Language-Parametric Techniques for Language-Specific Editors

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op woensdag 29 januari 2014 om 15 uur door

Maartje de JONGE

doctorandus wiskunde
geboren te Amsterdam

Dit proefschrift is goedgekeurd door de promotoren:

Prof. dr. E. Visser
Prof. dr. A. van Deursen

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. E. Visser	Delft University of Technology, promotor
Prof. dr. A. van Deursen	Delft University of Technology, promotor
Prof. dr. R. Grimm	New York University
Prof. dr. S. Thompson	University of Kent
Dr. J. J. Vinju	Centrum Wiskunde & Informatica
Prof. dr. ir. H. J. Sips	Delft University of Technology
Prof. dr. C. M. Jonker	Delft University of Technology

The work in this thesis has been carried out at the Delft University of Technology, under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Cover: *Breaking Waves – Dominic Alves*
(http://www.flickr.com/photos/dominicspics/4036085842)

Printed and bound in The Netherlands by CPI Wöhrmann Print Service.

ISBN 978-94-6203-515-7

Acknowledgments

I owe gratitude to many people that contributed to this dissertation. First and foremost, my sincere thanks go to my promotor Eelco Visser for his guidance and support, and for the genuine interest he has always shown in my research. I particularly appreciated the feedback on my papers, even when it came during the nightly hours just before the deadline. I also want to express my gratitude to my second promotor Arie van Deursen for his help and feedback, and for his optimism and positive encouragement that strengthened my confidence in bringing my PhD to a positive end.

Many people provided valuable feedback that greatly improved the quality of my work. First, I want to thank the members of the reading committee, Robert Grimm, Simon Thompson, Jurgen Vinju, Henk Sips and Catholijne Jonker, for reviewing my thesis. Next, I want to express my gratitude to the people that voluntary provided feedback on one or more chapters: Lennart Kats, Guido Wachsmuth, Bert Meerman and Hugo de Jonge. And finally, I want to thank the anonymous reviewers of my journal and conference papers for their critical reading and their useful suggestions.

All the work of this thesis has been done in collaboration with other people. I want to thank the co-authors of my publications for their contributions: Lennart Kats, Emma Söderberg and Eelco Visser. The techniques proposed in this thesis are implemented as part of the Spoofax language workbench and the JSGLR parser that it uses. I want to thank Karl Trygve Kalleberg for implementing JSGLR, and Lennart Kats for his continuous effort on developing Spoofax which made it into such a valuable platform for research. The WebDSL and Mobl languages turned out to be valuable for the purpose of testing and evaluation. I want to thank Zef Hemel for his implementation of Mobl, and I want to thank him and Danny Groenewegen for their work on the WebDSL language.

The Software Language Design and Engineering (SLDE) group has been a great place to do research. I want to thank all my SLDE colleagues for the teamwork, for providing an inspiring work environment, and for the enjoyable conversations at our coffee corner where we discussed many research, as well as non-research related topics. They are, in alphabetical order: Danny Groenewegen, Eelco Dolstra, Gabriël Konat, Guido Wachsmuth, Lennart Kats, Rob Vermaas, Sander van de Burg, Sander Vermolen, Vlad Vergu and Zef Hemel.

I would also like to thank my best friends Xander Wemmers and Wim van de Fliert with whom I share a common interest in chess, bridge and draughts. I wish to thank Xander for his continuous efforts to improve my chess skills, and the valuable time we spend together in bars and restaurants. My bridge partnership with Wim has been the most enjoyable partnership I can imagine, bridge has never been so much fun ever since. I also owe a lot

to Bert Meerman who inspired me to become a programmer. I will always remember you enlightening the people around you, and especially me, with your deep insights into programming. Thank you for teaching me, motivating me, and supporting me throughout all those years.

I thank my family for patiently listening to me and closely following my progress during my PhD time. I thank my sister, Floortje, for always being there for me, helping me through all ups and downs and inbetweens. I thank my parents, Hugo and Lydia, for their confidence in me and for supporting me in all the choices I made in life.

Finally I want to thank Lennart, the love of my live, for being my support and inspiration for the past five years. First during our happy time together as colleagues at the TU Delft, now as my boyfriend with whom I live together in a nice place in Amsterdam. I look forward to spend my live with you and our lovely son Vincent.

