Faster Ambiguity Detection by Grammar Filtering

H.J.S. Basten
Centrum Wiskunde & Informatica
P.O. Box 94079
NL-1090 GB Amsterdam
The Netherlands

J.J. Vinju
Centrum Wiskunde & Informatica
P.O. Box 94079
NL-1090 GB Amsterdam
The Netherlands

ABSTRACT

Real programming languages are often defined using ambiguous context-free grammars. Some ambiguity is intentional while other ambiguity is accidental. A good grammar development environment should therefore contain a static ambiguity checker to help the grammar engineer.

Ambiguity of context-free grammars is an undecidable property. Nevertheless, various imperfect ambiguity checkers exist. Exhaustive methods are accurate, but suffer from nontermination. Termination is guaranteed by approximative methods, at the expense of accuracy.

In this paper we combine an approximative method with an exhaustive method. We present an extension to the Noncanonical Unambiguity Test that identifies production rules that do not contribute to the ambiguity of a grammar and show how this information can be used to significantly reduce the search space of exhaustive methods. Our experimental evaluation on a number of real world grammars shows orders of magnitude gains in efficiency in some cases and negligible losses of efficiency in others.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting System

General Terms

Experimentation, Performance

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

In [2] we compared the practical usability of several ambiguity detection methods on a series of grammars[1]. The exhaustive derivation generator AMBER [10] 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 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 [8] (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 2).

In this paper 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 [1] 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 [4] 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

[1]In the current paper we also use CFG ANALYZER [1] which was not included in [2].

© 2010 Association for Computing Machinery. ACM acknowledges that this contribution was authored or coauthored by a contractor or affiliate of the Government of The Netherlands. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
ISBN: 978-1-4503-0063-6/10/03...$10.00

```
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 1: Excerpt from an ambiguity report by Amber on a Java grammar.

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 [5] and [6]. These can benefit from our grammar filtering similarly to AMBER and CFG ANALYZER.

Outline. In Section 2 we explain the NU TEST, how to extend it to identify harmless productions, and how to construct a filtered grammar. Section 3 contains an experimental validation of our method. We summarize our results in Section 4.

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 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 [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 $S \to A \bullet BC$.

2.2 The Noncanonical Unambiguity Test

The Noncanonical Unambiguity test [8] 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.

2.3 LR(0) Approximation

An equivalence relation that normally yields an approximated graph of analyzable size is the "item_0" relation [8]. 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 [7]. The nodes are labeled with the LR(0) items of the grammar and the edges correspond

```
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 2: Excerpt from an ambiguity report by NU test on a Java grammar.

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 $A \to \alpha \bullet B\gamma \overset{d_i}{\longmapsto} B \to \bullet\beta$, where i is the number of the production $B \to \beta$.
- Reduction transitions, of form $B \to \beta \bullet \overset{r_i}{\longmapsto} A \to \alpha B \bullet \gamma$, 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 d_i transition from $A \to \alpha \bullet B\gamma$ does not necessarily match an r_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.

2.4 Finding ambiguity in an item_0 position graph

To find possible ambiguity, 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 possible ambiguity.

An efficient representation of all such simultaneous traversals is a *position pair graph* (PPG). 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 PPG 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 PPG and looking for join points.

To optimize the process of generating PPGs 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 can significantly improve the size of the graph.

2.5 Filtering Harmless Production Rules

The NU TEST stops after a PPG is constructed and the ambiguous path pairs are reported to the user. In our approach we also use the PPG 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 PPG 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 subgraphs can be pruned. This removes 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 PPG. 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 d_i transition is followed from $A \to \alpha \bullet B\gamma$ to $B \to \bullet\beta$, the matching r_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 PPG, but the other is not, it can be safely removed, and the PPG 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 PPG 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 we investigate whether the reduction of the grammar will actually result in performance gains for exhaustive methods.

2.6 Grammar Reconstruction

From applying the previous filtering process we are left with the 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.

The filtered set of productions can represent an incomplete grammar for two reasons. Firstly, non-terminals from the top[2] of the grammar may have been filtered. Secondly, non-terminals might not have any productions left, but they could still occur in productions of other non-terminals (they have been terminalized). To restore the reachability and productivity properties of the grammar, a new start symbol, new terminals, non-terminals, and production rules will have to be introduced.

