for this is that the overapproximation in the NFA is too aggressive for character-level grammars. By applying grammar unfoldings we can limit the approximation, which improves the precision of our algorithm.

The problem with the overapproximation is that it becomes too aggressive when certain non-terminals are used very frequently. Remember that due to the absence of a stack, the derive and reduce transitions do not have to be followed in a balanced way. Therefore, after deriving from an item $A \rightarrow \alpha \bullet B\beta$ and shifting a string in the language of B, the NFA allows reductions to any item of form $C \rightarrow \gamma B \bullet \delta$. This way, a path can jump to another production while being in the middle of a first production. Of course, a little overapproximation is intended, but the precision can be affected seriously if certain non-terminals are used very frequently. Typical non-terminals like that in character-level grammars are those for whitespace and comments, which can appear in between almost all language constructs. Since these non-terminals can usually derive to ε, we can thus jump from almost any item to almost any other item by deriving and reducing them.

To restrict the overapproximation we can unfold the frequently used non-terminals in the grammar, with a technique similar to one used in [BGM10]. A non-terminal is unfolded by creating a unique copy of it for every place that it occurs in the right-hand sides of the production rules. For each of these copies we then also duplicate the entire sub-grammar of the non-terminal. The NFA thus gets a separate isolated sub-automaton for each occurence of an unfolded non-terminal. After the derivation from an item $A \rightarrow \alpha \bullet B\beta$ a path can now only reduce back to $A \rightarrow \alpha B \bullet \beta$, considering B is unfolded. After unfolding, the NFA contains more states, but has less paths through it because it is more deterministic. In the current implementation we unfold all non-terminals that describe whitespace, comments, or

Name	Produc- tions	SLOC	Non- terminals	Derive restrictions	Follow restrictions	Reserved keywords
C[1]	324	415	168	332	10	32
C++[2]	807	4172[a]	430	1	87	74
ECMAScript[3]	403	522	232	1	27	25
Oberon0[4]	189	202	120	132	31	27
SQL-92[5]	419	495	266	23	5	30
Java 1.5[6]	698	1629	387	297	78	56

[1] SDF2 grammar library, revision 27501, `http://www.meta-environment.org`

[2] TRANSFORMERS 0.4, `http://www.lrde.epita.fr/cgi-bin/twiki/view/Transformers/Transformers`

[3] ECMASCRIPT-FRONT, revision 200, `http://strategoxt.org/Stratego/EcmaScriptFront`

[4] RASCAL Oberon0 project (converted to SDF2), rev. 34580, `http://svn.rascal-mpl.org/oberon0/`

[5] SQL-FRONT, revision 20713, `http://strategoxt.org/Stratego/SqlFront`

[6] JAVA-FRONT, revision 17503, `http://strategoxt.org/Stratego/JavaFront`

[a] After removal of additional attribute code

Table 5.1: Character-level grammars used for validation.

literal strings like keywords, brackets and operators. Later on we will refer to this unfolding extension as CHAR+UNF.

5.7 Experimental Results

We have evaluated our ambiguity detection algorithm for character-level grammars on the grammar collection shown in Table 5.1. All grammars are specified in SDF2 [HHKR89, Vis97]. The selection of this set is important for external validity. We have opted for grammars of general purpose programming languages, which makes it easier for others to validate our results. For each grammar we give its name, number of productions, number of source lines (SLOC), number of non-terminals, number of priorities and associativities (derivation restrictions), number of follow restrictions and number of reserved keywords.

5.7.1 Experiment Setup

We have run both our NFA filtering and sentence generation algorithms on each of these grammars. Most measurements were carried out on an Intel Core2 Quad Q6600 2.40GHz with 8GB DDR2 memory, running Fedora 14. A few memory intensive runs were done on an Amazon computing cloud EC2 High-Memory Extra Large Instance with 17.1GB memory. The algorithms have been implemented in Java and are available for download at `http://homepages.cwi.nl/~basten/ambiguity`. In order to identify the effects of the various extensions, we present our empirical findings for the following combinations:

- **BASE**: the baseline algorithm for token-level grammars as described in Section 5.4, with the only addition that whole character-classes are shifted instead of individual

Grammar	**Method**	**Harmless productions**	**NFA edges filtered**	**Time** (sec)	**Memory** (MB)
C	BASE	48 / 324	343 / 14359	64	2128
	CHAR	62 / 324	2283 / 24565	120	3345
	CHAR+UNF	75 / 324	8637 / 30653	97	2616
C++	BASE	0 / 807	0 / 8644	32	1408
	CHAR	0 / 807	0 / 39339	527	7189
	CHAR+UNF[a]	–	–	>9594	>17.3G
ECMAScript	BASE	44 / 403	414 / 4872	12	547
	CHAR	46 / 403	1183 / 10240	46	1388
	CHAR+UNF	88 / 403	9887 / 19890	31	1127
Oberon0	BASE	0 / 189	0 / 3701	4.2	256
	CHAR	70 / 189	925 / 6162	9.0	349
	CHAR+UNF	73 / 189	10837 / 20531	14	631
SQL-92	BASE	13 / 419	98 / 4944	16	709
	CHAR	20 / 419	239 / 9031	83	2093
	CHAR+UNF	65 / 419	7285 / 14862	37	1371
Java 1.5	BASE	0 / 698	0 / 16844	60	2942
	CHAR	0 / 698	0 / 45578	407	7382
	CHAR+UNF[a]	189 / 698	180456 / 262030	1681	15568

[a]Run on Amazon EC2 High-Memory Extra Large Instance

Table 5.2: Timing and precision results of filtering harmless productions.

tokens. Even though this configuration can lead to incorrect results, it is included as a baseline for comparison.

- **CHAR**: the baseline algorithm extended for handling character-level grammars as described in Section 5.5, including extensions for follow restrictions, derive restrictions and rejects.

- **CHAR+UNF**: the **CHAR** algorithm combined with grammar unfolding (Section 5.6).

5.7.2 Results and Analysis

In Table 5.2 we summarize our measurements of the NFA filtering and harmless production rule detection. For each grammar and extension configuration we give the number of harmless productions found versus total number of productions, number of edges filtered from the NFA, execution time (in seconds) and memory usage (in MB).

Every configuration was able to filter an increasing number of productions and edges for each of the grammars. For C and ECMAScript **BASE** could already filter a small number rules and edges, although it remains unsure whether these are all harmless because the baseline algorithm cannot handle follow restrictions properly. For C and Oberon0 our character-level extensions of **CHAR** improved substantially upon **BASE**, without the risk of missing ambiguous sentences.

Of all three configurations **CHAR+UNF** was the most precise. For the grammar order of the table, it filtered respectively 23%, 0%, 22%, 39%, 16% and 27% of the production rules, and 28%, 0%, 50%, 53%, 49% and 69% of the NFA edges. Unfolding grammars leads to larger but more deterministic NFAs, which in turn can lead to smaller pair graphs and thus faster traversal times. This was the case for most grammars except the larger ones. ECMAScript, SQL-92 and Oberon0 were checkable in under 1 minute, and C in under 2 minutes, all requiring less than 3GB of memory. Java 1.5 was checkable in just under 16GB in 30 minutes, but for the C++ grammar — which is highly ambiguous — the pair graph became too large. However, the additional cost of unfolding was apparently necessary to deal with the complexity of Java 1.5.

Table 5.3 allows us to compare the sentence generation times for the unfiltered and filtered NFAs. For each grammar and NFA it shows the number of sentences of a certain length in the language of the NFA, and the times required to search them for ambiguities. The unfiltered sentence generation also takes disambiguation filters into account. C++ is not included because its NFA could not be filtered in the previous experiments.

For all grammars we see that filtering with **CHAR** and **CHAR+UNF** lead to reductions in search space and generation times. To indicate whether the investments in filtering time actually pay off, the last column contains the maximum speedup gained by either **CHAR** or **CHAR+UNF**. For sentence lengths that are already relatively cheap to generate, filtering beforehand has no added value. However, the longer the sentences get the greater the pay-off. We witnessed speedup factors ranging from a small 1.1 (C length 7) to a highly significant 3399 (Oberon0 length 26). Filtering Oberon0 with **CHAR+UNF** was so effective that it increased the sentence length checkable in around 15 minutes from 24 to 35.

For most grammars filtering already becomes beneficial after around 15 seconds to 6 minutes. For Java 1.5 this boundary lies around 35 minutes, because of its high filtering time. However, after that we see an immediate speedup of a factor 6.2. In all cases **CHAR+UNF** was superior to **CHAR**, due to its higher precision and lower run-times.

The third column of Table 5.3 contains the number of ambiguous non-terminals found at each length. Because of filtering, ambiguous non-terminals at larger lengths were found earlier in multiple grammars. There were 2 ambiguous non-terminals in C that were found faster, and 4 non-terminals in ECMAScript and 3 in SQL-92.

Concluding, we see that our character-level NFA filtering approach was very beneficial on the small to medium grammars. A relatively low investment in filtering time — under 2 minutes — lead to significant speedups in sentence generation. This enabled the earlier detection of ambiguities in these grammars. For the larger Java 1.5 grammar the filtering became beneficial only after 32 minutes, and for the highly ambiguous C++ grammar the filtering had no effect at all. Nevertheless, ambiguity detection for character-level grammars is ready to be used in interactive language workbenches.

		Ambig	**Unfiltered**		**CHAR**		**CHAR+UNF**		**Maximum**
Grammar	**Len**	**NTs**	#Sent.	Time	#Sent.	Time	#Sent.	Time	**speedup**
C	5	6	345K	**7.9**	273K	5.9	267K	5.9	0.08x
	6	8	5.06M	**35**	3.77M	25	3.66M	25	0.29x
	7	8	75.5M	398	53.4M	270	51.6M	**259**	1.1x
	8	9	1.13G	5442	756M	3466	727M	**3362**	1.6x
	9	10	17.0G	78987	10.8G	47833	10.3G	**47018**	1.7x
ECMAScript	3	6	14.2K	**4.5**	11.7K	3.5	9.29K	3.3	0.13x
	4	8	274K	**11**	217K	8.9	159K	6.7	0.29x
	5	10	5.17M	149	3.92M	120	2.64M	**69**	1.5x
	6	11	96.8M	2805	70.5M	2186	43.8M	**1184**	2.3x
	7	12	1.80G	54175	1.26G	41091	719M	**20264**	2.7x
Oberon0	22	0	21.7M	60	320	**1.0**	182	1.0	6.0x
	23	0	62.7M	186	571	**1.0**	248	1.0	19x
	24	0	247M	815	1269	**1.0**	468	1.0	82x
	25	0	1.39G	4951	3173	**1.1**	1343	1.1	490x
	26	0	9.56G	35007	9807	**1.3**	3985	1.3	3399x
	32	0			108M	172	13.8M	**28**	
	33	0			549M	885	55.6M	**101**	
	34	0			2.80G	4524	224M	**393**	
	35	0			14.3G	22530	906M	**1591**	
	36	0					3.66G	**6270**	
SQL-92	11	5	2.65M	**16**	1.54M	9.4	321K	4.2	0.39x
	12	6	15.8M	102	7.36M	47	1.66M	**14**	2.0x
	13	6	139M	1018	51.3M	379	11.5M	**90**	8.0x
	14	6	1.49G	11369	453M	3572	90.8M	**711**	15x
	15	7			4.39G	35024	742M	**5781**	
	16	8					6.13G	**47211**	
Java 1.5	7	0	187K	**33**			39.1K	6.8	0.02x
	8	1	3.15M	**115**			482K	20	0.07x
	9	1	54.7M	**1727**			6.05M	212	0.91x
	10	1	959M	39965			76.2M	**4745**	6.2x

Table 5.3: Timing results of sentence generation. Times are in seconds. For each sentence length, the run-time of the fastest configuration (after taking filtering time into account) is highlighted. Speedup is calculated as $\frac{\text{unfiltered sentence gen. time}}{\text{filtering time+sentence gen. time}}$.

5.7.3 Validation

In Chapter 4 we proved the correctness of our baseline algorithm. To further validate our character-level extensions and their implementations we applied them on a series of toy grammars and grammars of real world programming languages. We ran various combinations of our algorithms on the grammars and automatically compared the ambiguous sentences produced, to make sure that only those ambiguities that exist in a grammar were found, so not more and not less. For the version of our implementation that we used for the experiments above, we found no differences in the ambiguous strings generated. The validation was done in the following stages:

- First we built confidence in our baseline sentence generator by comparing it to the external sentence generators AMBER [Sch01] and CFG ANALYZER [AHL08]. For this we used a grammar collection also used in Chapter 2, which contains 87 small toy grammars and 25 large grammars of real-world programming languages.
- Then we validated the character-level extension of the baseline sentence generator by comparing it to a combination of our baseline sentence generator and the SGLR [vdBSVV02] parser used for SDF2. By running the baseline sentence generator on character-level grammars it will report more strings as ambiguous than actually exist in a grammar, because it does not regard disambiguation filters. We therefore filter out the truly ambiguous sentences by using the SGLR parser as an oracle, and test whether our character-level sentence generator finds exactly the same ambiguous sentences. In some situations SGLR will produce non-optimal parse trees, so we had to verify these by hand. In this step and the following we used the SDF2 grammars in Table 5.1.
- Third, we validated our NFA filtering algorithms by running the character-level sentence generator on both filtered and unfiltered NFAs. Because a filtered NFA contains only one reconstructed sentence for non-terminals with only harmless productions, it might produce less variations of ambiguous sentences. We therefore reduced all ambiguous sentences to their *core* ambiguous sentential forms — see Chapter 4 before comparison. This is done by removing the unambiguous substrings from an ambiguous sentence, and replacing them with their deriving non-terminal.

5.8 Conclusion

We have presented new algorithms for ambiguity detection for character-level grammars and by experimental validation we have found an affirmative answer to the question whether ambiguity detection can be scaled to this kind of grammars. We have achieved significant speedups of up to three orders of magnitude for ambiguity checking of real programming language grammars. Ambiguity detection for character-level grammars is ready to be used in interactive language workbenches, which is good news for the main application areas of these grammars: software renovation, language embedding and domain-specific languages.

Chapter 6

Implementing AMBIDEXTER

"I'd give my right arm to be ambidextrous."
Brian W. Kernighan

This chapter presents our tool implementation of AMBIDEXTER. *The tool was developed to experimentally validate the techniques presented throughout this thesis, but also to be used in real grammar development. The tool consists of two parts: a harmless production rule filter, and a parallel sentence generator. We discuss the architecture as well as implementation details of both of these parts, and finish with advise for their usage.*

6.1 Introduction

To be able to validate our grammar filtering techniques described in Chapters 3–5 on real programming language grammars, we needed to implement them in a tool. Furthermore, to test the effect of filtering on exhaustive searching techniques we also needed implementations of these methods. For validating the filtering of token-based grammars we could use existing tools like AMBER and CFG ANALYZER, but for character-level grammars there was no tool available. We therefore created our own sentence generator for this type of grammars, and combined it with our grammar filter into a single tool called AMBIDEXTER. In this chapter we discuss the implementations of both parts of the tool.

This chapter is based on a tool demonstation paper with the same title that was published in the proceedings of the Tenth IEEE Working Conference on Source Code Analysis and Manipulation (SCAM 2010) [BvdS10]. This paper was written together with Tijs van der Storm.

6.2 Grammar Filter

The first part of the tool implements the grammar filtering technique described in Chapters 3, 4 and 5. In this section we assume a basic understanding of the grammar filtering process. For a detailed introduction we refer to Sections 5.2 and 5.4.