Maartje de Jonge
December 16, 2013,
Amsterdam

Contents

Introduction

1

Programming languages are artificial languages that allow humans to control the behavior of machines. The list of programming languages available today is extensive and still growing. The implementation of those languages requires considerable effort. First, a compiler or interpreter must be implemented to execute programs written in a particular language. Secondly, accompanying tool support must be implemented to help programmers writing programs in that language.

The goal of this dissertation is to develop techniques that simplify the implementation of tool support for new languages. More specifically, we focus on language-parametric solutions for the implementation of language-specific editor support. In the first part of this dissertation we investigate generic techniques to recover from syntax errors that occur during interactive editing. In the second part we look into language-parametric techniques for the implementation of refactoring tools.

In this introductory chapter, we first introduce the research problems that are addressed in this dissertation. We then discuss these problems in more detail and outline our main research questions. Finally, we discuss the research method that we followed to answer these questions.

1.1 PROBLEM STATEMENT

Full-featured integrated development environments (IDEs) have become critical to the adoption of new languages. A key factor in the success of these IDEs is the provision of services specifically tailored to the language. Services for code comprehension help programmers to understand and navigate through the structure of the program, while services for code manipulation offer support to modify the source code in a controlled manner. Figure 1.1 shows a screenshot of an IDE, providing some examples of code comprehension services, e.g., syntax highlighting, outline view, hover help and reference resolving. Examples of code manipulation services are content completion (Figure 1.1), code formatting, quick fixes and refactorings.

To provide language-specific feedback, editor services operate on a structured representation of the program constructed by source code analysis tools. The source code is first syntactically analyzed by the parser which results in an abstract syntax tree (AST) that represents the grammatical structure of the source code. The output of the parser is then further analyzed by a semantic analyzer which adds static semantic information to the AST, e.g., name bindings and types. Figure 1.2 illustrates the process of reconstructing a semantically decorated AST from plain text source code.

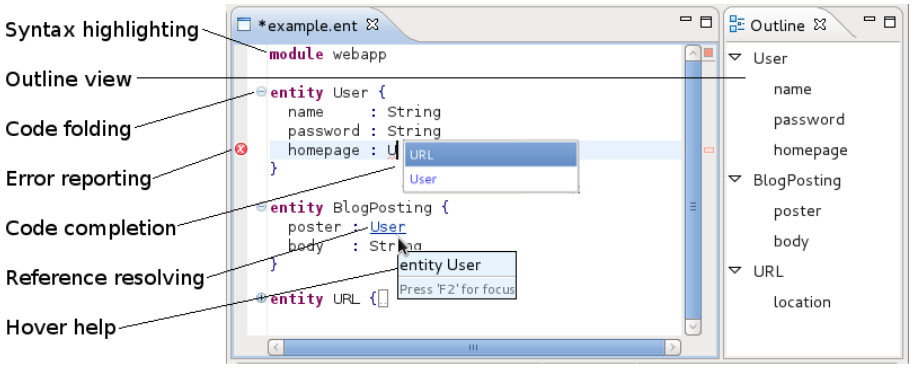


Figure 1.1 Syntactic and semantic editor services in an IDE.

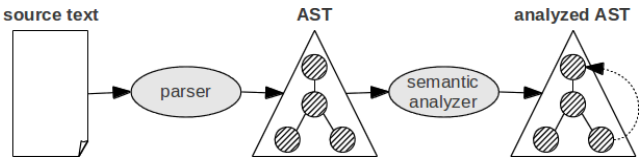


Figure 1.2 Editor services operate on a structured representation of the source program that results after syntactic analysis (parsing) and semantic analysis.

1.1.1 *Parse Error Recovery*

Traditionally, IDEs use hand-tailored parsers, optimized for the language at hand. However, the use of hand-tailored parsers results in high development and maintenance costs and reduces flexibility. Parser generators address this problem by automatically generating a parser from a context-free grammar definition. Context-free grammars are a much more declarative way of describing languages than the program code of a parser. Furthermore, parser generators can be reused for different grammars, and grammars can be reused for other language tooling as well.

