Ambiguity detection for programming language grammars
Basten, H.J.S.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
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
3.2. Filtering Unambiguous Productions

**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 -> type method_declarator throws .)
- (method_header -> type method_declarator throws . ,
  method_header -> VOID_TK method_declarator throws .)
- (method_header -> modifiers VOID_TK method_declarator throws .)
- (method_header -> modifiers VOID_TK 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 \(\Rightarrow\) denotes derivation. We say \(\alpha B \gamma\) directly derives \(\alpha \beta \gamma\), written as \(\alpha B \gamma \Rightarrow \alpha \beta \gamma\) if a production rule \(B \rightarrow \beta\) exists in \(P\).
The symbol $\Rightarrow^*$ means “derives in zero or more steps”. An *item* indicates a position in a production rule using a dot, for instance as $S \rightarrow A\cdot BC$.

### 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 “$\text{item}_0$” relation [Sch07b]. We use $\text{item}_0$ here to explain the NU TEST for simplicity’s sake, ignoring the intricacies of other equivalence relations.

The $\text{item}_0$ 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 $A \rightarrow \alpha \cdot X \beta \xrightarrow{X} A \rightarrow \alpha X \cdot \beta$

- **Derivation transitions**, of form $A \rightarrow \alpha \cdot B \gamma \xrightarrow{i} B \rightarrow \cdot \beta$, where $i$ is the number of the production $B \rightarrow \beta$.

- **Reduction transitions**, of form $B \rightarrow \beta \cdot \xrightarrow{\gamma_i} A \rightarrow \alpha B \cdot \gamma$, where $i$ is the number of the production $B \rightarrow \beta$.

The derivation and shift transitions are similar to those in an LR(0) automaton, but the reductions are different. The $\text{item}_0$ 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 (•) transition from $A \rightarrow \alpha \bullet B \gamma$ does not necessarily match an \( \rightarrow \) transition to
$A \rightarrow \alpha B \bullet \gamma$. 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:
Chapter 3. Faster Ambiguity Detection by Grammar Filtering

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 \(A \rightarrow \alpha B \gamma\) to \(B \rightarrow \beta\), the matching \(\rangle_i\) transition from \(B \rightarrow \beta\) to \(A \rightarrow \alpha B \gamma\) 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
3.2. Filtering Unambiguous Productions

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. The set of defined non-terminals of $P_h$: $N_{\text{def}} = \{ A | A \rightarrow \alpha \in P_h \}$.

2. The used but undefined non-terminals of $P_h$: $N_{\text{undef}} = \{ B | A \rightarrow \alpha B \beta \in P_h \} \setminus N_{\text{def}}$.

3. The unproductive non-terminals: $N_{\text{unpr}} = \{ A | A \in N_{\text{def}} \setminus \{ A \rightarrow \beta a \gamma \} \in P'$.

4. New terminals $t_A$ for each non-terminal $A \in N_{\text{undef}} \cup N_{\text{unpr}}$.

5. Productions to complete the unproductive and undefined non-terminals: $P' = P_h \cup \{ A \rightarrow (t_A)^k | A \in N_{\text{undef}} \cup N_{\text{unpr}}, k = \text{minlength}(A) \}$.

6. The new set of terminal symbols: $T' = \{ a | (A \rightarrow \beta a \gamma) \in P' \}$.

7. Finally, the new grammar: $G' = (N_{\text{def}} \cup N_{\text{undef}}, T', P', S')$.

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.

\footnote{Where \text{minlength}(A) = \min(\{ k | \exists u, A \Rightarrow^* 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.
3.3. Experimental Validation

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\textsuperscript{5} and CFG ANALYZER version 03/12/2007\textsuperscript{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 \texttt{time} utility and measured memory usage by polling a process with \texttt{pid} every 0.1 seconds.

