AMBIDEXTER: Practical Ambiguity Detection *Tool Demonstration*

Bas Basten
Centrum Wiskunde & Informatica
basten@cwi.nl

Tijs van der Storm
Centrum Wiskunde & Informatica
Universiteit van Amsterdam
storm@cwi.nl

Abstract

Ambiguity detection tools try to statically track down ambiguities in context-free grammars. Current ambiguity detection tools, however, either are too slow for large realistic cases, or produce incomprehensible ambiguity reports. AMBIDEXTER *is the ambiguity tool to have your cake and eat it too.*

1. Introduction

Context-free grammars are an important form of source code, both for the development of source code analysis and manipulation tools, and for prototyping (domain specific) languages [7]. A common requirement is that a grammar is not ambiguous,—i.e. does not allow multiple derivations for the same string. If a grammar is ambiguous this often indicates a *grammar bug* that needs to be fixed. It is, however, very hard to establish whether a grammar is ambiguous or not.

Ambiguity detection tools are used to statically try and find ambiguities. To be practical, such tools should (1) provide feedback that is comprehensible to the grammar developer, and (2) come up with an answer within reasonable time. In this tool demo we present AMBIDEXTER, a tool for practical ambiguity detection. It combines two approaches to ambiguity detection: exhaustive searching and approximative searching. As a result, AMBIDEXTER benefits from each technique's strengths, while avoiding their respective drawbacks.

2. Ambiguity Detection

A context-free grammar is ambiguous if it allows multiple derivations for the same string. A well-known example of an ambiguous grammar is the following one, describing if-then-else statements:

```
Stm → "if" Exp Stm
Stm → "if" Exp Stm "else" Stm
```

If this grammar is used to parse the string "`if x if y ... else ...`" then it is unclear whether the else-branch belongs to the inner-most if-statement or to the outer-most if-statement. In other words, there are two valid derivation trees for this snippet of source code. Both trees are shown in Figure 1.

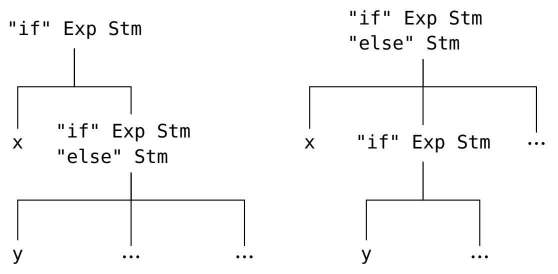

Figure 1. The "dangling else" ambiguity

The goal of ambiguity detection is to statically analyze a grammar and to uncover ambiguities such as the one in Figure 1. Unfortunately, ambiguity of context-free grammars is undecidable in the general case. Nevertheless, there are semi-decision techniques to approach the problem.

Ambiguity detection techniques can be divided over two categories:

1. *Exhaustive searching*: start generating all sentences in the language of a grammar and check if they have multiple parse trees [9, 1].
2. *Approximative searching*: the grammar is approximated into an alternative form that that can be analyzed in finite time [8, 4].

An advantage of exhaustive searching is that if an ambiguous sentence is found, the productions involved in the ambiguity can be derived from the parse trees (cf. Figure 1). This is valuable information for the grammar developer. Unfortunately, generating strings of increasing length is of exponential complexity and will not terminate if the grammar is unambiguous.

On the other hand, approximative searching techniques always terminate, but this comes at the expense of precision. If no ambiguities are found, the grammar is unambiguous. However, if ambiguities are found, they might include false positives. As a result, they require manual inspection. Finally, unlike in exhaustive approaches, the ambiguity reports of these tools tend to be hard to understand.

3. AMBIDEXTER

AMBIDEXTER combines exhaustive and approximative searching to benefit from both their strengths. The goal is to produce precise and comprehensible ambiguity reports as fast as possible. We use approximative filtering to narrow down the search space for an exhaustive checker. This also allows us to detect both ambiguity and unambiguity.

The tool operates in two stages:

1. *Harmless production filtering*: harmless productions are productions that cannot be involved in ambiguity. Using an extension of the approximative technique of [8] such productions are identified and removed from the grammar.

2. *Derivation generator*: for the productions that are not identified as harmless, an exhaustive derivation generator is applied to detect remaining ambiguities.

As a result, AMBIDEXTER leverages the strengths of both approaches to ambiguity detection:

- Unambiguity is detected if all productions are identified as harmless.

- Comprehensible ambiguity reports are produced as a consequence of employing a derivation generator.

- Performance is improved because the production filtering reduces the derivation generator's search space.

The filtering technique is described in more detail in [2]. It has been empirically validated on a collection of real-life programming language grammars [3]. We were able to filter between 12–78% of the productions rules of the various grammars. It turns out this improved the performance of derivation generation substantially, sometimes by orders of magnitude. For instance, on a C grammar with a "dangling else" ambiguity like the one in Figure 1, we witnessed a speedup of AMBER [9] of almost 4000x, after filtering 21% of the production rules.

Our current derivation generator was not included in [3], but early results show it is on average twice as fast as AMBER. Furthermore, we have parallelized it in order to exploit present-day multicore machines for even better performance.

AMBIDEXTER accepts input grammars in YACC [6] and SDF [5] format, and takes into account their disambiguation constructs like priorities and follow restrictions. It is available for download at `http://homepages.cwi.nl/~basten/ambiguity/`.

4. Conclusion

Context-free grammars are important software engineering artifacts. They are used for the development of source code analysis and manipulation tools and for prototyping computer languages. In this tool demo we have presented AMBIDEXTER, a tool for practical ambiguity detection. By combining exhaustive and approximative approaches AMBIDEXTER produces comprehensible ambiguity reports in a much faster time.

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*, 2008.

[2] 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)*. Springer, 2010. To appear.

[3] H. J. S. Basten and J. J. Vinju. Faster ambiguity detection by grammar filtering. In *Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications (LDTA 2010)*. ACM, 2010. To appear.

[4] C. Brabrand, R. Giegerich, and A. Møller. Analyzing ambiguity of context-free grammars. *Sci. Comput. Program.*, 75(3):176–191, 2010.

[5] 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.

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

[7] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Transactions on Software Engineering Methodology*, 14(3):331–380, 2005.

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