The effectiveness and applicability of parser generators in an IDE is determined by their quality and performance. With performance we mean the speed at which the generated parsers consume input tokens, i.e., the time complexity of the parse algorithm. With respect to quality, we distinguish two important criteria, first, the grammar classes that a parser generator supports, secondly, the quality of the syntax error recovery that the generated parsers provide. Below we elaborate on these aspects, describe the challenges in addressing them together, and formulate our first research problem.

Parser generators are distinguished by the grammar classes they support. Traditional parser generators such as Yacc (Johnson, 1975) only support certain subclasses of the context-free grammars, such as LR(k) or LL(k) grammars. The restriction on grammar classes is imposed by the underlying parsing technique. LR(k) parsers can have shift/reduce conflicts, while LL(k)

parsers do not support left recursion. LR(k) and LL(k) parsers are also restricted by a maximum of k lookahead, i.e., the number of incoming tokens that a parser can use to decide how to parse earlier symbols.

A major shortcoming of these parser generators is that they prohibit a natural expression of the intended syntax, i.e., grammars must be carefully crafted to encode them in the supported subclass. A second limitation is that these parser generators do not support modularity of syntax definitions. That is, when two LR(k) grammars are composed, there is no guarantee that the resulting grammar will be LR(k) again, since only the full class of context-free grammars is closed under composition (Hopcroft and Ullman, 1979).

Generalized parsers such as generalized LR (GLR) (Tomita, 1988) and generalized LL (GLL) (Scott and Johnstone, 2010) parse different interpretations of the input in parallel, thereby effectively implementing unlimited lookahead. Both GLR and GLL support the full class of context-free grammars with strict time complexity guarantees[1]. Using scannerless parsing (Salomon and Cormack, 1989, 1995), scannerless GLR (SGLR) (Visser, 1997b) even avoids scanner-level composition problems such as reserved keywords in one language, that may be valid identifiers in another language. The SGLR parsing algorithm supports the modular syntax definition formalism SDF2 (Heering et al., 1989a; Visser, 1997c). SDF's expressiveness allows defining syntax concisely and naturally, SDF's modularity facilitates reuse of existing grammars to compose new grammars. SDF and SGLR are successfully applied in the ASF + SDF Meta-environment (van den Brand et al., 2001) and in the Stratego/XT framework (Visser, 2004).

In Section 1.3 we provide some background on grammar classes and discuss alternative syntactic formalisms that also support grammar composition. For now, we focus on context-free grammars which provide the theoretical context in which our research is done.

Error Recovery for SGLR

To provide rapid syntactic and semantic feedback, IDEs interactively parse programs as they are edited; the parser runs in the background with each key press or after a small delay passes. As the user edits a program, it is often in a syntactically invalid state. Parse error recovery techniques are indispensable for interactive parsing, since they can diagnose and report parse errors, and can construct ASTs for syntactically invalid programs (Degano and Priami, 1995).

Parse error handling encompasses two concerns: error recovery and error reporting. Recovery from parse errors allows the parser to continue the analysis of the source code after the detection of a syntax error, thus allowing the detection of multiple errors in a file. The resulting parse tree, representing the corrected input, is used by editor services to provide syntactic and semantic feedback. Error reporting, by itself, has an important role in giving feedback

[1]Generalized LR (Tomita, 1988) and generalized LL (Scott and Johnstone, 2010) parse LR(1), respectively LL(1) grammars in linear time and gracefully cope with non-determinism and ambiguity with a polynomial worst-case time complexity.

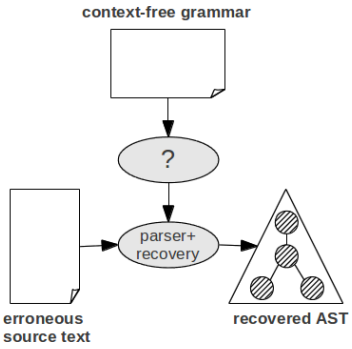


Figure 1.3 *Research Problem 1: How to generate parsers that support the full class of context-free grammars and automatically provide support for error recovery?*

to the user. An error handling technique should accurately report all syntactic errors without introducing spurious errors. The problem of handling syntax errors during parsing has been widely studied (Lévy, 1971; Mauney and Fischer, 1988; Pai and Kieburtz, 1980; Barnard and Holt, 1982; Tai, 1978; Fischer et al., 1980; Degano and Priami, 1995; McKenzie et al., 1995; Corchuelo et al., 2002).