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[3]:

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: $N_{\mathsf{unpr}} = \{A | A \in N_{\mathsf{def}}, \neg\exists u : A \Longrightarrow^* u$ using only productions in $P_\mathsf{h}\}$.
4. The start symbols of P_h: $S_\mathsf{h} = \{A | A \in N_{\mathsf{def}}, \neg\exists (B \to \beta A \gamma) \in P_\mathsf{h}\}$.
5. New terminals t_A, b_A, e_A for each non-terminal A.
6. New productions to define a new start-symbol S': $P'_S = \{S' \to (b_A)^k A (e_A)^l |\ A \in S_\mathsf{h}, k = \mathsf{minprefix}(A), l = \mathsf{minpostfix}(A)\}$.
7. Productions to complete the unproductive and undefined non-terminals: $P' = P_\mathsf{h} \cup P'_S \cup \{A \to (t_A)^k \mid A \in N_{\mathsf{undef}} \cup N_{\mathsf{unpr}}, k = \mathsf{minlength}(A)\}$.
8. The new set of terminal symbols: $T' = \{a | (A \to \beta a \gamma) \in P'\}$.
9. Finally, the new grammar: $G' = (N_{\mathsf{def}} \cup \{S'\}, T', P', S')$.

Surrounding the non-terminals in S_h with unique terminals at step 6 prevents the rules of S' from being ambiguous with eachother. Also, they make sure that in all derivations of S' up to a certain length, the non-terminals in S_h can not be expanded further than in the original grammar. At step 7 we prevent the non-terminals from being expanded less far than in the original grammar. This way every derivation of the original grammar corresponds to a derivation of equal 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.

[2]This are the non-terminals that are injected directly into the start symbol.

[3]Where $\mathsf{minlength}(A) = \mathsf{min}(\{k | \exists u,\ A \Longrightarrow^* u : k = |u|\})$, $\mathsf{minprefix}(A) = \mathsf{min}(\{k | \exists u, \alpha : S \Longrightarrow^* uA\alpha,\ k = |u|\})$, and $\mathsf{minpostfix}(A) = \mathsf{min}(\{k | \exists u, \alpha : S \Longrightarrow^* \alpha A u,\ k = |u|\})$.

Grammar	Rules	Rules filtered				Time				Memory (Mb)			
		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 1: Results of Filtering (LR1 was run on C and Java after filtering first with SLR1, due to excessive memory usage).

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) PPG 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.

3.1 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 [12], C [13], Pascal [14] and SQL [15] 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. 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[4]: 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[5]:

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[6] and CFG ANALYZER version 03/12/2007[7]. 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.

3.2 Experimental Results

Results of Filtering Prototype. All measurement results of running our filtering prototype on the benchmark grammars are shown in Table 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 filtered more rules than SLR(1). 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 position 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,

[4]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.

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

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

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

Grammar	Time Orig	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 2: Running Amber on filtered and non-filtered grammars.

Grammar	Time Orig	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: Running Cfg Analyzer on filtered and non-filtered grammars.

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 position 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 [9], it should filter all our grammars with LR(0) or SLR(1) in under 4 seconds, if extended.

Results of Amber**.** Table 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.

This confirms our first hypothesis. To test our second hypothesis, we also need to take the run-time of our filtering tool into account. The left part of Figure 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 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 shows the same results as Table 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.

The right part of Figure 4 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,

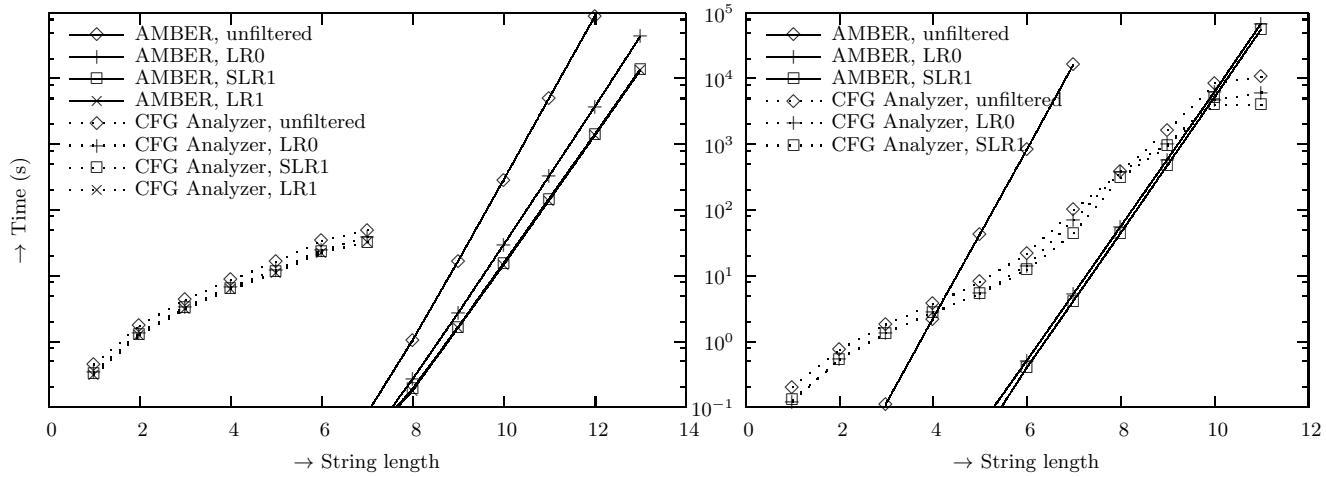

Figure 3: Run-time of Amber and Cfg Analyzer on grammars Java.1 (syntax overloading) left and C.2 (dangling-else) right.