\textsuperscript{5}downloaded from http://accent.compilertools.net/
\textsuperscript{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).
### 3.3. Experimental Validation

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Rules filtered</th>
<th>Time</th>
<th>Memory (Mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LR0</td>
<td>SLR1</td>
<td>LALR1</td>
</tr>
<tr>
<td>SQL.0</td>
<td>79</td>
<td>79</td>
<td>79</td>
</tr>
<tr>
<td>SQL.1</td>
<td>79</td>
<td>65</td>
<td>65</td>
</tr>
<tr>
<td>SQL.2</td>
<td>80</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>SQL.3</td>
<td>80</td>
<td>54</td>
<td>54</td>
</tr>
<tr>
<td>SQL.4</td>
<td>80</td>
<td>71</td>
<td>71</td>
</tr>
<tr>
<td>SQL.5</td>
<td>80</td>
<td>68</td>
<td>68</td>
</tr>
<tr>
<td>Pascal.0</td>
<td>176</td>
<td>21</td>
<td>30</td>
</tr>
<tr>
<td>Pascal.1</td>
<td>177</td>
<td>21</td>
<td>25</td>
</tr>
<tr>
<td>Pascal.2</td>
<td>177</td>
<td>21</td>
<td>25</td>
</tr>
<tr>
<td>Pascal.3</td>
<td>177</td>
<td>21</td>
<td>30</td>
</tr>
<tr>
<td>Pascal.4</td>
<td>177</td>
<td>20</td>
<td>24</td>
</tr>
<tr>
<td>Pascal.5</td>
<td>177</td>
<td>21</td>
<td>25</td>
</tr>
<tr>
<td>C.0</td>
<td>212</td>
<td>41</td>
<td>44</td>
</tr>
<tr>
<td>C.1</td>
<td>213</td>
<td>41</td>
<td>44</td>
</tr>
<tr>
<td>C.2</td>
<td>213</td>
<td>41</td>
<td>44</td>
</tr>
<tr>
<td>C.3</td>
<td>213</td>
<td>40</td>
<td>43</td>
</tr>
<tr>
<td>C.4</td>
<td>213</td>
<td>41</td>
<td>44</td>
</tr>
<tr>
<td>C.5</td>
<td>213</td>
<td>40</td>
<td>43</td>
</tr>
<tr>
<td>Java.0</td>
<td>349</td>
<td>56</td>
<td>70</td>
</tr>
<tr>
<td>Java.1</td>
<td>350</td>
<td>56</td>
<td>70</td>
</tr>
<tr>
<td>Java.2</td>
<td>350</td>
<td>53</td>
<td>66</td>
</tr>
<tr>
<td>Java.3</td>
<td>350</td>
<td>56</td>
<td>70</td>
</tr>
<tr>
<td>Java.4</td>
<td>350</td>
<td>55</td>
<td>69</td>
</tr>
<tr>
<td>Java.5</td>
<td>350</td>
<td>53</td>
<td>66</td>
</tr>
</tbody>
</table>

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

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

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

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
3.3. Experimental Validation

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

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

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.](image-url)
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.

8We 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 thorough NFA pruning algorithm, which consists of a full reachability analysis of the NFA. For a complete description 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.
Table 3.4: Updated results of grammar filtering experiments shown in Table 3.1, measured with optimized version of the AMBI DEXTER tool. Improvements in number of harmless production rules found are highlighted in boldface.
3.5. Appendix: Updated Measurement Results

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.

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Previous best</th>
<th>Time AMBER</th>
<th>SLR1</th>
<th>LALR1</th>
<th>LR1</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.2</td>
<td>&gt;15.3h (SLR1)</td>
<td>-</td>
<td>19.1h</td>
<td>8.4s</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>Java.1</td>
<td>3.7h (LR1)</td>
<td>3.1h</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Java.3</td>
<td>2m21s (LR1)</td>
<td>1m54s</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>11</td>
</tr>
</tbody>
</table>

*only reached string length of 11.

Table 3.5 shows 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.