The scannerless, generalized nature of SGLR makes it possible to parse composite languages, but also introduces challenges for the diagnosis and recovery of syntax errors. Parse error recovery for SGLR has been a long-standing open issue (Bravenboer et al., 2006a, Section 8.3.2), which hindered the application of this algorithm in interactive environments with services as illustrated in Figure 1.1. This motivates our first research problem:

Research Problem 1

> *How to generate parsers that support the full class of context-free grammars and automatically provide support for error recovery?*

In the first part of this dissertation we show how parser generators can both be general – supporting the full class of context-free grammars – and automatically provide support for error recovery. This addresses the problem stated above and illustrated in Figure 1.3.

1.1.2 *Refactoring Techniques*

Refactorings are program transformations that improve the design of a program without changing its behavior (Fowler, 2002). Some well known examples are: Rename, Extract method, Move method and Introduce constant. Refactoring tools offer support for a set of predefined structural modifications that are frequently applied by programmers. Refactoring tools automate source code modifications and report errors and warnings for possible behavioral changes. The implementation of program modifications and behavior preservation conditions is an error-prone and tedious task. We are

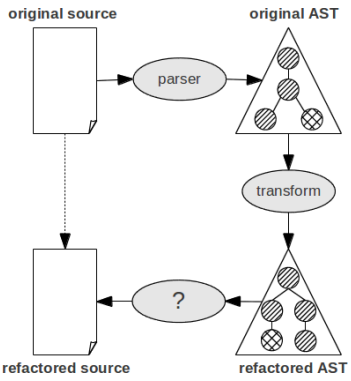


Figure 1.4 *Research Problem 2: How to restore the consistency between the concrete and abstract representation, after transformation of the abstract syntax tree?*

investigating language generic techniques for the efficient implementation of refactoring tools.

Source code reconstruction

Refactorings require deep analysis and transformation of the structure of a program. Therefore, they are most effectively implemented on the abstract syntax tree that results after parsing and semantic analysis. After the transformation on the abstract syntax tree, the consistency between abstract and concrete syntax (source text) must be restored while preserving the original layout of the source code. This is not a trivial task since all layout, i.e., whitespace and comments, is discarded in the AST constructed by the parser. Abstracting over layout simplifies the specification of transformations and analyses, moreover, storing layout in the AST is problematic since it requires a unique mapping between layout elements and AST terms. We formulate the following research problem (see also Figure 1.4):

Research Problem 2

> *How to restore the consistency between the concrete and abstract representation, after transformation of the abstract syntax tree?*

In Chapter 6 we present a language generic algorithm for text reconstruction that translates AST changes to textual changes. The algorithm uses heuristics to reconstruct the layout of the affected code fragments.

Behavior preservation

Refactorings are supposed to preserve the behavior of a program. A prerequisite for behavior preservation is the preservation of *static semantic invariants* (Opdyke, 1992; Schäfer et al., 2009; Tip et al., 2011), i.e., program properties that can be determined at compile time such as name bindings, control-flow and data-flow. Refactoring transformations can accidentally change static se-

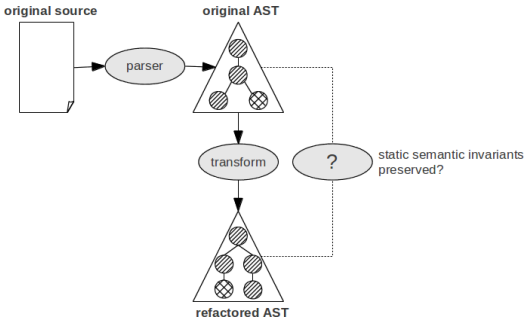


Figure 1.5 *Research Problem 3: How to guarantee preservation of static semantic invariants for refactoring transformations?*

mantic invariants of a program, for example when variable declarations become shadowed by newly introduced names. Users rely on refactoring tools to warn against accidental changes in the static semantics of the program. Therefore, refactoring tools must enforce conditions that guarantee preservation of static semantic invariants. This is not a trivial task, requiring complex name binding and flow analyses. We investigate the research problem stated below and illustrated in Figure 1.5.