6.2.1 Requirements

The main functional requirement of the tool is of course that it should correctly implement our grammar filtering technique and its character-level extensions. But apart from that, it should also fulfill the following non-functional requirements:

- **Memory efficient** Because the number of item pairs can be very high, the pair graph traversal is much more memory intensive than it is cpu-intensive.
- **Extensible** To allow for easy experimenting with new techniques or improvements.
- **Explanatory** The tool should provide insight into the applied techniques, and might serve as a reference implementation.

In the next section we describe the architecture of the grammar filtering tool, and describe how it meets these requirements.

6.2.2 Architecture and Design

Figure 6.1 shows an overview of the architecture of the grammar filtering tool. The process starts from an input grammar and a precision setting for the approximation. The input grammar can be read in YACC [Joh]/BISON [DS05], SDF2 [HHKR89, Vis97] or RASCAL [KvdSV11] format. A nondeterministic finite automaton (NFA) is build from the grammar that approximates its parse trees with the specified precision. The states of this NFA are the items of the grammar and are, depending on the chosen precision, extended with lookahead information. Our tool supports the following precisions: LR(0), SLR(1), LALR(1) and LR(1). The precisions are named after parsing algorithms, because their resulting NFAs resemble their corresponding nondeterministic parse automata.

After the basic NFA is constructed, several precision improving operations are made to it, depending on the type of grammar or options selected by the user. Special *NFA modifiers* can be loaded that each alter the nodes and edges of the NFA, for instance by removing edges or unfolding certain paths. The priority and associativity annotations of character-level grammars as described by Algorithm 7 in Section 5.5.2 are implemented like this, as well as the follow restriction propagation of Algorithm 8. To satisfy the extensibility requirement, new modifiers can be added modularly.

After the NFA is extended, we filter out the states that are not used in the description of parse trees of potentially ambiguous strings. To find these states we build the pair graph. The nodes of the pair graph are pairs of NFA nodes, extended with some additional information. A path through this graph marks two paths through the NFA that describe different parse trees for the same potentially ambiguous string. Therefore, the nodes that need to be filtered from

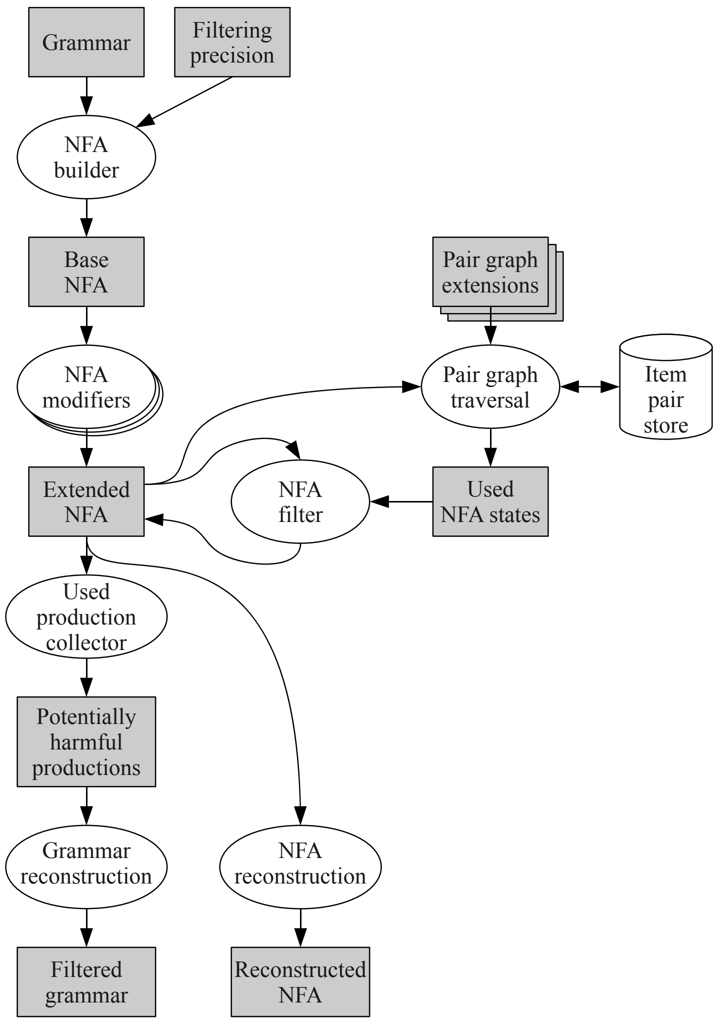


Figure 6.1: Architecture of AMBIDEXTER's grammar filter.

the NFA are those that are not used on any complete path through the pair graph. The pair graph is traversed while it is constructed and the used NFA states are collected — as described in Algorithm 4. After that, the unused states are filtered from the NFA, and loose ends are pruned.

The process of pair graph traversal and NFA filtering is repeated until no more states can be filtered. After that the NFA contains only those states that contribute to possible ambiguity under the current approximation precision. At this point, this information can be used in two ways. The first one is to collect the potentially harmful production rules from the NFA and reconstruct a new grammar from them. This grammar can then be searched further with other ambiguity detection methods. Output formats for the following tools are supported: YACC/BISON, ACCENT/AMBER [Sch06, Sch01], CFG ANALYZER [AHL08] and the implementation of the “Ambiguity Checking with Language Approximation” framework [BGM10]. The other option is to reconstruct unproductive states in the NFA and store it to disk, so that it can be used with AMBIDEXTER’s sentence generator immediately.

To support modifications to the pair graph while keeping a single traversal algorithm, special *pair graph extensions* can be loaded. An extension consists of a set of functions that are called during the traversal of each item pair. The functions are able to store additional information in an item pair, and can use this information to abort the traversal at certain points. As an example, the handling of reject filters in character-level grammars — as described in Section 5.5.2 — is implemented as a pair graph extension. The reject extension stores two additional sets of reduced non-terminals with each item pair, and aborts the traversal of an item pair if it encounters the reduction of a rejected non-terminal.

Another important component in the architecture of the pair graph traversal is the *item pair store*. The purpose of the item pair store is to keep track of all created item pairs and their traversal related information in a memory efficient way. Because the number of item pairs is quadratic in the number of NFA states, pair graphs can become very large. As Table 6.1 shows, the pair graphs required for checking real programming language grammars can grow up to hundreds of millions of pairs. Without efficient storage, they would be impossible to traverse on modern machines. The following section discusses the current implementation of the item pair store in more detail.

6.2.3 Implementation Details

The biggest problem with creating a practically usable grammar filtering tool was to find a memory efficient pair graph representation. For checking character-level grammars, we need to record at least the following information per item pair:

- Two NFA states
- Two conflict flags
- *number* and *lowlink* (see Algorithm 4)
- *alive* and *on stack* flags (see Algorithm 4)

Furthermore, we need to store all created item pairs in a data structure to be able to test whether a newly visited item pair is already visited before or is new. In the first prototype implementation we used plain Java objects to represent an item pair, together with a simple hashmap to store the created objects. For the experiments on token-based grammars shown in Chapter 3 this design performed sufficiently well. However, for running the character-level grammars experiments of Chapter 5, it turned out to be too memory inefficient.

In order to take the reject filters into account during the pair graph traversal, we needed to add two additional sets of reduced non-terminals to our item pairs (see Section 5.5.2). Together

Table 6.1: Automaton sizes during grammar filtering experiments shown in Table 5.2.

Grammar	Rules	Method	NFA states	Item pairs	Memory (MB)
C	324	BASE	1300	937,407	2,128
		CHAR	2485	2,453,230	3,345
		CHAR+UNF	6225	5,730,374	2,616
C++	807	BASE	3014	5,530,348	1,408
		CHAR	9728	65,761,491	7,189
		CHAR+UNF[a]	51723	>399,300,000	>17,293
ECMAScript	403	BASE	1458	1,124,354	547
		CHAR	2695	3,096,719	1,388
		CHAR+UNF	8648	7,956,678	1,127
Oberon0	189	BASE	731	307,192	256
		CHAR	1232	494,143	349
		CHAR+UNF	9038	1,904,378	631
SQL-92	419	BASE	1648	1,473,630	709
		CHAR	2549	4,579,967	2,093
		CHAR+UNF	6287	7,429,850	1,371
Java 1.5	698	BASE	2639	4,579,132	2,942
		CHAR	5892	20,309,357	7,382
		CHAR+UNF[a]	114299	293,592,241	15,568

[a]Run on Amazon EC2 High-Memory Extra Large Instance

with the information mentioned above, our item pair class now consisted of 7 fields. On a 64-bit Java virtual machine, the object size for such a class is 80 bytes of memory. Furthermore, each object required an additional 32 bytes for a bucket entry object in the hashmap. Running the **CHAR+UNF** experiment on the Java grammar shown in Table 6.1 would therefore take up at least 33GB of memory. However, this number gets even larger if we also take into account the space required for storing the loaded grammar and the NFA. Our initial implementation was not efficient enough for checking character-level grammars, so we needed a more compact item pair representation.

We looked for inspiration in the closely related field of model checking, where checkers also need to explore large state spaces. The first idea we adopted was to use bit strings to represent item pairs. In the current implementation we use one 64-bit Java long to represent a pair's identity (NFA states, flags and extension information), and another long for the traversal related information (*number*, *lowlink*, *alive* and *on stack*). The first long contains two stretches of bits to represent an NFA state pair, of which the size depends on the total number of states, and two bits for the conflict flags. The remainder of the bits can be reserved by the used item pair extensions to store additional information. The traversal related bit string contains two 31-bit fields to represent a pair's *number* and *lowlink*, and two single bit flags for *alive* and *on stack*. The maximum number of item pairs that can be created with this solution is therefore limited to 2^{31}, but this is more than enough for checking the programming language grammars shown in Table 6.1.

To limit the memory overhead of storing the already traversed item pairs we used a custom hashmap. Instead of using linked lists for its buckets like the `java.util.HashMap` does, it uses plain arrays to store the bit strings. This requires only a small logarithmic space overhead, instead of the constant 32 bytes for linked list elements. The arrays are searched for item pairs linearly, which does not result in noticeable speed losses if they are not that long. Therefore, the hashmap is re-hashed when needed to make sure the arrays do not grow beyond a certain size — which is currently 2048. This way, the item pair store makes efficient use of heap space without large performance penalties.

We also considered implementing other techniques used in model checking. For instance, we experimented with storing the bit strings in BDDs [Ake78]. However, the typical distribution of the bit strings and their relatively short lengths seem to be unsuitable for reaching good compression rates. Other space saving options would be to store chunks of bit strings to disk or compress them in memory. However, this will probably result in a loss of speed. We therefore chose for the array-backed item pair store described above, because it performed sufficiently well for the real world grammars tested in Chapter 5, as can be seen from Table 5.2.

6.3 Sentence Generator

In order to validate the effect of our character-level grammar filter, we also needed an exhaustive detection method for character-level grammars. We chose for a simple depth-first sentence generation technique, similar to that of AMBER. This section highlights the design decisions we made while developing the sentence generation tool.

6.3.1 Requirements

The main functional requirement for our exhaustive ambiguity detection tool was that it should find ambiguous sentences in character-level grammars as well as token-based grammars. Together with their parse trees, ambiguous sentences are the most descriptive proofs of the ambiguity of a grammar. To not report ambiguities that are already solved by the grammar developer, the tool should take disambiguation filters into account.

The main non-functional requirement of the exhaustive searching tool was that it should be as fast as possible. The higher the number of sentences that can be explored per minute, the higher the chance that ambiguities will be found. Furthermore, we wanted our tool to be suitable for short interactive use, as well as longer overnight runs.

6.3.2 Architecture and Design

To meet the above requirements we chose to implement a simple depth-first sentence generation technique, similar to that of AMBER. AMBER uses a modified Earley parser that generates sentences on the fly while parsing them. In our case however, we chose to use an SGLR parser, because knowledge about applying disambiguation filters in this parsing technique was readily available [vdBSVV02, Vis97]. Furthermore, LR parsers are generally faster than Earley parsers because they use a precomputed parse table.

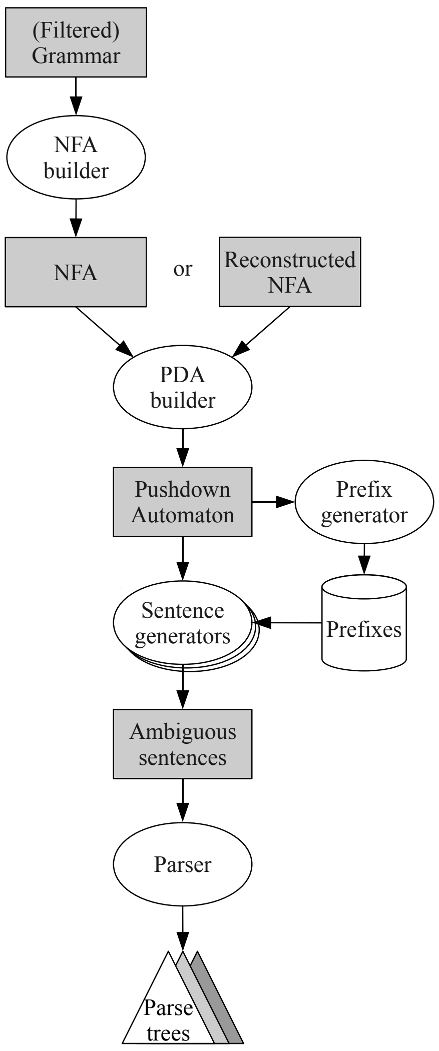


Figure 6.2: Architecture of AMBIDEXTER's parallel sentence generator.

Our second design decision was to generate sentences in a depth-first fashion. This requires only very little memory, whereas with breadth-first searching the memory usage grows with the number of sentences produced. From the results presented in Section 3.3.2, we see that the breadth-first CFG ANALYZER already used 1.3GB of memory after searching for 106 minutes. We therefore expect depth-first search to be a better candidate for running longer checks, even more so because the languages of character-level grammars are typically much larger than the token-based grammars explored in Section 3.3.2. Furthermore, depth-first searching is much easier to parallelize than breadth-first searching. We can therefore take advantage of present-day multicore processors for even better performance.

Figure 6.2 shows the architecture of the sentence generator. The sentence generation process can start either from a — possibly filtered — grammar or an NFA that was reconstructed after filtering. In case a grammar is given, its LR(0) NFA is generated from it, which in turn is converted into a deterministic LR(0) pushdown automaton. A reconstructed NFA is converted to deterministic form immediately.

The pushdown automaton is then used to generate sentences of a predefined length k. This can be done by a certain number of sentence generators in parallel. First, a single sentence generator is used to generate all prefixes of a given length $l < k$ which are stored in a set. After that, the parallel sentence generators take the prefixes from this set and generate all their completions up to length k. All ambiguous sentences that they find are parsed with a small parser and are reported to the user, together with their parse trees. These parse trees can then be searched for the causes of ambiguity in the grammar, either by hand, or with an expert system like DR. AMBIGUITY (see Chapter 7).

6.3.3 Implementation Details

The sentence generator is implemented as a normal SGLR parser (see [Vis97], Chapter 3), with the following modifications:

- After all stacks at a certain level are reduced, a set of shiftable characters is calculated.
- For each level such a set of candidate characters is stored, using an additional stack.
- The characters to shift are picked and removed from these sets, instead of read from an input string.
- If the maximum string length is reached, or if the set of candidates at a certain level is empty, all stacks are backtracked one level, and a new shift is tried.
- The garbage collection only removes stack nodes that are popped during backtracking.
- Follow restrictions do not have to be checked, because they are already propagated through the parse automaton before determinisation — see Algorithm 8 in Section 5.5.2.
- For speed, parse trees are not build during generation, but are obtained by re-parsing ambiguous strings.

