Natural and Flexible Error Recovery for Generated Parsers

Maartje de Jonge¹, Emma Nilsson-Nyman², Lennart C.L. Kats¹, and Eelco Visser¹

¹ Dept. of Software Technology, Delft University of Technology, The Netherlands
m.dejonge@tudelft.nl, l.c.l.kats@tudelft.nl, visser@acm.org

² Dept. of Computer Science, Lund University, Sweden
emma@cs.lth.se

Abstract. Parser generators are an indispensable tool for rapid language development. However, they often fall short of the finesse of a hand-crafted parser, built with the language semantics in mind. One area where generated parsers have provided unsatisfactory results is that of error recovery. Good error recovery is both natural, giving recovery suggestions in line with the intention of the programmer; and flexible, allowing it to be adapted according to language insights and language changes. This paper describes a novel approach to error recovery, taking into account not only the context-free grammar, but also indentation usage. We base our approach on an extension of the SGLR parser that supports fine-grained error recovery rules and can be used to parse complex, composed languages. We take a divide-and-conquer approach to error recovery: using indentation, erroneous regions of code are identified. These regions constrain the search space for applying recovery rules, improving performance and ensuring recovery suggestions local to the error. As a last resort, erroneous regions can be discarded. Our approach also integrates bridge parsing to provide more accurate suggestions for indentation-sensitive language constructs such as scopes. We evaluate our approach by comparison with the JDT Java parser used in Eclipse.

1 Introduction

Domain-specific languages offer substantial gains in expressiveness and ease of use for a particular problem domain. To efficiently construct and use domain-specific languages, language development environments should be used, such as IMP [6], the Meta-Environment [27], MontiCore [14], openArchitectureWare [8], or Spoofax/IMP [13]. With these tools, languages are constructed using a grammar as the principal artifact. Using a parser generator, a grammar can be used to automatically generate a parser. When deployed, the parser constructs abstract syntax trees (ASTs) from programs, used to provide the user with syntactical and semantic editor services, such as an outline view and error marking.

Parser generators are an indispensable tool for rapid language development, allowing the language to be quickly changed according to new domain insights and needs. Yet general-purpose programming languages are often still constructed using handcrafted or partially handcrafted parsers. For example, the Java parser used in the popular Eclipse JDT Java editor, is based on a parser generated by JikesPG (now known as LPG) [5].
However, the parser employs handwritten recovery rules as well as a number of large, customized Java components.

The reason often stated for not using a purely generated parser is that they fall short of the finesse of a handcrafted one, built with the language semantics in mind. A particular area where generated parsers have provided unsatisfactory results is that of error recovery, which is essential for parsing incomplete and syntactically incorrect programs, and thus indispensable for interactive editors. Problems with error recovery in generated parsers are the quality of the recovered program and the reported errors, and finding a good trade-off between recovery quality and performance.

Some parser generators allow custom recovery rules to improve error recovery quality \cite{2,5,10,12}. Custom recovery rules allow a language engineer to inspect and improve an error recovery strategy. Compared to a handcrafted parser, a rule-based recovery specification is much easier to maintain, especially as languages are changed or reused to build new languages. Another way to improve error recovery is through grammar analysis, such as LPG’s scope detection \cite{5}.

In previous work we introduced an approach to error recovery that derives properties from grammars to produce explicit, customizable recovery rules \cite{12}. Using scannerless generalized-LR (SGLR) parsing, the approach supports languages with a complex lexical syntax, such as AspectJ \cite{3}, and language embeddings and extensions, such as the Stratego program transformation language with embedded Java fragments \cite{30}. Using generalized parsing, SGLR can parse ambiguous grammars. By considering the different ambiguous meanings of a syntactically incorrect program, through inspection of an expanding search space for applying the set of recovery rules, the approach can provide recovery suggestions that local recovery methods cannot.

An open problem we identified with our approach is that some search space-based suggestions are too “creative” and not natural (i.e., as a programmer would suggest them) \cite{12}; in some cases it is simply better to ignore a small part of the input file, rather than to try and fix it using a combination of insertions and discarded substrings. Another open problem is that for tight clusters of errors, it is not always feasible to provide good suggestions in an acceptable time span.

In order to provide better, more natural suggestions, the present paper proposes an approach to identify the region in which a parse error is found. By restricting the search space for applying the recovery rules to this region, it becomes much less likely that the user is presented with “creative” suggestions that are nowhere near the original error. Using a smaller search space also helps performance. To further help performance, we add a form of “panic mode” \cite{7}: if no solution of applying the recovery rules is found within an acceptable time span, the entire region can be skipped and marked as erroneous. This way, the parser can still continue, report other errors, and construct a partial AST.

We select erroneous regions based on indentation usage. Using indentation, programs typically form logical, nested regions of code. The approach of using layout information for partitioning files has been inspired by the technique of bridge parsing \cite{20}. Bridge parsing is a supplementary technique to grammar-based error recovery. It uses structural information, such as typical use of indentation for bracket placement,