Research Problem 3

> *How to guarantee preservation of static semantic invariants for refactoring transformations?*

In Chapter 7 we show how a preservation criterion for static name bindings can be implemented reliably by reusing the name analysis implemented in a compiler. Application of the presented technique for the implementation of other preservation criteria such as semantic correctness, data-flow and control-flow is left as future work. An exploratory study is provided in (de Jonge and Visser, 2013).

1.2 RESEARCH QUESTIONS

In this section we elaborate on the research problems introduced in the previous section, and formulate the research questions that drive the work presented in this dissertation. In Chapter 8, we revisit these questions and present our conclusions.

1.2.1 *Parse Error Recovery*

Unlike conventional parsing algorithms, scannerless generalized-LR parsing supports the full set of context-free grammars, which is closed under composition, and hence can parse languages composed from separate grammar modules. To apply this algorithm in an interactive environment, we investigate parse error recovery techniques for scannerless generalized parsers.

Error recovery for scannerless generalized parsing The scannerless, generalized nature of SGLR is essential for parsing composite languages, but also introduces challenges for the diagnosis and recovery of syntax errors. We have identified two main challenges. First, generalized parsing implies parsing multiple branches (representing different interpretations of the input) in parallel. Syntax errors can only be detected at the point where the last branch failed, which may not be local to the actual root cause of an error. Second, scannerless parsing implies that the parser consumes characters instead of tokens. Common error recovery techniques based on token insertion and deletion are ineffective when applied to characters, as many insertions or deletions are required to modify complete keywords, identifiers, or phrases. Together, these two challenges make it hard to apply traditional error recovery approaches to scannerless generalized parsers. Parse error recovery for SGLR has been a long-standing open issue (Bravenboer et al., 2006a, Section 8.3.2), which motivates our first research question:

Research Question 1

> *What techniques are needed to efficiently recover from syntax errors with scannerless, generalized parsers?*

Chapter 2 introduces a technique for automatic relaxation of grammars, to make them more permissive of their inputs. Based on the analysis of the original grammar, a set of recovery rules that simulate token insertion or deletion is automatically derived. To cope with the added complexity of grammars with recovery rules, the parser implementation is adapted to apply the recovery rules in an on-demand fashion, using a backtracking algorithm.

An indentation based technique for locating parse errors A parser that supports error recovery typically operates by consuming tokens (or characters) until an unexpected token is found. At the point of detection of an error, the recovery mechanism is activated. A key problem for error recovery techniques is the difference between the point of detection and the actual location of an error in the source program (Degano and Priami, 1995).

Local recovery techniques try to modify the token stream at the point of detection so that at least one more original token can be parsed (Degano and Priami, 1995). These techniques work well in some cases, but they risk choosing a poor repair that leads to further problems as the parser continues ("spurious errors"). In contrast to local recovery techniques, global techniques examine the entire program and make a minimum of changes to repair all syntax errors (Aho and Peterson, 1972; Lyon, 1974). While global techniques give an adequate repair in the majority of cases, they are not efficient. As an improvement over local and global recovery techniques, regional techniques only consider the direct context of an error by identifying the region of code in which the error resides (Lévy, 1971; Mauney and Fischer, 1988; Pai and Kieburtz, 1980). The erroneous region can be repaired by a correcting recovery technique, or, as a fall back option, the erroneous region can be discarded. With the aim to improve the quality and performance of the recovery technique we implemented for SGLR, we investigated the following research question:

Research Question 2

What language generic techniques can be used to detect erroneous regions?

Chapter 3 presents an approach that uses layout information to partition files and detect erroneous regions. Evaluation of this approach showed that constraining the search space for recovery rule applications to these regions improves the quality and performance of our recovery technique for SGLR (research question 1). Furthermore, discarding erroneous regions as a fallback recovery helps cope with pathological cases not easily addressed with only recovery rules and backtracking.

Automated evaluation of parse error recovery techniques To assess the quality and performance of our error recovery technique, we need to evaluate the technique against a set of representative test inputs. Evaluations in the literature often use manually constructed test suites based on assumptions about which kind of syntax errors are the most common (Horning, 1976; Kats et al., 2009a; Nilsson-Nyman et al., 2009). The lack of empirical evidence for these assumptions raises the question how representative the test cases are, and how well the technique works in general.