These changes were relatively easy to implement, except for the selection of the candidate characters to shift at each level. Especially with character-level grammars, just gathering all characters that can be shifted from each stack can lead to an unneccesarily high number of generated sentences. For instance, consider the typical lexical definition `[a-zA-Z][a-zA-Z0-9_]+` for an identifier. This definition generates a very high number of possible identifiers. However, these do not all have to be explored if they will never lead to ambiguities.

Algorithm 10 Finding an approximation for the smallest set of characters that together continue the current gss with all possible combinations of paths.

```
function GET-SHIFTABLE-CHARACTERS(top ∈ $\mathcal{P}(Q)$) =
  // gather all shift actions of states in top
  $\mathcal{A} = \{q \overset{a}{\longmapsto} q' \in \text{shift} \mid q \in \text{top}\}$

  // find a set $B \subseteq T$ s.t. $\forall q_1 \overset{a}{\longmapsto} q_1', q_2 \overset{a}{\longmapsto} q_2' \in \mathcal{A} : \exists b \in B : q_1 \overset{b}{\longmapsto} q_1', q_2 \overset{b}{\longmapsto} q_2' \in \mathcal{A}$,
  // preferably the smallest
  $B = \emptyset$
  while $|\mathcal{A}| > 0$ do
      pick a $b \in T$ that occurs the most often in $\mathcal{A}$
      remove the elements $q \overset{a}{\longmapsto} q'$ from $\mathcal{A}$ for which hold that $q \overset{b}{\longmapsto} q'$
      add $b$ to $B$
  od
  return $B$
```

An ambiguity only appears in case two stacks merge upon reduction of the same substring. Therefore, it suffices to select only enough candidate characters such that every possible combination of stacks will be explored, and of course every individual stack as well. This will lead to the fastest detection of existing ambiguities.

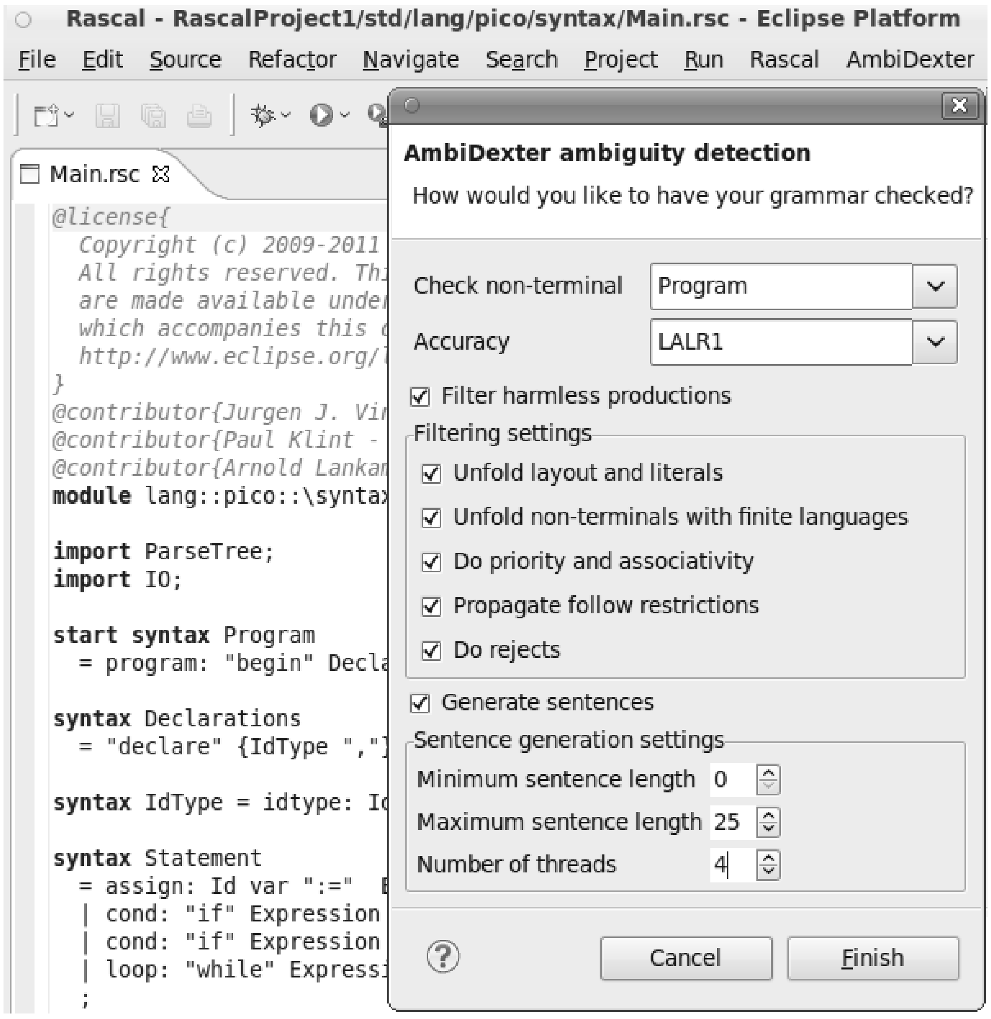
Selecting this minimum set of shiftable characters to cover all combinations of stack continuations is an instance of the hitting set problem [Kar72], which is NP-complete. In our current implementation we use a simple non-optimal computation shown in Algorithm 10. As of yet, it has never led to noticeable performance losses, because the number of stacks and possible shift transitions are usually quite low. More experiments are required to test whether finding the smallest possible set of candidates will pay off.

6.4 Usage

To get the fastest results, we advise the following strategy for finding ambiguities with AMBIDEXTER:

When the grammar under investigation contains a lot of ambiguous production rules, their ambiguity will propagate to other production rules as well during the approximative search for harmless productions. This reduces the chance of actual harmless rules being found. Therefore, we advise to quickly test for ambiguities with the sentence generator first. If ambiguities pop-up during this search, they can be solved first before trying the harmless rule filter.

When there are no 'low hanging' ambiguities to be found anymore, the sentence generation can be sped up by filtering harmless productions from the grammar. Depending on the size and shape of the grammar, the higher filtering precisions like LR(1) or grammar unfolding might require very large amounts of memory. Therefore, its best to start checking with lower precision settings first, and then gradually increase until a configuration is found that runs within acceptable time and memory limits. Judging from the figures in Table 5.2, this process


Rascal - RascalProject1/std/lang/pico/syntax/Main.rsc - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Rascal AmbiDexter
Main.rsc
Cancel
Finish

```
@license{
  Copyright (c) 2009-2011
  All rights reserved. Thi
  are made available unde
  which accompanies this 
  http://www.eclipse.org/
}
@contributor{Jurgen J. Vi
@contributor{Paul Klint -
@contributor{Arnold Lanka
module lang::pico::\synta

import ParseTree;
import IO;

start syntax Program
  = program: "begin" Decl

syntax Declarations
  = "declare" {IdType ","

syntax IdType = idtype: I

syntax Statement
  = assign: Id var ":="  E
  | cond: "if" Expression
  | cond: "if" Expression
  | loop: "while" Expressi
  ;
```

AmbiDexter ambiguity detection
How would you like to have your grammar checked?

Check non-terminal: Program
Accuracy: LALR1

☑ Filter harmless productions

Filtering settings
- ☑ Unfold layout and literals
- ☑ Unfold non-terminals with finite languages
- ☑ Do priority and associativity
- ☑ Propagate follow restrictions
- ☑ Do rejects

☑ Generate sentences

Sentence generation settings
- Minimum sentence length: 0
- Maximum sentence length: 25
- Number of threads: 4

Figure 6.3: Running AMBIDEXTER from within the RASCAL IDE in Eclipse.

will probably not take more than a couple of minutes for small to medium sized grammars.

If the grammar filter finds harmless production rules, the filtered grammar can be tested again with the sentence generator. Hopefully the sentence generation will now be faster so it can search for ambiguities at longer sentence lengths. If more ambiguities are found they can be removed from the grammar. Then, the grammar filter can be run again to see if more production rules are harmless. This process of filtering and sentence generation can be continued until it becomes unfeasible, or until the grammar filter finds the grammar to be unambiguous.

Package	**SLOC**
Grammar (incl. import/export)	3958
Automata	3408
Pair graph traversal	1817
Sentence generation	1009
Parser	888
Utilities	2039
Tests	563
Main	492
Total:	14180

Table 6.2: Sizes of AMBIDEXTER's packages in non-comment source lines of Java code (SLOC). Figures generated using David A. Wheeler's 'SLOCCount'.

6.5 Conclusion

In this chapter we have presented our tool implementation of AMBIDEXTER. The tool implements our grammar filtering techniques, as well as a sentence generator for token-based grammars and character-level grammars. We have discussed the architectural design and implementation details of these two parts, together with advice on how they can be used together in an optimal way.

The AMBIDEXTER tool is integrated in the RASCAL IDE in Eclipse. Figure 6.3 shows the wizard that can be used to configure it. For checking YACC or SDF2 grammars AMBIDEXTER can be run from the command-line as well. To get an indication of the development effort invested in the tool, Figure 6.2 shows the sizes of the AMBIDEXTER's packages measured in lines of code.

Chapter 7

Parse Forest Diagnostics with DR. AMBIGUITY

"A fool sees not the same tree that a wise man sees."
William Blake

Once an ambiguity detection method finds an unwanted ambiguity, it should be removed from the grammar. However, it is not always straightforward for the grammar developer to see which modifications solve his ambiguity and in which way. In this chapter we present an expert system called Dr. Ambiguity, that can automatically propose applicable cures for an ambiguous sentence. After giving an overview of different causes of ambiguity and ambiguity resolutions, the internals of Dr. Ambiguity are described. The chapter ends with a small experimental validation of the usefulness of the expert system, by applying it on a realistic character-level grammar for Java.

7.1 Introduction

This work is motivated by the use of parsers generated from general context-free grammars (CFGs). General parsing algorithms such as GLR and derivates [Tom85, vdBSVV02, AH99, BG06, Eco06], GLL [SJ10, JS11], and Earley [Ear70, Sch06] support parser generation for highly non-deterministic context-free grammars. The advantages of constructing parsers using such technology are that grammars may be modular and that real programming languages

This chapter was published in the proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011) [BV11]. It was co-authored by Jurgen Vinju.

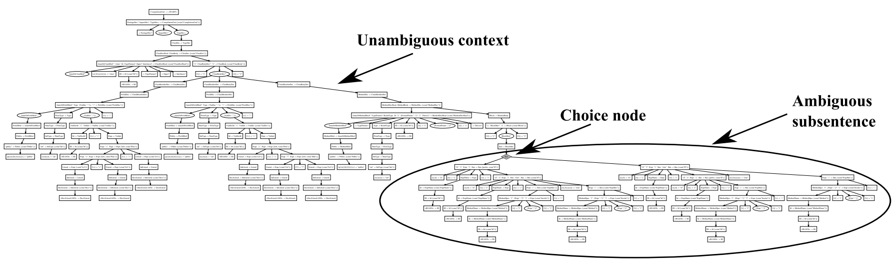


Figure 7.1: The complexity of a parse forest for a trivial Java class with one method; the indicated subtree is an ambiguous if-with-dangling-else issue (180 nodes, 195 edges).

(often requiring parser non-determinism) can be dealt with efficiently[1]. It is common to use general parsing algorithms in (legacy) language reverse engineering, where a language is given but parsers have to be reconstructed [LV01], and in language extension, where a base language is given which needs to be extended with unforeseen syntactical constructs [BTV06].

The major disadvantage of general parsing is that multiple parse trees may be produced by a parser. In this case, the grammar was not only non-deterministic, but also *ambiguous*. We say that a grammar is ambiguous if it generates more than one parse tree for a particular input sentence. Static detection of ambiguity in CFGs is undecidable in general [Can62, Flo62, CS63].

It is not an overstatement to say that ambiguity is the Achilles' heel of CFG-general parsing. Most grammar engineers who are building a parser for a programming language intend it to produce a single tree for each input program. They use a general parsing algorithm to efficiently overcome problematic non-determinism, while ambiguity is an unintentional and unpredictable side-effect. Other parsing technologies, for example Ford's PEG [For04] and Parr's LL(*) [PF11], do not report ambiguity. Nevertheless, these technologies also employ disambiguation techniques (ordered choice, dynamic lookahead). In combination with a debug-mode that does produce all derivations, the results in this chapter should be beneficial for these parsing techniques as well. It should help the user to intentionally select a disambiguation method. In any case, the point of departure for this chapter is any parsing algorithm that will produce all possible parse trees for an input sentence.

In Chapters 3–5 we present a fast ambiguity detection approach that combines approximative and exhaustive techniques. The output of this method are the ambiguous sentences found in the language of a tested grammar. Nevertheless, this is only a observation that the patient is ill, and now we need a cure. We therefore will diagnose the sets of parse trees produced for specific ambiguous sentences. The following is a typical grammar engineering scenario:

1. While testing or using a generated parser, or after having run a static ambiguity detection tool, we discover that one particular sentence leads to a set of multiple parse trees. This

[1]Linear behavior is usually approached and most algorithms can obtain cubic time worst time complexity [Lan74]

```
If (                                  <
ExprName ( Id ( "a" ) ),              <
IfElse (                                IfElse (
                                      > ExprName ( Id ( "a" ) ),
                                      > If (
ExprName ( Id ( "b" ) ),                ExprName ( Id ( "b" ) ),
ExprStm (                               ExprStm (
Invoke (                                Invoke (
Method ( MethodName ( Id ( "a" ) ) ),   Method ( MethodName ( Id ( "a" ) ) ),
[                                       [
] ) ),                                | ] ) ) ),
ExprStm (                               ExprStm (
Invoke (                                Invoke (
Method ( MethodName ( Id ( "b" ) ) ),   Method ( MethodName ( Id ( "b" ) ) ),
[                                       [
] ) ) ) )                             | ] ) ) )
```

Figure 7.2: Using `diff -side-by-side` to diagnose a trivial ambiguous syntax tree for a dangling else in Java (excerpts of Figure 7.1).

set is encoded as a single parse forest with choice nodes where sub-sentences have alternative sub-trees.

2. The parser reports the location in the input sentence of each choice node. Note that such choice nodes may be nested. Each choice node might be caused by a different ambiguity in the CFG.

3. The grammar engineer extracts an arbitrary ambiguous sub-sentence and runs the parser again using the respective sub-parser, producing a set of smaller trees.

4. Each parse tree of this set is visualized on a 2D plane and the grammar engineer spots the differences, or a (tree) diff algorithm is run by the grammar engineer to spot the differences. Between two alternative trees, either the shape of the tree is totally different (rules have moved up/down, left/right), or completely different rules have been used, or both. As a result the output of diff algorithms and 2D visualizations typically require some effort to understand. Figure 7.1 illustrates the complexity of an ambiguous parse forest for a 5 line Java program that has a dangling else ambiguity. Figure 7.2 depicts the output of diff on a strongly simplified representation (abstract syntax tree) of the two alternative parse trees for the same nested conditional. Realistic parse trees are not only too complex to display here, but are often too big to visualize on screen as well. The common solution is to prune the input sentence step-by-step to eventually reach a very minimal example that still triggers the ambiguity but is small enough to inspect.

5. The grammar engineer hopefully knows that for some patterns of differences there are typical solutions. A solution is picked, and the parser is regenerated.