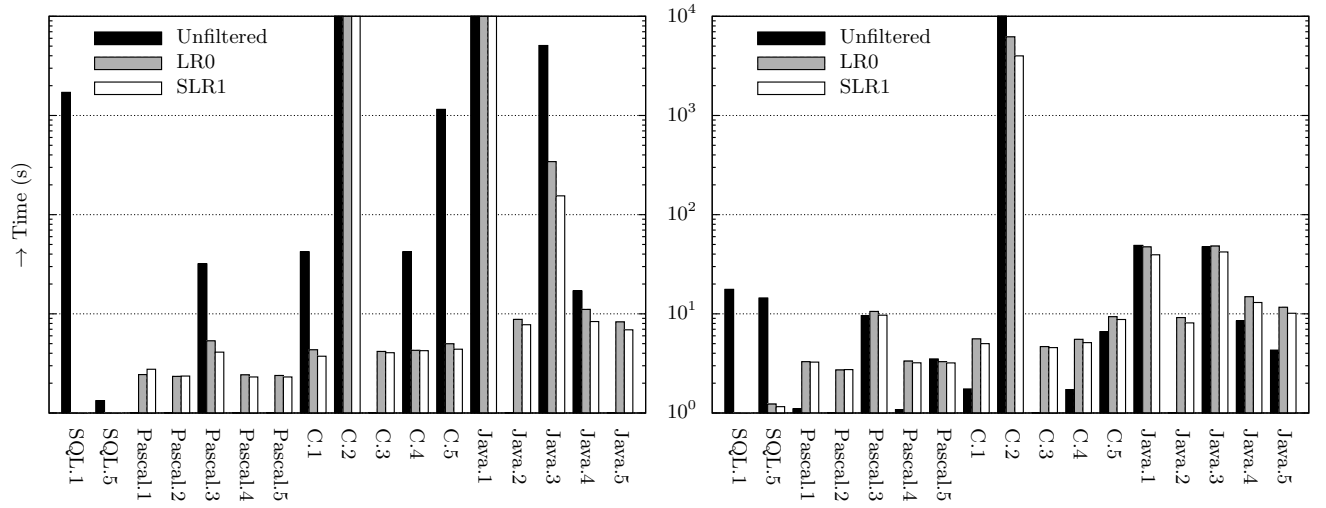

Figure 4: Added run-time of grammar filtering and ambiguity checking with Amber (left) and Cfg Analyzer (right).

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 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.

3.4 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 [9], 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's original implementation in C. An optimized version would eliminate some of the overhead we observed while analyzing small grammars[8].

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

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 tryouts made by language engineers.

About the application of our method to scannerless grammars, such as used by SDF [11], we do not have any information. Assuming that the average token length is about 8 in a language like Java, then to let ambiguity detection methods scale to a scannerless grammars would mean to scale to 8 times the currently maximally feasible sentence length. Also, it remains to be seen if our method applied to scannerless grammars would have a similarly positive effect since such grammars are quite different.

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.

5. REFERENCES

[1] 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'08)*, volume 5126 of *LNCS*, July 2008.

[2] 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.

[3] H. J. S. Basten. Tracking down the origins of ambiguity in context-free grammars. Technical Report SEN-1005, CWI, Amsterdam, The Netherlands, 2010.

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

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

[6] 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`.

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

[8] S. Schmitz. Conservative ambiguity detection in context-free grammars. In L. Arge, C. Cachin, T. Jurdziński, and A. Tarlecki, editors, *ICALP'07: 34th International Colloquium on Automata, Languages and Programming*, volume 4596 of *LNCS*, 2007.

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

[10] F. W. Schröer. AMBER, an ambiguity checker for context-free grammars. Technical report, compilertools.net, 2001. See `http://accent.compilertools.net/Amber.html`.

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

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

[13] C grammar, `ftp://ftp.iecc.com/pub/file/c-grammar.gz`.

[14] Pascal grammar, `ftp://ftp.iecc.com/pub/file/pascal-grammar`.

[15] 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`.