To gain insight into the type and distribution of syntax errors that occur during interactive editing, we performed a statistical analysis on collected edit data for different languages. The analysis focused on the following research question:

Research Question 3a

What kinds of syntax errors occur during interactive editing? How are syntax errors typically distributed over a file?

Section 4.2 summarizes our findings, including a classification of syntax errors that are common for interactive editing and statistical data on how syntax errors are typically spread over a file.

To assess how well a recovery technique meets practical standards, we need to compare the technique with other techniques used in common IDEs. An objective and automated evaluation method is essential to do benchmark comparisons. Unfortunately, objective standards and methods for evaluating error recovery techniques are currently missing in the research literature. We identified two challenges: *1)* the recovery technique must be evaluated against a representative set of test inputs, *2)* the recovery outputs must be automatically evaluated against a quality metric. We formulated the following research question:

Research Question 3b

How to obtain test inputs for error recovery techniques that are representative for practical editing? How to automate quality assessment of the recovered test outputs?

To address this question, we implemented an evaluation framework that combines automated generation of test inputs with automated evaluation of

the outputs. The framework is described in Chapter 4. We used the framework to perform an extensive evaluation of our recovery technique for SGLR which is covered in Section 4.5. The evaluation shows that the technique works for different languages, that the technique is scalable with respect to performance, and that the recovery quality holds up to practical standards.

Integrating error recovery in the Spoofax language workbench Ultimately, error recovery provides a speculative interpretation of the intended program, which may not always be the desired interpretation. As such, it is both unavoidable and not uncommon that editor services operate on inaccurate or incomplete information. Experience with modern IDEs shows that this is not a problem in itself, as programmers are shown both syntactic and semantic errors directly in the editor. Still, there are a number of editor services that inherently require some interaction with the recovery strategy. With the aim to improve the feedback provided by these services, we investigated the following research question:

Research Question 4

> *What general techniques can be used to improve the feedback provided by editor services that interact with the parse error recovery technique?*

In Chapter 5 we describe the implementation of our recovery technique in Spoofax (Kats and Visser, 2010), an integrated environment for the specification of languages and accompanying IDE support in Eclipse. The chapter presents general techniques for the implementation of an IDE based on scannerless, generalized parsing.

1.2.2 *Refactoring Techniques*

Refactorings are behavior preserving source-to-source transformations with the objective of improving the design of existing code (Fowler, 2002). The implementation of refactoring tools is challenging, since different concerns must be handled, e.g., user interaction, the structural transformation, checking behavior preservation conditions and modifying the source code while preserving the original layout. In the second part of this dissertation, we investigate language-parametric techniques for the implementation of refactorings.

Source code reconstruction Automatic refactorings are most effectively implemented on abstract syntax trees which represent the formal structure of a program, abstracting from comments and layout. Abstracting from the arbitrary layout of the source code simplifies the specification of the refactoring, moreover, the structural representation of the program is necessary to reliably perform the analyses needed for correct application.

An intrinsic limitation of transformation techniques based on abstract syntax trees is the loss of layout, i.e., comments and whitespace. This is especially relevant in the context of refactorings, which produce source code that is edited by humans. Thus, we need a technique to restore the consistency between the concrete and abstract representation, while preserving the original

layout. The technique must be generically applicable to different languages, must preserve the originally layout of the unaffected parts and reconstruct the whitespace and comments at the edges of the affected parts. Finally, the technique must also correctly handle lexical symbols such as brackets and separators. This motivates our next research question:

Research Question 5a

> *What language-parametric techniques can be used to derive the textual transformation from the transformation applied to the abstract syntax tree? How to migrate comments and adjust the whitespace at the edges of the changed fragments?*

Chapter 6 presents an algorithm for fully automatic source code reconstruction for source-to-source transformations. The algorithm computes text patches based on the differences between the abstract syntax tree before and after the transformation, relying on origin tracking (van Deursen et al., 1993) as a technique to identify the origins of subtrees. The algorithm preserves the whitespace and comments of the unaffected parts and uses heuristics to reconstruct the layout of the affected parts.