6. The smaller sentence is parsed again to test if only one tree (and which tree) is produced.

7. The original sentence is parsed again to see if all ambiguity has been removed or perhaps more diagnostics are needed for another ambiguous sub-sentence. Typically, in programs one cause of ambiguity would lead to several instances distributed over the source file. One disambiguation may therefore fix more "ambiguities" in a source file.

The issues we address in this chapter are that the above scenario is (a) an expert job, (b) time consuming and (c) tedious. We investigate the invention of an *expert system* that can automate finding a concise grammar-level explanation for any choice node in a parse forest and propose a set of solutions that will eliminate it. This expert system is shaped as a set of algorithms that analyze sets of alternative parse trees, simulating what an expert would do when confronted with an ambiguity.

The contributions of this chapter are an overview of common causes of ambiguity in grammars for programming language (Section 7.3), an automated tool (Dr. Ambiguity) that diagnoses parse forests to propose one or more appropriate disambiguation techniques (Section 7.4) and an initial evaluation of its effectiveness (Section 7.5). In 2006 we published a manual [Vin11] to help users disambiguate SDF2 grammars. This well-read manual contains recipes for solving ambiguity in grammars for programming languages. Dr. Ambiguity automates all tasks that users perform when applying the recipes from this manual, except for finally adding the preferred disambiguation declaration.

7.1.1 Preliminaries

In this chapter we will use the following definitions: A *context-free grammar* G is defined as a 4-tuple (T, N, P, S), namely finite sets of terminal symbols T and non-terminal symbols N, production rules P in $A \times (T \cup N)^*$ written like $A \rightarrow \alpha$, and a start symbol S. A *sentential form* is a finite string in $(T \cup N)^*$. A *sentence* is a sentential form without non-terminal symbols. An ε denotes the empty string. We use the other lowercase greek characters $\alpha, \beta, \gamma, \ldots$ for variables over sentential forms, uppercase roman characters for non-terminals $(A, B, \ldots)$ and lowercase roman characters and numerical operators for terminals $(a, b, +, -, *, /)$. By applying production rules as substitutions we can generate new sentential forms. One substitution is called a *derivation step*, e.g. $\alpha A \beta \Rightarrow \alpha\gamma\beta$ with rule $A \rightarrow \gamma$. We use $\Rightarrow^*$ to denote sequences of derivation steps. A *full derivation* is a sequence of production rule applications that starts with a start symbol and ends with a sentence. The *language* of a grammar is the set of all sentences derivable from S. In a *bracketed derivation* [GH67] we record each application of a rule by a pair of brackets, for example $S \Rightarrow (\alpha E \beta) \Rightarrow (\alpha(E+E)\beta) \Rightarrow (\alpha((E*E)+E)\beta)$. Brackets are (implicitly) indexed with their corresponding rule.

A *non-deterministic derivation sequence* is a derivation sequence in which a $\diamond$ operator records choices between different derivation sequences. I.e. $\alpha \Rightarrow (\beta) \diamond (\gamma)$ means that either β or γ may be derived from α using a single derivation step. Note that β does not necessarily need to be different from γ. An example non-deterministic derivation is $E \Rightarrow (E+E) \diamond (E*E) \Rightarrow (E+(E*E)) \diamond ((E+E)*E)$. A *cyclic derivation sequence* is any sequence $\alpha \Rightarrow^{+} \alpha$, which is only possible by applying rules that do not have to eventually generate terminal symbols, such as $A \rightarrow A$ and $A \rightarrow \varepsilon$.

A *parse tree* is an (ordered) *finite* tree representation of a bracketed *full* derivation of a specific sentence. Each pair of brackets is represented by an internal node labeled with the rule that was applied. Each leaf node is labeled with a terminal. This implies that the leafs of a parse tree form a sentence. Note that a single parse tree may represent several equivalent derivation sequences. Namely in sentential forms with several non-terminals one may always choose which non-terminal to expand first. From here on we assume a canonical left-most form for such equivalent derivation sequences, in which expansion always occurs at the left-most non-terminal in a sentential form.

A *parse forest* is a set of parse trees possibly extended with *ambiguity nodes* for each use of choice ($\diamond$). Like parse trees, parse forests are limited to represent full derivations of a *single* sentence, each child of an ambiguity node is a derivation for the same sub-sentence. One such child is called an *alternative*. For simplicity's sake, and without loss of generality, we assume that all ambiguity nodes have exactly two alternatives.

A parse forest is *ambiguous* if it contains at least one ambiguity node. A sentence is *ambiguous* if its parse forest is ambiguous. A grammar is *ambiguous* if it can generate at least one ambiguous sentence. An *ambiguity* in a sentence is an ambiguity node. An *ambiguity* of a grammar is the cause of such aforementioned ambiguity. We define *cause of ambiguity* precisely in Section 7.3. Note that *cyclic derivation* sequences can be represented by parse forests by allowing them to be graphs instead of just trees [Rek92].

A *recognizer* for G is a terminating function that takes any sentence α as input and returns true if and only if $S \Rightarrow^{*} \alpha$. A *parser* for G is a terminating function that takes any finite sentence α as input and returns an error if the corresponding recognizer would not return true, and otherwise returns a *parse forest* for α. A *disambiguation filter* is a function that takes a parse forest for α and returns a smaller parse forest for α [KV94]. A *disambiguator* is a function that takes a parser and returns a parser that produces smaller parse forests. Disambiguators may be implemented as parser actions, or by parser generators that take additional disambiguation constructs as input [vdBSVV02]. We use the term *disambiguation* for both disambiguation filters and disambiguators.

7.2 Solutions to Ambiguity

There are basically two kinds of solutions to removing ambiguity from grammars. The first involves restructuring the grammar to accept the same set of sentences but using different rules. The second leaves the grammar as-is, but adds disambiguations (see above). Although grammar restructuring is a valid solution direction, we restrict ourselves to disambiguations described below. The benefit of disambiguation as opposed to grammar restructuring is that the shape of the rules, and thus the shape of the parse trees remains unchanged. This allows language engineers to maintain the intended semantic structure of the language, keeping parse trees directly related to abstract syntax trees (or even synonymous) [HHKR89].

Any solution may be *language preserving*, or not. We may change a grammar to have it generate a different language, or we may change it to generate the same language differently. Similarly, a disambiguation may remove sentences from a language, or simply remove some ambiguous derivation without removing a sentence. This depends on whether or not the filter is applied always in the context of an ambiguous sentence, i.e. whether another tree is guaranteed to be left over after a certain tree is filtered. It may be hard for a language engineer who adds a disambiguation to understand whether it is actually language preserving. Whether or not it is good to be language preserving depends entirely on ad-hoc requirements. We therefore do not answer this question. Where possible, we do indicate whether adding a certain disambiguation is expected to be language preserving. Proving this property is out-of-scope.

Solving ambiguity is sometimes confused with making parsers deterministic. From the perspective of this chapter, non-determinism is a non-issue. We focus solely on solutions to ambiguity.

We now quote a number of disambiguation methods here. Conceptually, the following list contains nothing but disambiguation methods that are commonly supported by lexer and parser generators [ASU86]. Still, the precise semantics of each method we present here may be specific to the parser frameworks of SDF2 [HHKR89, Vis97] and Rascal [KvdSV11]. In particular, some of these methods are specific to *scannerless* parsing, where a context-free grammar specifies the language down to the character level [Vis97, SC89]. We recommend [BBV07], to appreciate the intricate differences between semantics of operator priority mechanisms between parser generators.

Priority disallows certain direct edges between pairs of rules in parse trees in order to affect operator priority. For instance, the production for the + operator may not be a direct child of the $\star$ production [vdBSVV02].
Formally, let a priority relation $>$ be a partial order between recursive rules of an expression grammar. If $A \rightarrow \alpha_1 A \alpha_2 > A \rightarrow \beta_1 A \beta_2$ then all derivations $\gamma A \delta \Rightarrow \gamma(\alpha_1 A \alpha_2)\delta \Rightarrow \gamma(\alpha_1(\beta_1 A \beta_2)\alpha_2)$ are illegal.

Associativity is similar to priority, but father and child are the same rule. It can be used to affect operator associativity. For instance, the production of the + operator may not be a direct *right* child of itself because + is left associative [vdBSVV02]. Left and right associativity are duals, and *non-assocativity* means no nesting is allowed at all. Formally, if a recursive rule $A \rightarrow A\alpha A$ is defined left associative, then any derivation $\gamma A \delta \Rightarrow \gamma(A\alpha A)\delta \Rightarrow \gamma(A\alpha(A\alpha A))\delta$ is illegal.

Offside disallows certain derivations using the would-be indentation level of an (indirect) child. If the child is "left" of a certain parent, the derivation is filtered [Lan66]. One example formalization is to let $\Pi(x)$ compute the start column of the sub-sentence generated by a sentential form x and let $>$ define a partial order between production rules. Then, if $A \rightarrow \alpha_1 X \alpha_2 > B \rightarrow \beta$ then all derivations $\gamma A \delta \Rightarrow \gamma(\alpha_1 X \alpha_2)\delta \Rightarrow^* \gamma(\alpha_1(\ldots(\beta)\ldots)\alpha_2)\delta)$ are illegal if $\Pi(\beta) < \Pi(\alpha_1)$. Parsers may employ subtly different offside disambiguators, depending on how Π is defined for each different language or even for each different production rule within a language.

Preference removes a derivation, but only if another one of higher preference is present. Again, we take a partial ordering $>$ that defines preference between rules for the same non-terminal. Let $A \rightarrow \alpha > A \rightarrow \beta$, then from all derivations $\gamma A \delta \Rightarrow \gamma((\alpha) \diamond (\beta))\delta$ we must remove (β) to obtain $A \Rightarrow \gamma(\alpha)\delta$.

Reserve disallows a fixed set of terminals from a certain (non-)terminal, commonly used to reserve keywords from identifiers. Let K be a set of sentences and let I be a non-terminal from which they are declared to be reserved. Then, for every $\alpha \in K$, any derivation $I \Rightarrow^* \alpha$ is illegal.

Reject disallows the language of a certain non-terminal from that of another one. This may be used to implement **Reserve**, but it is more powerful than that [vdBSVV02]. Let $(I$ **-** $R)$ declare that the non-terminal R is rejected from the non-terminal I. Then any derivation sequence $I \Rightarrow^* \alpha$ is illegal if and only if $R \Rightarrow^* \alpha$.

Not Follow/Precede declarations disallow derivation steps if the generated sub-sentence in its context is immediately followed/preceded by a certain terminal. This is used to affect longest match behavior for regular languages, but also to solve "dangling else" by not allowing the short version of `if`, when it would be immediately followed by `else` [vdBSVV02]. Formally, we define follow declaration as follows. Given $A \Rightarrow^* \alpha$ and a declaration A **not-follow** β, where β is a sentence, any derivation $S \Rightarrow^* \gamma A \beta \delta \Rightarrow^* \gamma(\alpha)\beta\delta$ is illegal. We should mention that **Follow** declarations may simulate the effect of "shift before reduce" heuristics that deterministic — LR, LALR — parsers use when confronted with a shift/reduce conflict.

Dynamic Reserve disallows a dynamic set of sub-sentences from a certain non-terminal, i.e. using a symbol table [ASU86]. The semantics is similar to **Reject**, where the set R is dynamically changed as certain derivations (i.e. type declarations) are applied.

Types removes certain type-incorrect sub-trees using a type-checker, leaving correctly typed trees as-is [BVVV05]. Let $C(d)$ be true if and only if derivation d (represented by a tree) is a type-correct part of a program. Then all derivations $\gamma A \delta \Rightarrow \gamma(\alpha)\delta$ are illegal if $C(\alpha)$ is false.

Heuristics There are many kinds of heuristic disambiguation that we bundle under a single definition here. The preference of "Islands" over "Water" in island grammars is an example [Moo01]. Preference filters are sometimes generalized by counting the number of preferred rules as well [vdBSVV02]. Counting rules is used sometimes to choose a "simplest" derivation, i.e. the most shallow trees are selected over deeper ones. Formally, Let $C(d)$ be any function that maps a derivation (parse tree) to an integer. If $C(A \Rightarrow \alpha) > C(A \Rightarrow \beta)$ then from all derivations $A \Rightarrow^* (\alpha) \diamond (\beta)$ we must remove (β) to obtain $A \Rightarrow (\alpha)$.

Not surprisingly, each kind of disambiguation characterizes certain properties of derivations. In the following section we link such properties to causes of ambiguity. Apart from **Types** and **Heuristics** (which are too general to automatically report specific suggestions for), we can then link the causes explicitly back to the solution types.

7.3 Causes of Ambiguity

Ambiguity is caused by the fact that the grammar can derive the same sentence in at least two ways. This is not a particularly interesting cause, since it characterizes all ambiguity in general. We are interested in explaining to a grammar engineer what is wrong for a very particular grammar and sentence and how to possibly solve this particular issue. We are interested in the *root causes* of specific occurrences of choice nodes in parse forests.

For example, let us consider a particular grammar for the C programming language for which the sub-sentence "`{S * b;}`" is ambiguous. In one derivation it is a block of a single statement that multiplies variables `S` and `b`, in another it is a block of a single declaration of a pointer variable `b` to something of type `S`. From a language engineer's perspective, the causes of this ambiguous sentence are that:

- "`*`" is used both in the rule that defines multiplication, and in the rule that defines pointer types, *and*
- type names and variable names have the same lexical syntax, *and*
- blocks of code start with a possibly empty list of declarations and end with a possibly empty list of statements, *and*
- both statements and declarations end with "`;`".

The conjunction of all these causes explains us why there is an ambiguity. The removal of just one of them fixes it. In fact, we know that for C the ambiguity was fixed by introducing a disambiguator that reserves any declared type name from variable names using a symbol table at parse time, effectively removing the second cause.

We now define a *cause* of an ambiguity in a sub-sentence to be the existence of any edge that is in the parse tree of one alternative of an ambiguity node, but not in the other. In other words, each *difference* between two alternative parse trees in a forest is *one cause* of the ambiguity. For example, two parse tree edges differ if they represent the application of a different production rule, span a different part of the ambiguous sub-sentence, or are located at different heights in the tree.

We define an *explanation* of an ambiguity in a sentence to be the conjunction of all causes of ambiguity in a sentence. An explanation is a set of differences. We call it an explanation because an ambiguity exists if and only if all of its causes exist. A *solution* is any change to the grammar, addition of a disambiguation filter or use of a disambiguator that removes at least one of the causes.

Some causes of ambiguity may be solvable by the disambiguation methods defined in Section 7.2, some may not. Our goals are therefore to first explain the cause of ambiguity as concisely as possible, and then if possible propose a palette of applicable disambiguations. Note that even though the given common disambiguations have limited scope, disambiguation in general is always possible by writing a disambiguation filter in any computationally complete programming language.

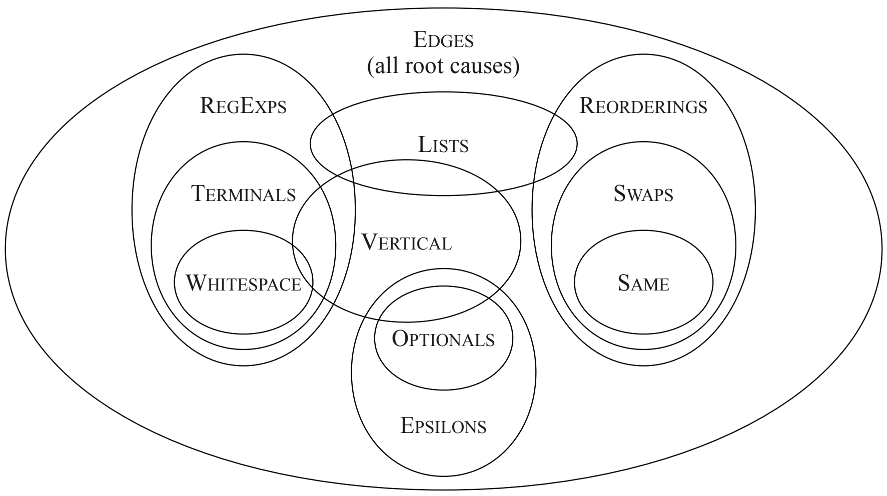


Figure 7.3: Euler diagram showing the categorization of parse tree differences.

7.3.1 Classes of Parse Tree Differences

Having defined ambiguity and the causes thereof, we can now categorize different kinds of causes into classes of differences between parse trees. The difference classes are the theory behind the workings of Dr. Ambiguity (Section 7.5). Figure 7.3 summarizes the cause classes that we will identify in the following.

For completeness we should explain that ambiguity of CFGs is normally bisected into a class called HORIZONTAL ambiguity and a class called VERTICAL ambiguity [BGM10, AL90, Sch01]. VERTICAL contains all the ambiguity that causes parse forests that have two different production rules directly under a choice node. For instance, all edges of derivation sequences of form $\gamma A\delta \Rightarrow \gamma((\alpha) \diamond (\beta))\delta$ provided that $\alpha \neq \beta$ are in VERTICAL. VERTICAL clearly identifies a difference class, namely the trees with different edges directly under a choice node.

HORIZONTAL ambiguity is defined to be all the other ambiguity. HORIZONTAL does not identify any difference class, since it just implies that the two top rules are the same. Our previous example of ambiguity in a C grammar is an example of such ambiguity. We conclude that in order to obtain full explanations of ambiguity the HORIZONTAL/VERTICAL dichotomy is not detailed enough. VERTICAL provides only a partial explanation (a single cause), while HORIZONTAL provides no explanations at all.

We now introduce a number of difference classes with the intention of characterizing differences which can be solved by one of the aforementioned disambiguation methods. Each element in a different class points to a single cause of ambiguity. A particular disambiguation method may be applicable in the presence of elements in one or more of these classes.

The EDGES class is the universe of all difference classes. In EDGES are all single derivation steps (equivalent to edges in parse forests) that occur in one alternative but not in the other. If no such derivation steps exist, the two alternatives are exactly equal. Note that EDGES $=$ HORIZONTAL $\cup$ VERTICAL.

The TERMINALS class contains all parse tree edges to non-ε leafs that occur in one alternative but not in the other. If an explanation contains a difference in TERMINALS, we know that the alternatives have used different terminal tokens—or in the case of scannerless, different character classes—for the same sub-sentences. This is sometimes called *lexical ambiguity*. If no differences are in TERMINALS, we know that the terminals used in each alternative are equal.

The WHITESPACE class ($\subset$ TERMINALS) simply identifies the differences in TERMINALS that produce terminals consisting of nothing but spaces, tabs, newlines, carriage returns or linefeeds.

The REGEXPS class contains all edges of derivation steps that replace a non-terminal by a sentential form that generates a regular language, occurring in one derivation but not in the other, i.e. $A \Rightarrow (\rho)$ where ρ is a regular expression over terminals. Of course, TERMINALS $\subset$ REGEXPS. In character level grammars (scannerless [vdBSVV02]), the REGEXPS class often represents lexical ambiguity. Differences in REGEXPS may point to solutions such as **Reserve**, **Follow** and **Reject**, since longest match and keyword reservation are typical solution scenarios for ambiguity on the lexical level.

In the SWAPS class we put all edges that have a corresponding edge in the other alternative of which the source and target productions are equal but have swapped order. For instance, the lower edges in the parse tree fragment $((E * E) + E) \diamond (E * (E + E))$ are in SWAPS. If all differences are in SWAPS, the set of rules used in the derivations of both alternatives are the same and each rule is applied the same number of times—only their order of application is different.

The SAME class is the subset of edges in SWAPS that have the same source and target productions. In this case, the only difference between two corresponding edges are the substrings they span. For instance, the lower edges in the parse tree fragment $((E+E)+E) \diamond (E+(E+E))$ are in SAME. Differences in this class typically require **Associativity** solutions.

The REORDERINGS class generalizes SWAPS with more than two rules to permute. This may happen when rules are not directly recursive, but mutually recursive in longer chains. Differences in REORDERINGS or SWAPS obviously suggest a **Priority** solution, but especially for non-directly recursive derivations **Priority** will not work. For example, the notorious "dangling else" issue [ASU86] generates differences in application order of mutually recursive statements and lists of statements. For some grammars, a difference in REORDERINGS may also imply a difference in VERTICAL, i.e. a choice between an `if` with an `else` and one without. In this case a **Preference** solution would work. Some grammars (e.g. the IBM COBOL VS2 standard) only have differences in HORIZONTAL and REORDERINGS. In this case a **Follow** solution may prevent the use of the `if` without the `else` if there is an `else` to be parsed. Note that the **Offside** solution is an alternative method to remove ambiguity caused by REORDERINGS. Apparently, we need even smaller classes of differences before we can be more precise about suggesting a solution.

The LISTS class contains differences in the length of certain lists between two alternatives. For instance, we consider rules $L \rightarrow LE$ and observe differences in the amount of times these rules are applied by the derivation steps in each alternative. More precisely, for any L and E with the rule $L \rightarrow LE$ we find chains of edges for derivation sequences $\alpha L\beta \Rightarrow \alpha LE\beta \Rightarrow\Rightarrow^* \alpha LE^+\beta$, and compute their length. The edges of such chains of different lengths in the two alternatives are members of LISTS. Examples of ambiguities caused by LISTS are those caused by not having “longest match” behavior: an identifier “`aa`” generated using the rules $I \rightarrow a$ and $I \rightarrow I\ a$ may be split up in two shorter identifiers “`a`” and “`a`” in another alternative. We can say that LISTS $\cap$ REGEXPS $\neq \emptyset$.

Note that differences in LISTS $\cap$ REORDERINGS indicate a solution towards **Follow** or **Offside** for they flag issues commonly seen in dangling constructs. On the other hand a difference in LISTS $\setminus$ REORDERINGS indicates that there must be another important difference to explain the ambiguity. The “‘`{S * a}`”’ ambiguity in C is of that sort, since the length of declaration and statement lists differ between the two alternatives, while also differences in TERMINALS are necessary.

The EPSILONS class contains all edges to ε leaf nodes that only occur in one of the alternatives. They correspond to derivation steps $\alpha A\beta \Rightarrow \alpha()\beta$, using $A \rightarrow \varepsilon$. All cyclic derivations are caused by differences in EPSILONS because one of the alternatives of a cyclic ambiguity must derive the empty sub-sentence, while the other eventually loops back. However, differences in EPSILONS may also cause other ambiguity than cyclic derivations.

The OPTIONALS class ($\subset$ EPSILONS) contains all edges of a derivation step $\alpha A\beta \Rightarrow \alpha()\beta$ that only exist in one alternative, while a corresponding edge of $\delta A\zeta \Rightarrow \delta(\gamma)\zeta$ only exists in the other alternative. Problems that are solved using longest match (**Follow**) are commonly caused by optional whitespace for example.

7.4 Diagnosing Ambiguity

We provide an overview of the architecture and the algorithms of DR. AMBIGUITY in this section. In Section 7.5 we demonstrate its output on example parse forests for an ambiguous Java grammar.

7.4.1 Architecture

Figure 7.4 shows an overview of our diagnostics tool: DR. AMBIGUITY. We start from the parse forest of an ambiguous sentence that is either encountered by a language engineer or produced by a static ambiguity detection tool like AMBIDEXTER. Then, either the user

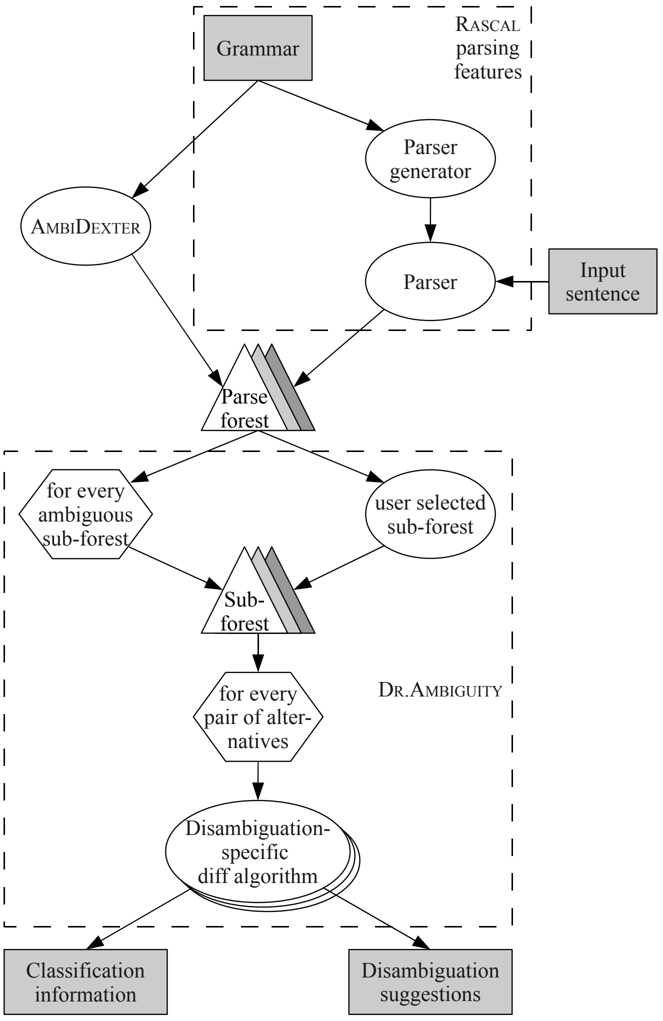

RASCAL parsing features
Grammar
Parser generator
AMBIDEXTER
Parser
Input sentence
Parse forest
for every ambiguous sub-forest
user selected sub-forest
Sub-forest
for every pair of alternatives
DR.AMBIGUITY
Disambiguation-specific diff algorithm
Classification information
Disambiguation suggestions

Figure 7.4: Contextual overview (input/output) of Dr. Ambiguity.

points at a specific sub-sentence[2], or DR. AMBIGUITY finds all ambiguous sub-sentences (e.g. choice nodes) and iterates over them. For each choice node, the tool then generates all unique combinations of two children of the choice node and applies a number of specialized diff algorithms to them.

Conceptually there exists one diff algorithm per disambiguation method (Section 7.2).

[2]We use Eclipse IMP [CFJ+09] as a platform for generating editors for programming languages defined using RASCAL [KvdSV11]. IMP provides contextual pop-up menus.

However, since some methods may share intermediate analyses there is some additional intermediate stages and some data-dependency that is not depicted in Figure 7.4. These intermediate stages output information messages about the larger difference classes that are to be analyzed further if possible. This output is called "Classification Information" in Figure 7.4. The other output, called "Disambiguation Suggestions" is a list of specific disambiguation solutions (with reference to specific production rules from the grammar).

If no specific or meaningful disambiguation method is proposed the classification information will provide the user with useful information on designing an ad-hoc disambiguation.

DR. AMBIGUITY is written in the RASCAL domain specific programming language [KvdSV11]. This language is specifically targeted at analysis, transformation, generation and visualization of source code. Parse trees are a built-in data-type which can be queried using (higher order) pattern matching, visiting and set, list and map comprehension facilities. To understand some of the RASCAL snippets in this section, please familiarize yourself with this definition for parse trees (as introduced by [Vis97]):

```
data Tree
  = appl(Production prod, list[Tree] args) // production nodes
  | amb(set[Tree] alternatives)            // choice nodes
  | char(int code);                        // terminal leaves
data Production
  = prod(Symbol lhs, list[Symbol] rhs, Attributes atts); // rules
```

DR. AMBIGUITY, in total, is 250 lines of RASCAL code that queries and traverses terms of this parse tree format. The count includes source code comments. It is slow on big parse forests[3], which is why the aforementioned user-selection of specific sub-sentences is important.

7.4.2 Algorithms

Here we show some of the actual source code of DR. AMBIGUITY.

First, the following two small functions iterate over all (deeply nested) choice nodes (`amb`) and over all possible pairs of alternatives. This code uses deep match (`/`), set matching, and set or list comprehensions. Note that the match operator (`:=`) iterates over all possible matches of a value against a pattern, thus generating all different bindings for the free variables in the pattern. This feature is used often in the implementation of DR. AMBIGUITY.

```
list[Message] diagnose(Tree t) {
  return [findCauses(x) | x <- {a | /a:amb(_) := t}];
}
list[Message] findCauses(Tree a) {
  return [findCauses(x, y) | {x, y, _*} := a.alternatives];
}
```

[3]The current implementation of RASCAL lacks many trivial optimizations.

The following functions each implement one of the diff algorithms from Figure 7.4. The following two (slightly simplified[4]) functions detect opportunities to apply priority or associativity disambiguations.

```
list[Message] priorityCauses(Tree x, Tree y) {
  if (/appl(p,[appl(q,_),_*]) := x,
      /t:appl(q,[_*,appl(p,_)]) := y, p != q) {
    return [error("You␣might␣add␣this␣priority␣rule:␣<p>␣\>␣<q>")
           ,error("You␣might␣add␣this␣associativity␣group:"
                  + "␣left␣(<p>␣|␣<q>)")];
  }
  return [];
}
list[Message] associativityCauses(Tree x, Tree y) {
  if (/appl(p,[appl(p,_),_*]) := x,
      /Tree t:appl(p,[_*,appl(p,_)]) := y) {
    return [error("You␣might␣add␣this␣associativity␣declaration:"
                  + "␣left␣<p>")];
  }
  return [];
}
```

Both functions “simultaneously” search through the two alternative parse trees p and q, detecting a vertical swap of two different rules p and q (priority) or a horizontal swap of the same rule p under itself (associativity).

This slightly more involved function detects dangling-else and proposes a follow restriction as a solution:

```
list[Message] danglingCauses(Tree x, Tree y) {
  if (appl(p,/appl(q,_)) := x, appl(q,/appl(p,_)) := y) {
    return danglingOffsideSolutions(x, y)
         + danglingFollowSolutions(x, y);
  }
  return [];
}
list[Message] danglingFollowSolutions(Tree x, Tree y) {
  if (prod(_, rhs, _) := x.prod,
      prod(_, [prefix*, _, l:lit(_), more*], _) := y.prod,
      rhs == prefix) {
    return [error("You␣might␣add␣a␣follow␣restriction␣for␣<l>"
                  + "␣on:␣<x.prod>")];
  }
  return [];
}
```

[4]We have removed references to location information that facilitates IDE features.

The function `danglingCauses` detects re-orderings of arbitrary depth, after which the outermost productions are compared by `danglingFollowSolutions` to see if one production is a prefix of the other.

DR. AMBIGUITY currently contains 10 such functions, and we will probably add more. Since they all employ the same style —(a) simultaneous deep match, (b) production comparison and (c) construction of a feedback message— we have not included more source code[5].

7.4.3 Discussion on Correctness

These diagnostics algorithms are typically wrong if one of the following four errors is made:

- no suggestion is given, even though the ambiguity is of a quite common kind;
- the given suggestion does not resolve any ambiguity;
- the given suggestion removes both alternatives from the forest, resulting in an empty forest (i.e., it removes the sentence from the language and is thus not language preserving);
- the given suggestion removes the proper derivation, but also unintentionally removes sentences from the language.