Syntactic sugar provides new language constructs that support expression of functionality that can already be expressed in the base language. These new syntactic constructs make the language "sweeter" for programmers to use; things can be expressed more clearly, more concisely, or in an alternative style that someone may prefer. Desugaring is a step in the transformation process that transforms an abstract syntax tree into an equivalent tree in the core syntax. The specification of refactorings is considerably simplified by desugaring, since the transformation and the semantic analysis only need to be implemented on the core syntax. However, the syntactic sugar must be restored in the result of the refactoring, that is, the language constructs used in the refactored code must be the same as in the original code. Preservation of syntactic sugar is challenging since the information about the original syntactic constructs is lost in the desugaring stage. We formulated the following research question:

Research Question 5b

> *How to extend the text reconstruction algorithm so that it preserves syntactic sugar for refactorings that take as input a desugared AST?*

In Section 6.8 we present an extended version of the text reconstruction algorithm that restores the original syntactic constructs for transformations that are applied to desugared abstract syntax trees.

Name binding preservation Refactoring tools must implement behavior preservation conditions for the supported refactoring transformations. The implementation of behavior preservation conditions is challenging, requiring deep analysis of the semantic structure of the program. Traditionally, conditions for behavior preservation are implemented as preconditions that are checked

before the transformation (Opdyke, 1992; Roberts, 1999). However, it is extremely difficult to define a correct set of preconditions that guarantees behavior preservation, and this set must be updated each time the language evolves.

Another way to approach behavior preservation is to focus on the preservation of *static semantic invariants* (Opdyke, 1992; Ekman et al., 2008; Schäfer et al., 2009; Tip et al., 2011), i.e., program properties that can be determined at compile time, such as name bindings, control-flow, and data-flow. (Ekman et al., 2008) proposes an approach that uses *attribute grammars* to guarantee the preservation of several static semantic invariants. Compared to preconditions, the specification of the invariants more closely follows compiler analyses that define the static semantics of a language. Refactorings implemented using this approach proved to be more reliable and required less effort in terms of lines of code.

Attribute grammars allow for a high-level declarative specification of semantic analysis. However, they offer no specific language features to declaratively express AST transformations. An alternative approach is to use term rewriting for implementing refactorings. Term rewriting makes it easy to describe syntax tree transformations, but is less declarative with respect to semantic analysis. The solution described in (Ekman et al., 2008) crucially relies on attribute grammars[2] and therefore can not be applied directly to term rewriting systems. With the aim to implement an invariant-based approach within the paradigm of term rewriting, we investigated the following research question:

Research Question 6

> *Is it possible to guarantee the preservation of static semantic invariants in term rewriting systems?*

Chapter 7 introduces a static semantic invariant for name binding preservation, which is implemented by reusing the name analysis defined in the compiler front end. We show how the technique can be applied using the Stratego rewriting language for implementing refactorings on Java, Stratego itself, and Mobl. The implementation of other preservation criteria such as semantic correctness, data-flow and control-flow is left as future work. An exploratory study is provided in (de Jonge and Visser, 2013).

1.3 BACKGROUND AND CONTEXT

For our approach to error recovery it is useful to give a bit more background and context, since we base it on the technique of Scannerless Generalized-LR (SGLR) parsing. SGLR is a powerful parsing technique that addresses some of the important problems in parsing: it makes it possible to define languages naturally and declaratively, and it makes it possible to compose languages.

[2] More specifically, the approach depends on circular reference attribute grammars (Hedin, 2000), a specialized form of attribute grammars.

Still, SGLR is by no means a silver bullet, so to give a bit more background we also discuss alternative parsing approaches that also tackle these problems.

1.3.1 *Parsers for Different Grammar Classes*

Parser generators support different classes of grammars. Each grammar class is a specific subset of all possible grammars, and is usually characterized by certain restrictions. For example, some parser generators generate parsers with limited lookahead, limiting the class of supported grammars. Others lack support for grammars that use left recursion, or that have production rules with overlapping left-hand sides.

Restrictive grammar classes can make it hard to directly express languages. They can make it hard to directly map language concepts to grammar rules, without refactoring them to comply to all restrictions in a class. In other words, it may not be possible to express the language in a natural way. Another limitation of restricted grammar classes is that they can make it hard or impossible to compose grammars or production rules, as the composed grammar may not comply to the class restrictions. We discuss different grammar classes and their properties below.