We address the first threat by demonstrating DR. AMBIGUITY on Java in Section 7.5. However, we do believe that the number of detection algorithms is open in principle. For instance, for any disambiguation method that characterizes a specific way of solving ambiguity we may have a function to analyze the characteristic kind of difference. As an "expert tool", automating proposals for common solutions in language design, we feel that an open-ended solution is warranted. More disambiguation suggestion algorithms will be added as more language designs are made. Still, in the next section we will demonstrate that the current set of algorithms is complete for all disambiguations applied to a scannerless definition of Java 5 [BVdGD11], which actually uses all disambiguations offered by SDF2.

For the second and third threats, we claim that no currently proposed solution removes both alternatives and all proposed solutions remove at least one. This is the case because each suggestion is solely deduced from a *difference* between two alternatives, and each disambiguation removes an artifact that is only present in one of the alternatives. We are considering to actually prove this, but only after more usability studies.

The final threat is an important weakness of DR. AMBIGUITY, inherited from the strength of the given disambiguation solutions. In principle and in practice, the application of rejects, follow restrictions, or semantic actions in general renders the entire parsing process stronger than context-free. For example, using context-free grammars with additional disambiguations we may decide language membership of many non-context-free languages. On the one hand, this property is beneficial, because we want to parse programming languages that have no or awkward context-free grammars. On the other hand, this property is cumbersome, since we

[5]The source code is available at `http://svn.rascal-mpl.org/rascal/trunk/src/org/rascalmpl/library/Ambiguity.rsc`.

Disambiguations	**Grammar snippet (Rascal notation)**
7 levels of expression priority	`Expr = Expr "++"`
	`     > "++" Expr`
1 father/child removal	`MethodSpec = Expr callee "." TypeArgs?`
	`   Id { if (callee is ExprName) filter;`
	`}`
9 associativity groups	`Expr = left ( Expr "+" Expr`
	`           | Expr "-" Expr )`
10 rejects	`ID = ( [$A-Z_a-z] [$0-9A-Z_a-z]* ) \`
	`     Keywords`
30 follow restrictions	`"+" = [\+] !>> [\+]`
4 vertical preferences	`Stm = @prefer "if" "(" Expr ")" Stm`
	`    | "if" "(" Expr ")" Stm "else" Stm`

Table 7.1: Disambiguations applied in the Java 5 grammar [BVdGD11]

can not easily predict or characterize the effect of a disambiguation filter on the accepted set of sentences.

Only in the SWAPS class, and its sub-classes we may be (fairly) confident that we do not remove unforeseen sentences from a language by introducing a disambiguation. The reason is that if one of the alternatives is present in the forest, the other is guaranteed to be also there. The running assumption is that the other derivation has not been filtered by some other disambiguation. We might validate this assumption automatically in many cases. So, application of priority and associativity rules suggested by DR. AMBIGUITY are safe if no other disambiguations are applied.

7.5 Demonstration

In this section we evaluate the effectiveness of DR. AMBIGUITY as a tool. We applied DR. AMBIGUITY to a scannerless (character level) grammar for Java [BVdGD11, BTV06]. This well tested grammar was written in SDF2 by Bravenboer et al. and makes ample use of its disambiguation facilities. For the experiment here we automatically transformed the SDF2 grammar to RASCAL's EBNF-like form.

Table 7.1 summarizes which disambiguations were applied in this grammar. RASCAL supports all disambiguation features of SDF2, but some disambiguation filters are implemented as libraries rather than built-in features. The `@prefer` attribute is interpreted by a library function for example. Also, in SDF2 one can (mis)use a non-transitive priority to remove a direct father/child relation from the grammar. In RASCAL we use a semantic action for this.

7.5.1 Evaluation Method

DR. AMBIGUITY is effective if it can explain the existence of a significant amount of choice nodes in parse forests and proposes the right fixes. We measure this effectiveness in terms

	Diagnoses								
Experiment	**P**	**A**	**R**	**F**	**c**	**v**	**O**	**Precision**	**Recall**
1. Remove priority between `"*"` and `"+"`	**1**	1	0	0	0	1	0	33%	100%
2. Remove associativity for `"+"`	0	**1**	0	0	0	0	0	100%	100%
3. Remove reservation of `true` keyword from `ID`	0	0	**1**	0	0	1	0	50%	100%
4. Remove longest match for identifiers	0	0	0	**6**	0	0	0	16%	100%
5. Remove package name vs. field access priority	0	0	0	0	**6**	1	0	14%	100%
6. Remove vertical preference for dangling else	0	0	0	1	14	**1**	1	7%	100%
7. *All the above changes at the same time*	**1**	**2**	**1**	**7**	**20**	**4**	1	17%	100%

Table 7.2: Precision/Recall results for each experiment, including (P)riority, (A)ssociativity, (R)eject, (F)ollow restrictions, A(c)tions filtering edges, A(v)oid/prefer suggestions, and (O)ffside rule. For each experiment, the figures of the removed disambiguation are highlighted.

of precision and recall. DR. AMBIGUITY has high precision if it does not propose too many solutions that are useless or meaningless to the language engineer. It has high recall if it finds all the solutions that the language engineer deems necessary. Our evaluation method is as follows:

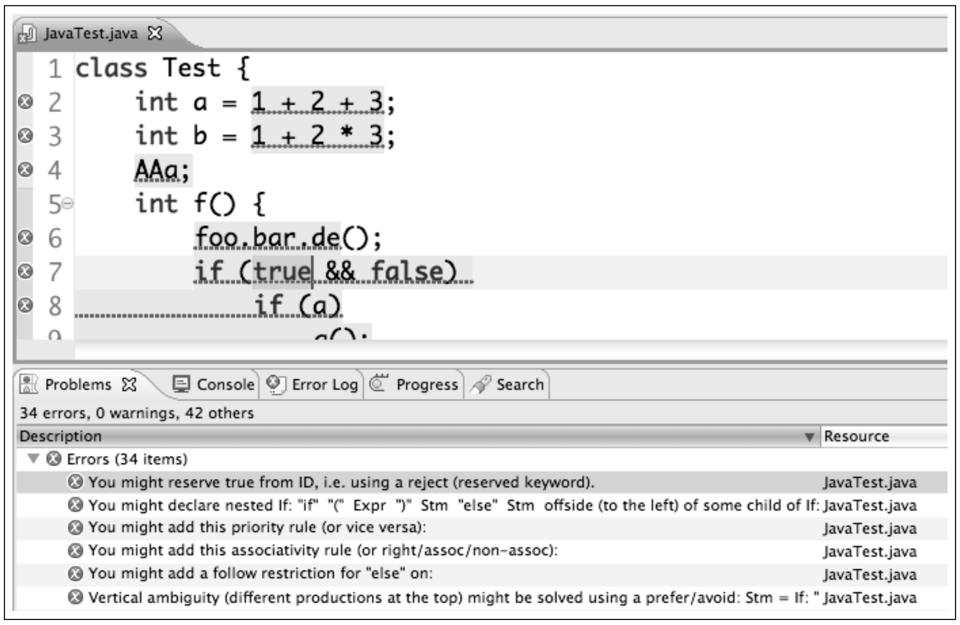
- The set of disambiguations that Bravenboer applied to his Java grammar is our "golden standard".
- The disambiguations in the grammar are selectively removed, which results in different ambiguous versions of the grammar. New parsers are generated for each version.
- An example Java program is parsed with each newly generated parser. The program is unambiguous for the original grammar, but becomes ambiguous for each altered version of the grammar.
- We measure the total amount and which kinds of suggestions are made by DR. AMBIGUITY for the parse forests of each grammar version, and compute the precision and recall.

Precision is computed by $\frac{|\text{FoundDisambiguations} \cap \text{RemovedDisambiguations}|}{|\text{FoundDisambiguations}|} \times 100\%$. We expect low precision, around 50%, because each particular ambiguity often has many different solution types. Low precision is not necessarily a bad thing, provided the total amount of disambiguation suggestions remains human-checkable.

Recall is computed by $\frac{|\text{FoundDisambiguations} \cap \text{RemovedDisambiguations}|}{|\text{RemovedDisambiguations}|} \times 100\%$. From this number we see how much we have missed. We expect the recall to be 100% in our experiments, since we designed our detection methods specifically for the disambiguation techniques of SDF2.

7.5.2 Results

Table 7.2 contains the results of measuring the precision and recall on a number of experiments. Each experiment corresponds to a removal of one or more disambiguation constructs and the



```
class Test {
    int a = 1 + 2 + 3;
    int b = 1 + 2 * 3;
    AAa;
    int f() {
        foo.bar.de();
        if (true && false)
            if (a)
```

34 errors, 0 warnings, 42 others

Description	Resource
Errors (34 items)	
You might reserve true from ID, i.e. using a reject (reserved keyword).	JavaTest.java
You might declare nested If: "if" "(" Expr ")" Stm "else" Stm offside (to the left) of some child of If:	JavaTest.java
You might add this priority rule (or vice versa):	JavaTest.java
You might add this associativity rule (or right/assoc/non-assoc):	JavaTest.java
You might add a follow restriction for "else" on:	JavaTest.java
Vertical ambiguity (different productions at the top) might be solved using a prefer/avoid: Stm = If: "	JavaTest.java

Figure 7.5: DR. AMBIGUITY reports diagnostics in the RASCAL language workbench.

parsing of a single Java program file that triggers the introduced ambiguity.

Table 7.2 shows that we indeed always find the removed disambiguation among the suggestions. Also, we always find more than one suggestion (the second experiment is the only exception).

The dangling-else ambiguity of experiment 6 introduces many small differences between two alternatives, which is why many (arbitrary) semantic actions are proposed to solve these. We may learn from this that semantic actions need to be presented to the language engineer as a last resort. For these disambiguations the risk of collateral damage (a non-language preserving disambiguation) is also quite high.

The final experiment tests whether the simultaneous analysis of different choice nodes that are present in a parse forest may lead to a loss of precision or recall. The results show that we find exactly the same suggestions. Also, as expected the precision of such an experiment is very low. Note however, that DR. AMBIGUITY reports each disambiguation suggestion per choice node, and thus the precision is usually perceived per choice node and never as an aggregated value over an entire source file. Figure 7.5 depicts how DR. AMBIGUITY may report its output.

7.5.3 Discussion

We have demonstrated the effectiveness of DR. AMBIGUITY for only one grammar. Moreover this grammar already contained disambiguations that we have removed, simultaneously creating a representative case and a golden standard.

We may question whether DR. AMBIGUITY would do well on grammars that have not been written with any disambiguation construct in mind. We may also question whether DR. AMBIGUITY works well on completely different grammars, such as for COBOL or PL/I. More experimental evaluation is warranted. Nevertheless, this initial evaluation based on Java looks promising and does not invalidate our approach.

Regarding the relatively low precision, we claimed that this is indeed wanted in many cases. The actual resolution of an ambiguity is a language design question. DR. AMBIGUITY should not a priori promote a particular disambiguation over another well known disambiguation. For example, reverse engineers have a general dislike of the offside rule because it complicates the construction of a parser, while the users of a domain specific language may applaud the sparing use of bracket literals.

7.6 Conclusions

We have presented theory and practice of automatically diagnosing the causes of ambiguity in context-free grammars for programming languages and of proposing disambiguation solutions. We have evaluated our prototype implementation on an actively used and mature grammar for Java 5, to show that DR. AMBIGUITY can indeed propose the proper disambiguations.

Future work on this subject includes further extension, further usability study and finally proofs of correctness. To support development of front-ends for many programming languages and domain specific languages, we will include DR. AMBIGUITY in releases of the RASCAL IDE (a software language workbench).

Chapter 8

Conclusions

"I left the ending ambiguous, because that is the way life is."
Bernardo Bertolucci

This thesis investigates ambiguity detection methods for context-free grammars. Its goal is to advance the usability of ambiguity detection to a level that is practical for checking grammars of real world programming languages. The main result of this thesis is a novel approach called AMBIDEXTER, *which detects production rules that do not contribute to ambiguity. By filtering these rules from a grammar, the runtime of further ambiguity detection can be reduced significantly. Furthermore, we present* DR. AMBIGUITY, *an expert system that automatically proposes possible cures for ambiguity. This section concludes the thesis by summarizing our contributions, and discussing future work.*

8.1 Contributions to Research Questions

In the introduction we posed a series of research questions that focus on improving the usability of ambiguity detection. The subsequent chapters each try to answer one or more of these questions. The following summarizes our contributions to each of the questions.

Research Question 1

How to assess the practical usability of ambiguity detection methods?

An ambiguity detection method (ADM) is practically usable on a given grammar if it can tell within a reasonable amount of time whether the grammar is ambiguous or not. Therefore, performance, accuracy and termination are very important usability criteria. Furthermore, a method becomes more usable if it can be run with various accuracy settings, such that its behaviour can be adjusted to the available time and memory. Finally, the usefulness of an ADM increases if, in case of ambiguity, it can indicate the causes of the ambiguity in the grammar, and advise on possible solutions. All these usability criteria are discussed in Chapter 2.

Research Question 2

What is the usability of the state-of-the-art in ambiguity detection?

Also in Chapter 2, we investigate the usability of three different ADMs. Their implementations are tested on two sets of benchmark grammars: one set with 84 ambiguous and unambiguous grammar snippets, and one set containing 25 ambiguous and unambiguous variants of 5 real programming language grammars. The methods under investigation are the sentence generator AMBER by Schröer [Sch01], the Noncanonical Unambiguity (NU) Test by Schmitz [Sch07b], and the LR(k) test by Knuth [Knu65]. Their scores on each of the above mentioned criteria are measured and analyzed, and the methods are compared to each other.

Two of the investigated ADMs are quite usable on our grammars. The approximative NU Test shows good accuracy, performance, and termination characteristics, but is only able to decide unambiguity. On the other hand, the exhaustive sentence generator AMBER can only detect the existence of ambiguity, with reasonable performance, but it will never terminate on unambiguous grammars.

Research Question 3

How to improve the performance of ambiguity detection?

In Chapter 3 we present AMBIDEXTER, a novel ambiguity detection approach that uses grammar filtering to speed up exhaustive searching. It uses an extension of the NU Test that enables the detection of *harmless production rules* in a grammar. These are the rules that certainly do not contribute to ambiguity. If all productions of a grammar are identified as harmless then the grammar is unambiguous. Otherwise, the harmless rules can safely be filtered to produce a smaller grammar that contains the same ambiguities as the original one. This filtered grammar can then be searched with an exhaustive technique in less time, because of its smaller language.

The effectiveness of this approach is tested on the same set of programming language grammars that was used in Chapter 2. The results show that the filtering of harmless rules from these grammars significantly reduces sentence generation times, sometimes with several orders of magnitude.

In Chapter 5 we present a series of extensions to our grammar filtering approach, to make it suitable for filtering character-level grammars. These grammars include the full lexical definitions of their languages, and are therefore more ambiguity-prone. We present extensions for including disambiguation filters in the test, as well as a grammar unfolding technique to deal with the increased complexity of character-level grammars.

Again, an implementation of the extensions is evaluated on a series of real-world grammars. Although the reported gains in sentence generation time are not as large as for token-based grammars, our technique proves to be very useful on all but one grammar.

Research Question 4

How to improve the accuracy of approximative ambiguity detection?

In Chapter 4 we present the theoretical foundations for our grammar filtering technique and prove its correctness. We show how to extend both the Regular Unambiguity (RU) Test and the more accurate Noncanonical Unambiguity (NU) Test to find harmless production rules. With the RU Test our approach is able to find production rules that can only be used to derive unambiguous strings. With the NU Test it can also find productions that can only be used to derive unambiguous substrings of ambiguous strings. We also show that the number of detected harmless rules can be increased if the filtering is applied in an iterative fashion. This shows that grammar filtering has an accuracy increasing effect on the approximative RU Test and the NU Test.

Furthermore, the character-level extensions presented in Chapter 5 also increase the accuracy of our grammar filtering technique and the RU Test and NU Test in general. By taking disambiguation filters into account, the tests can ignore ambiguities that are already solved by the grammar developer. Furthermore, the general grammar unfolding technique increases accuracy by taming the approximation in relevant areas of the grammar.

Research Question 5

How to improve the usefulness of ambiguity detection reports?

Our contribution to this question is twofold. First, the harmless production rules produced by our grammar filtering approach serve as useful ambiguity reports, because they can give confidence about the unambiguity of certain parts of a grammar. Second, in Chapter 7 we present an expert system called DR. AMBIGUITY, that can automatically propose applicable cures for ambiguous sentences that are for instance found by a sentence generator.

The chapter gives an overview of different types of ambiguity and ambiguity resolutions, and shows how they are linked together by DR. AMBIGUITY. The usefulness of the expert system is evaluated by applying it on a mature character-level grammar for Java. We remove several disambiguation filters from the grammar and test whether DR. AMBIGUITY is able to detect them as a possible solution. Initial results show that in all cases the removed solution was among the proposed cures.

8.2 Discussion

Despite the performance and accuracy characteristics on an ADM, its practical usability on a certain grammar also depends on the shape of the grammar, and available resources like time, memory and computing power. It is therefore very hard to determine whether the current state of the art is usable enough for checking real programming language grammars. Furthermore, because the ambiguity detection problem is undecidable, there are grammars for which one can never find a satisfying answer.

Nevertheless, as the results presented throughout this thesis show, grammar filtering is a general technique that can have very beneficial effects in many situations, independent of the checked grammar, available hardware or applied detection method. Filtering harmless production rules from a grammar can significantly reduce the runtime of exhaustive ADMs, as well as improve the accuracy of approximative ADMs. Furthermore, detected harmless production rules serve as a useful ambiguity report that guarantee the unambiguity of part of a grammar.

Our experiments also show that most grammars can already be filtered in an efficient manner on present day hardware. Also, a small investment in filtering time leads to the earlier detection of ambiguities in existing programming language grammars. Therefore, grammar filtering is ready to be included in language workbenches, preferably in combination with an exhaustive sentence generator and an expert system like DR. AMBIGUITY.

8.3 Future Work

An advantage of the undecidability of the ambiguity problem is that there will always be room for improvement. We envisage that research into improving the performance, accuracy and reporting of ambiguity detection will always be relevant. To encourage further investigations, we list a number of possible research directions.

ADM Comparison The evaluation of ADMs in Chapter 2 only includes the implementations of three methods. It would be interesting to also compare the methods of Brabrand, Giegerich and Møller [BGM10], Chueng and Uzgalis [CU95], CFG ANALYZER [AHL08], and our own sentence generator described in Chapter 6.

Approximative Testing Although grammar approximation has already been studied quite substantially [MN01, Sch07a, BGM10, YJ03], we hope it is still possible to find better approximations that are suitable for ambiguity detection of programming language grammars. We see the following challenges and possible directions:

- Since we target grammars of programming languages, we could exploiting their typical characteristics. Very often, most syntactic structures of a programming language are regular, and context-free embedding is only used for scoping and expression nesting. If the regular structures can be separated from the context-free ones, only the latter have to be approximated.
- Furthermore, the effectiveness of various approximation improvements can vary a lot per grammar. It would be helpful if certain properties in a grammar can be found that indicate which type of approximation will result in suitable accuracy and performance.
- Another challenge for approximation techniques is to ignore known or intended [vdBKMV03] ambiguities in a grammar. This way, they can provide a guarantee that no other unknown ambiguities exist.

- In Section 4.7 we discussed refining the approximation precision with every iteration of filtering a grammar. Like in model checking [CGJ$^+$00], we envisage this can be done in an automatic way. More research is needed to investigate this approach in the setting of ambiguity detection.
- From process theory it is known that bisimulation is decidable for processes described by context-free grammars [HM95]. Bisimulation equivalence of grammars also implies language equivalence, but it is strictly weaker. Since language equivalence is closely related to the ambiguity problem, these results might be extended into an approximative ambiguity test.

Harmless Production Rules The technique for detecting harmless production rules as described in Chapter 4 is able to detect production rules that can only derive unambiguous sentences or subsentences. However, there can also be productions that can derive ambiguous sentences while they are unambiguous themselves. These productions then only appear in parse trees *above* ambiguity nodes, and do therefore not directly contribute to ambiguity. We expect these productions can be found using the maEq^* relations described in Section 4.4.2. More research is needed to work out this idea.

Sentence Generation Sentence generation could possibly be sped up enormously by sharing the many common substrings that exist in the language of a grammar. This is presumably easier to achieve with breadth first searching. The results of Chapter 3 show that the breadth first sentence generator CFG ANALYZER [AHL08], which uses an incremental SAT-solver for its searching, performs very well. Our guess is that by using more domain knowledge about grammars in the searching, this SAT-based approach might be beaten.

Reporting Initial experiments show that DR. AMBIGUITY is able of reporting all possible cures for an ambiguous sentence. However, which one to apply is a language design question, and also depends on its effects on other parse trees. More research is needed to help a grammar developer to make an informed choice from the — possibly long — list of available solutions.

Bibliography

[AH99] J. Aycock and R. N. Horspool. Faster generalized LR parsing. In S. Jähnichen, editor, *Proceedings of the Eigth International Conference on Compiler Construction (CC 1999)*, volume 1575 of *LNCS*, pages 32–46. Springer-Verlag, 1999. Cited on page 105.

[AH01] J. Aycock and R. N. Horspool. Schrödinger's token. *Software: Practice & Experience*, 31(8):803–814, 2001. Cited on page 70.

[AHL08] R. Axelsson, K. Heljanko, and M. Lange. Analyzing context-free grammars using an incremental SAT solver. In *Proceedings of the 35th International Colloquium on Automata, Languages, and Programming (ICALP 2008)*, volume 5126 of *LNCS*, 2008. Cited on pages 14, 28, 32, 91, 96, 128, and 129.

[Ake78] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, 1978. Cited on page 98.

[AL90] T. Altman and G. Logothetis. A note on ambiguity in context-free grammars. *Information Processing Letters*, 35(3):111–114, 1990. Cited on pages 52 and 113.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. Cited on pages 110, 111, and 114.

[Bas07] H. J. S. Basten. Ambiguity detection methods for context-free grammars. Master's thesis, Universiteit van Amsterdam, August 2007. Cited on pages 23 and 38.

[Bas09] H. J. S. Basten. The usability of ambiguity detection methods for context-free grammars. In A. Johnstone and J. Vinju, editors, *Proceedings of the Eigth Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, volume 238 of *ENTCS*, 2009. Cited on pages 19 and 21.

[Bas10] H. J. S. Basten. Tracking down the origins of ambiguity in context-free grammars. In *Proceedings of the Seventh International Colloquium on Theoretical Aspects of Computing (ICTAC 2010)*, volume 6255 of *LNCS*, pages 76 – 90. Springer, 2010. Cited on pages 19 and 51.

[BBV07] E. Bouwers, M. Bravenboer, and E. Visser. Grammar engineering support for precedence rule recovery and compatibility checking. In A. Sloane and A. Johnstone, editors, *Proceedings of the Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA 2007)*, pages 82–96, Braga, Portugal, March 2007. Cited on pages 28 and 110.

[BG06] A. Begel and S. L. Graham. XGLR–an algorithm for ambiguity in programming languages. *Science of Computer Programming*, 61(3):211 – 227, 2006. Special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA 2004). Cited on page 105.

[BGM10] C. Brabrand, R. Giegerich, and A. Møller. Analyzing ambiguity of context-free grammars. *Science of Computer Programming*, 75(3):176–191, 2010. Cited on pages 14, 22, 28, 32, 52, 86, 96, 113, and 128.

[BKV11] H. J. S. Basten, P. Klint, and J. J. Vinju. Ambiguity detection : scaling to scannerless. In J. Saraiva, U. Assmann, and A. Sloane, editors, *Proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011)*, LNCS. Springer, 2011. Cited on pages 19 and 69.

[BTV06] M. Bravenboer, É. Tanter, and E. Visser. Declarative, formal, and extensible syntax definition for AspectJ. *SIGPLAN Notices*, 41:209–228, October 2006. Cited on pages 106 and 120.

[BV04] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In J. M. Vlissides and D. C. Schmidt, editors, *OOPSLA*, pages 365–383. ACM, 2004. Cited on page 70.

[BV10] H. J. S. Basten and J. J. Vinju. Faster ambiguity detection by grammar filtering. In C. Brabrand and P. E. Moreau, editors, *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications (LDTA 2010)*. ACM, 2010. Cited on pages 19, 31, and 41.

[BV11] H. J. S. Basten and J. J. Vinju. Parse forest diagnostics with Dr. Ambiguity. In J. Saraiva, U. Assmann, and A. Sloane, editors, *Proceedings of the Fourth International Conference on Software Language Engineering (SLE 2011)*, LNCS. Springer, 2011. Cited on pages 19 and 105.

[BVdGD11] M. Bravenboer, R. Vermaas, R. de Groot, and E. Dolstra. Java-front: Java syntax definition, parser, and pretty-printer. Technical report, www.program-transformation.org, 2011. `http://www.program-transformation.org/Stratego/JavaFront`. Cited on pages 119 and 120.

[BvdS10] H. J. S. Basten and T. van der Storm. AmbiDexter: Practical ambiguity detection, tool demonstration. In *Proceedings of the Tenth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2010)*, pages 101 –102. IEEE, September 2010. Cited on pages 19 and 93.

[BVVV05] M. Bravenboer, R. Vermaas, J. J. Vinju, and E. Visser. Generalized type-based disambiguation of meta programs with concrete object syntax. In R. Glück and M. R. Lowry, editors, *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *LNCS*, pages 157–172, Tallinn, Estonia, 2005. Springer. Cited on page 111.

[Can62] D. G. Cantor. On the ambiguity problem of Backus systems. *Journal of the ACM*, 9(4):477–479, 1962. Cited on pages 13, 22, 52, and 106.

[CFJ$^+$09] P. Charles, R. M. Fuhrer, S. M. Sutton Jr., E. Duesterwald, and J. J. Vinju. Accelerating the creation of customized, language-specific IDEs in Eclipse. In S. Arora and G. T. Leavens, editors, *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*, 2009. Cited on page 116.

[CGJ$^+$00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000. Cited on pages 66 and 129.

[Cho56] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956. Cited on page 11.

[CS63] N. Chomsky and M. P. Schützenberger. The algebraic theory of context-free languages. In P. Braffort, editor, *Computer Programming and Formal Systems*, pages 118–161. North-Holland, Amsterdam, 1963. Cited on pages 13, 22, 52, and 106.

[CU95] B. S. N. Cheung and R. C. Uzgalis. Ambiguity in context-free grammars. In *Proceedings of the 1995 ACM Symposium on Applied Computing (SAC 1995)*, pages 272–276, New York, NY, USA, 1995. ACM Press. Cited on pages 14, 32, and 128.

[DeR69] F. L. DeRemer. *Practical translators for LR(k) languages*. PhD thesis, Dep. Electrical Engineering, Massachusetts Institute of Technology, Cambridge, 1969. Cited on page 15.

[DS05] C. Donnely and R. Stallman. *Bison version 2.5*, May 2005. `http://www.gnu.org/software/bison/manual/`. Cited on page 94.

[Ear70] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970. Cited on pages 15 and 105.

[Eco06] G. R. Economopoulos. *Generalised LR parsing algorithms*. PhD thesis, Royal Holloway, University of London, August 2006. Cited on page 105.

[Flo62] R. W. Floyd. On ambiguity in phrase structure languages. *Communications of the ACM*, 5(10):526–534, 1962. Cited on pages 13, 22, 52, and 106.

[For04] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. *SIGPLAN Notices*, 39:111–122, January 2004. Cited on pages 15 and 106.

[GH67] S. Ginsburg and M. A. Harrison. Bracketed context-free languages. *Journal of Computer and System Sciences*, 1(1):1–23, 1967. Cited on pages 54, 73, and 108.

[GJ08] D. Grune and C. J. H. Jacobs. *Parsing techniques a practical guide – Second edition*. Springer, 2008. Cited on pages 25 and 26.

[Gor63] S. 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`. Cited on pages 14 and 32.

[GrC] C grammar, `ftp://ftp.iecc.com/pub/file/c-grammar.gz`. Cited on page 38.

[GrJ] Java grammar, GCC, `http://gcc.gnu.org/cgi-bin/cvsweb.cgi/gcc/gcc/java/parse.y?rev=1.475`. Cited on page 38.

[GrP] Pascal grammar, `ftp://ftp.iecc.com/pub/file/pascal-grammar`. Cited on page 38.

[GrS] SQL grammar, GRASS, `grass-6.2.0RC1/lib/db/sqlp/yac.y` from `http://grass.itc.it/grass62/source/grass-6.2.0RC1.tar.gz`. Cited on page 38.

[HHKR89] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989. Cited on pages 16, 74, 87, 94, 109, and 110.

[HM95] Y. Hirshfeld and F. Moller. Decidability results in automata and process theory. In F. Moller and G. M. Birtwistle, editors, *Banff Higher Order Workshop*, volume 1043 of *LNCS*, pages 102–148. Springer, 1995. Cited on page 129.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979. Cited on page 53.

[Hut92] G. Hutton. Higher-order functions for parsing. *Journal of Functional Programming*, 2(3):323–343, 1992. Cited on page 15.

[Jam05] S. 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/`. Cited on pages 14 and 28.

[Joh] S. C. Johnson. *Yacc: Yet Another Compiler-Compiler*. AT&T Bell Laboratories. `http://dinosaur.compilertools.net/yacc/`. Cited on page 94.

[JS11] A. Johnstone and E. Scott. Modelling GLL parser implementations. In B. Malloy, S. Staab, and M. van den Brand, editors, *Proceedings of the Third International Conference on Software Language Engineering (2010)*, volume 6563 of *LNCS*, pages 42–61. Springer Berlin / Heidelberg, 2011. Cited on page 105.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972. Cited on page 101.

[KLV05] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14:331–380, July 2005. Cited on page 12.

[Knu65] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965. Cited on pages 15, 18, 22, 25, 34, 53, 57, and 126.

[Knu71] D. E. Knuth. Top-down syntax analysis. *Acta Informatica*, 1:79–110, 1971. Cited on page 15.

[KV94] P. Klint and E. Visser. Using filters for the disambiguation of context-free grammars. In *Proceedings of the ASMICS Workshop on Parsing Theory*, Technical Report 126-1994, pages 1–20. Università di Milano, 1994. Cited on pages 16, 73, and 109.

[KvdSV11] P. Klint, T. van der Storm, and J. J. Vinju. EASY meta-programming with Rascal. In J. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *LNCS*, pages 222–289. Springer Berlin / Heidelberg, 2011. Cited on pages 16, 94, 110, 116, and 117.

[KVW10] L. C. L. Kats, E. Visser, and G. Wachsmuth. Pure and declarative syntax definition: paradise lost and regained. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *OOPSLA*, pages 918–932. ACM, 2010. Cited on page 70.

[Lan66] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9:157–166, March 1966. Cited on page 110.

[Lan74] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In *Proceedings of the Second Colloquium on Automata, Languages, and Programming (ICALP 1974)*, volume 14 of *LNCS*, pages 255–269. Springer, 1974. Cited on page 106.

[LV01] R. Lämmel and C. Verhoef. Semi-automatic grammar recovery. *Software: Practice & Experience*, 31:1395–1448, December 2001. Cited on page 106.

[Mak95] V. Makarov. *MSTA (syntax description translator)*, May 1995. See `http://cocom.sourceforge.net/msta.html`. Cited on pages 22 and 25.

[MN01] M. Mohri and M. J. Nederhof. Regular approximations of context-free grammars through transformation. In J. C. Junqua and G. van Noord, editors, *Robustness in Language and Speech Technology*, chapter 9, pages 153–163. Kluwer Academic Publishers, 2001. Cited on page 128.

[Moo01] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE 2001)*, pages 13–, Washington, DC, USA, 2001. IEEE Computer Society. Cited on page 111.

[NF91] R. Nozohoor-Farshi. GLR parsing for epsilon-grammars. In *Generalized LR Parsing*, pages 60 – 75. Kluwer Academic Publishers, The Netherlands, 1991. Cited on page 15.

[PF11] T. Parr and K. Fisher. LL(*): the foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI 2011)*, pages 425–436, New York, NY, USA, 2011. ACM. Cited on pages 15 and 106.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992. Cited on page 109.

[SC89] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Notices*, 24:170–178, June 1989. Cited on pages 12 and 110.

[Sch01] F. W. Schröer. AMBER, an ambiguity checker for context-free grammars. Technical report, compilertools.net, 2001. See `http://accent.compilertools.net/Amber.html`. Cited on pages 14, 18, 22, 32, 71, 91, 96, 113, and 126.

[Sch06] F. W. Schröer. ACCENT, a compiler compiler for the entire class of context-free grammars, second edition. Technical report, compilertools.net, 2006. See `http://accent.compilertools.net/Accent.html`. Cited on pages 96 and 105.

[Sch07a] S. Schmitz. *Approximating Context-Free Grammars for Parsing and Verification*. PhD thesis, Université de Nice - Sophia Antipolis, 2007. Cited on pages 28, 52, 77, and 128.

[Sch07b] S. Schmitz. Conservative ambiguity detection in context-free grammars. In L. Arge, C. Cachin, T. Jurdziński, and A. Tarlecki, editors, *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP 2007)*, volume 4596 of *LNCS*, 2007. Cited on pages 14, 18, 22, 26, 28, 32, 34, 52, 56, 57, 62, 63, 71, and 126.

[Sch10] S. Schmitz. An experimental ambiguity detection tool. *Science of Computer Programming*, 75(1-2):71–84, 2010. Cited on pages 22, 26, 40, 46, and 52.

[SJ10] E. Scott and A. Johnstone. GLL parsing. In *Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009)*, volume 253, pages 177 – 189, 2010. Cited on pages 15 and 105.

[SSS88] S. Sippu and E. Soisalon-Soininen. *Parsing theory. Vol. 1: languages and parsing*. Springer-Verlag New York, Inc., New York, NY, USA, 1988. Cited on pages 53 and 61.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. Cited on page 77.

[Tom85] M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985. Cited on pages 15 and 105.

[vdBKMV03] M. van den Brand, S. Klusener, L. Moonen, and J. J. Vinju. Generalized parsing and term rewriting: Semantics driven disambiguation. *ENTCS*, 82(3):575–591, 2003. Cited on page 128.

[vdBSVV02] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In *Proceedings of the 11th International Conference on Compiler Construction (CC 2002)*, pages 143–158, London, UK, 2002. Springer-Verlag. Cited on pages 12, 69, 70, 73, 80, 91, 98, 105, 109, 110, 111, and 114.

[Vin11] J. J. Vinju. SDF disambiguation medkit for programming languages. Technical Report SEN-1107, Centrum Wiskunde & Informatica, 2011. Cited on page 108.

[Vis97] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997. Cited on pages 16, 74, 87, 94, 98, 100, 110, and 117.

[VS07] E. Van Wyk and A. Schwerdfeger. Context-aware scanning for parsing extensible languages. In C. Consel and J. L. Lawall, editors, *Proceedings of the Sixth International Conference on Generative Programming and Component Engineering (GPCE 2007)*, pages 63–72. ACM, 2007. Cited on page 70.

[YJ03] A. Yli-Jyrä. Regular approximations through labeled bracketing. In G. Jäger, P. Monachesi, G. Penn, and S. Wintner, editors, *Proceedings of the 8th conference on Formal Grammar 2003 "FG Vienna"*, pages 189–201, 2003. Cited on page 128.

Summary

Context-free grammars are the most suitable and most widely used method for describing the syntax of programming languages. They can be used to generate parsers, which transform a piece of source code into a tree-shaped representation of the code's syntactic structure. These parse trees can then be used for further processing or analysis of the source text. In this sense, grammars form the basis of many engineering and reverse engineering applications, like compilers, interpreters and tools for software analysis and transformation. Unfortunately, context-free grammars have the undesirable property that they can be ambiguous, which can seriously hamper their applicability.

A grammar is ambiguous if at least one sentence in its language has more than one valid parse tree. Since the parse tree of a sentence is often used to infer its semantics, an ambiguous sentence can have multiple meanings. For programming languages this is almost always unintended. Ambiguity can therefore be seen as a grammar bug.

A specific category of context-free grammars that is particularly sensitive to ambiguity are character-level grammars, which are used to generate scannerless parsers. Unlike traditional token-based grammars, character-level grammars include the full lexical definition of their language. This has the advantage that a language can be specified in a single formalism, and that no separate lexer or scanner phase is necessary in the parser. However, the absence of a scanner does require some additional lexical disambiguation. Character-level grammars can therefore be annotated with special disambiguation declarations to specify which parse trees to discard in case of ambiguity. Unfortunately, it is very hard to determine whether all ambiguities have been covered.

The task of searching for ambiguities in a grammar is very complex and time consuming, and is therefore best automated. Since the invention of context-free grammars, several ambiguity detection methods have been developed to this end. However, the ambiguity problem for context-free grammars is undecidable in general, so the perfect detection method cannot exist. This implies a trade-off between accuracy and termination. Methods that apply exhaustive searching are able to correctly find all ambiguities, but they might never terminate. On the other hand, approximative search techniques do always produce an ambiguity report, but these might contain false positives or false negatives. Nevertheless, the fact that every method has flaws does not mean that ambiguity detection cannot be useful in practice.

This thesis investigates ambiguity detection with the aim of checking grammars for programming languages. The challenge is to improve upon the state-of-the-art, by finding accurate enough methods that scale to realistic grammars. First we evaluate existing methods with a set of criteria for practical usability. Then we present various improvements to ambiguity detection in the areas of accuracy, performance and report quality.

The main contributions of this thesis are two novel techniques. The first is an ambiguity detection method that applies both exhaustive and approximative searching, called AMBIDEXTER. The key ingredient of AMBIDEXTER is a grammar filtering technique that can remove harmless production rules from a grammar. A production rule is harmless if it does not contribute to any ambiguity in the grammar. Any found harmless rules can therefore safely be removed. This results in a smaller grammar that still contains the same ambiguities as the original one. However, it can now be searched with exhaustive techniques in less time.

The grammar filtering technique is formally proven correct, and experimentally validated. A prototype implementation is applied to a series of programming language grammars, and the performance of exhaustive detection methods are measured before and after filtering. The results show that a small investment in filtering time can substantially reduce the run-time of exhaustive searching, sometimes with several orders of magnitude.

After this evaluation on token-based grammars, the grammar filtering technique is extended for use with character-level grammars. The extensions deal with the increased complexity of these grammars, as well as their disambiguation declarations. This enables the detection of productions that are harmless due to disambiguation. The extentions are experimentally validated on another set of programming language grammars from practice, with similar results as before. Measurements show that, even though character-level grammars are more expensive to filter, the investment is still very worthwhile. Exhaustive search times were again reduced substantially.

The second main contribution of this thesis is DR. AMBIGUITY, an expert system to help grammar developers to understand and solve found ambiguities. If applied to an ambiguous sentence, DR. AMBIGUITY analyzes the causes of the ambiguity and proposes a number of applicable solutions. A prototype implementation is presented and evaluated with a mature Java grammar. After removing disambiguation declarations from the grammar we analyze sentences that have become ambiguous by this removal. The results show that in all cases the removed filter is proposed by DR. AMBIGUITY as a possible cure for the ambiguity.

Concluding, this thesis improves ambiguity detection with two novel methods. The first is the ambiguity detection method AMBIDEXTER that applies grammar filtering to substantially speed up exhaustive searching. The second is the expert system DR. AMBIGUITY that automatically analyzes found ambiguities and proposes applicable cures. The results obtained with both methods show that automatic ambiguity detection is now ready for realistic programming language grammars.

Samenvatting

Context-vrije grammatica's zijn de methode bij uitstek voor het beschrijven van de syntax van programmeertalen. Ze worden gebruikt voor het genereren van parsers. Met een parser kan de syntactische structuur van een stuk broncode worden ontleed. Het resultaat is een boomstructuur waarmee verdere verwerking of analyse van de broncode mogelijk is. Op deze manier vormen grammatica's de basis voor veel (domein specifieke) taalimplementaties, zoals compilers en interpreters, maar ook voor *reverse engineering* toepassingen zoals broncodeanalyse en -transformatie. Helaas hebben context-vrije grammatica's de ongewenste eigenschap dat ze ambigu kunnen zijn.

Een grammatica is ambigu als één of meerdere zinnen of programma's in haar taal op verschillende manieren ontleed kunnen worden. De syntactische ontleding is vaak het uitgangspunt voor semantische analyse, dus een ambigue zin kan meerdere betekenissen hebben. Omdat programmeertalen juist zijn bedoeld om ondubbelzinnig programma's in te schrijven, kan ambiguïteit worden gezien als een grammatica-bug.

Een specifieke klasse van context-vrije grammatica's die zeer gevoelig zijn voor ambiguïteit zijn *character-level* grammatica's. Deze grammatica's zijn bedoeld om zogenaamde scannerless parsers mee te genereren. In tegenstelling tot traditionele *token-based* grammatica's specificeren character-level grammatica's ook de lexicale syntax van hun taal. Dit heeft als voordelen dat een taal in één formalisme gedefinieerd kan worden en dat de parser geen aparte lexer of scanner nodig heeft. Desalniettemin, door de afwezigheid van een scanner is er wel additionele lexicale disambiguatie nodig. Character-level grammatica's kunnen daarom worden geannoteerd met speciale disambiguatiedeclaraties, om aan te geven welke alternatieve ontledingen van een ambigue zin genegeerd kunnen worden. Helaas is het erg moeilijk om vast te stellen of alle ambiguïteiten zijn afgedekt.

Het zoeken naar ambiguïteiten in een grammatica is erg complex en tijdrovend. Daarom kan deze taak het beste worden geautomatiseerd. Sinds de ontdekking van context-vrije grammatica's zijn hiervoor verschillende ambiguïteitsdetectiemethoden ontwikkeld. Echter, de perfecte detectiemethode zal nooit kunnen bestaan omdat het ambiguïteitsprobleem voor context-vrije grammatica's in het algemeen onbeslisbaar is. Hierdoor is er altijd een afweging tussen accuraatheid en terminatie. Methoden die een uitputtende zoekstrategie toepassen kunnen in principe alle ambiguïteiten vinden, maar dit kan oneindig lang duren. Aan de andere kant zijn er methoden die bij benadering zoeken om wel altijd te kunnen termineren, maar hun resultaten kunnen hierdoor niet altijd correct zijn. Het feit dat elke methode onvolkomenheden heeft hoeft de praktische bruikbaarheid echter niet in de weg te staan.

Dit proefschrift behandelt ambiguïteitsdetectie gericht op grammatica's van programmeertalen. Hierbij is het de uitdaging om de state-of-the-art te verbeteren door het vinden van methoden die accuraat genoeg zijn en tegelijkertijd schalen naar realistische grammatica's. Als uitgangspunt voor deze zoektocht evalueren we eerst bestaande methoden aan de hand van bruikbaarheidscriteria's. Vervolgens verbeteren we de accuraatheid, prestatie en rapportage-kwaliteit van ambiguïteitsdetectie.

De belangrijkste bijdragen van dit proefschrift zijn twee nieuwe technieken. De eerste is een detectiemethode genaamd AMBIDEXTER, die een uitputtende zoekstrategie combineert met zoeken bij benadering. Het hoofdbestanddeel van AMBIDEXTER is een filtertechniek om onschadelijke productieregels uit een grammatica te verwijderen. Onschadelijke regels dragen op geen enkele wijze bij aan ambiguïteit en kunnen daarom veilig uit de grammatica verwijderd worden. Na het filteren blijft er een kleinere grammatica over die dezelfde ambiguïteiten bevat als de originele grammatica. Het voordeel hiervan is dat de kleinere grammatica nu sneller doorzocht kan worden met uitputtende zoektechnieken.

De grammaticafiltering is formeel correct bewezen en experimenteel gevalideerd. Met een prototype implementatie zijn een aantal bestaande grammatica's van programmeertalen gefilterd. Daarna is de prestatie van uitputtende zoekmethoden gemeten op de originele en de gefilterde grammatica's. De uitkomst is dat een kleine investering in filtertijd de zoektijd van deze methoden substantieel kan verminderen, soms zelfs met meerdere ordes van grootte.

Na deze evaluatie op token-based grammatica's passen we de filtertechniek aan voor character-level grammatica's. We presenteren een aantal uitbreidingen om de additionele complexiteit van deze grammatica's het hoofd te bieden en om rekening te houden met disambiguatiedeclaraties. Hierdoor kunnen er ook regels gefilterd worden die onschadelijk zijn gemaakt door disambiguatie. De uitbreidingen zijn wederom experimenteel gevalideerd op een serie programmeertaalgrammatica's, met vergelijkbare resultaten. Ondanks het feit dat character-level grammatica's duurder zijn om te filteren, blijft de investering de moeite waard. Ook hier nemen de zoektijden van uitputtende methoden substantieel af.

De tweede hoofdbijdrage van dit proefschrift is DR. AMBIGUITY, een expertsysteem om grammatica-ontwikkelaars te helpen met het begrijpen en oplossen van gevonden ambiguïteiten. Hiermee kan een ambigue zin worden geanalyseerd om de oorzaken van de ambiguïteit in de grammatica te vinden. Vervolgens worden er een aantal toepasbare oplossingen voor de ambiguïteit voorgesteld. We presenteren een prototype implementatie en evalueren deze op een grammatica van Java. Door disambiguatiedeclaraties te verwijderen genereren we ambigue zinnen, die we vervolgens analyseren met DR. AMBIGUITY. In alle gevallen worden de verwijderde declaraties voorgesteld als mogelijke oplossing voor de geanalyseerde ambiguïteit.

Samenvattend, dit proefschrift verbetert ambiguïteitsdetectie met twee nieuwe methoden. AMBIDEXTER past grammaticafiltering toe om de zoektijd van uitputtende methoden substantieel te verkleinen. Het expertsysteem DR. AMBIGUITY kan gevonden ambiguïteiten analyseren en mogelijke oplossingen voorstellen. De resultaten van beide methoden laten zien dat ambiguiteitsdetectie klaar is voor realistische grammatica's van programmeertalen.

Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multidisciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

You picked the wrong parse tree…

Bas Basten

Detection for Programming Language Grammars

Ambiguity

Detection for Programming Language Grammars

Bas Basten