Two of the most well-known grammar classes are LL(k) and LR(k). These grammar classes were first introduced in the 1960s (Knuth, 1965; Lewis II and Stearns, 1968) and have since been supported in various popular parser generators. They are relatively small grammar classes, but using the LL(k) and LR(k) parsing algorithms it is possible to construct efficient, O(n) parsers. Expressing languages in LL(k) or LR(k) is not always easy as the grammar classes enforce various restrictions that manifest themselves in the errors reported by the parser generator (e.g., a shift/reduce conflict for an LR(k) parser) or more obvious restrictions such as not supporting left recursion (for an LL(k) parser). They are also restricted to a lookahead of a maximum of a constant k symbols. The classes are generally not well-suited for composition, since there is no guarantee that composing two LR(k) or LL(k) grammars will give a valid LR(k) or LL(k) grammar again. Schwerdfeger and Van Wyk (2009a) have shown, however, that it is possible to identify certain grammars that can be safely composed with these classes.

In (Ford, 2002), Ford introduces *packrat parsers* which support the associated grammar class *Parsing Expression Grammars* (PEGs) (Ford, 2002, 2004). PEGs do not support the full class of context-free grammars, especially since they lack support for full left recursion (Ford, 2002; Tratt, 2010). However, they are also not a strict subset of this class; using syntactic predicates, parsing expression grammars can recognize languages that are not expressible as context-free grammars. PEG grammars are closed under composition; for example, Hirzel and Grimm (2007) show how all of C and all of Java can be composed using the PEG based *Rats!* generator. PEGs are implemented using packrat parsing. Packrat parsers are recursive descent parsers that use backtracking to attempt alternative productions in a specified order. Packrat

parsers ensure linear time performance by memoization of all intermediate results.

The ANTLR parser generator currently uses the LL(*) parsing strategy (Parr and Fisher, 2011). LL(*) combines packrat parsing with LL-style parsing techniques, applying a grammar analysis algorithm to statically avoid backtracking in many cases. LL(*) provides the same expressivity as PEGs, and is therefore also closed under composition. As with PEGs, LL(*) requires programmers to avoid left recursive grammar rules.

Generalized parsers such as generalized LR (GLR) (Tomita, 1988) and generalized LL (GLL) (Scott and Johnstone, 2010) parse different interpretations of the input in parallel, thereby effectively implementing unlimited lookahead. Both GLR and GLL support the full class of context-free grammars which is closed under composition. Generalized LR and generalized LL parse LR(1) grammars and LL(1) grammars, respectively, in linear time and cope with non-determinism and ambiguity with a polynomial worst-case time complexity (McPeak and Necula, 2004). Empirical evaluation (Bravenboer et al., 2006a) shows that, in practice, parsing with SGLR is linear in the size of the input.

Ambiguities

Grammars can be *ambiguous*, which means that they allow multiple interpretations for the same input string. Ambiguities pose a challenge for grammar writers; in addition to the intended interpretation, an alternate unintended interpretation may exist that must be filtered out. Ambiguities also pose a challenge for composition; combining two unambiguous grammars may result in an ambiguous grammar. Below, we discuss three different approaches to handle ambiguities, taken by the different parsing techniques.

In traditional parsers, supporting LL(k) and LR(k) grammars, any ambiguities in grammars would be impossible because of the restrictions posed on the grammar classes. An error or conflict is reported for grammars outside of these classes, which may indicate an ambiguity but may also be simply something outside of the grammar class. Because of these restrictions, it is hard to express languages in LL(k) or LR(k) in a natural way, and, in general, it is not possible to compose them without adapting the resulting grammar.

PEGs take a very different approach to ambiguity. Ambiguities are always explicitly or implicitly resolved based on greedy matching and the use of the ordered choice operator (i.e., `/`). A composition of two language constructs will always give precedence to one over the other, which resolves ambiguities but can also lead to subtle errors.[3] It's possible that the wrong interpretation is taken unintentionally, but also that a correct interpretation is unreachable because of greedy matching, resulting in a parse error (Schmitz, 2006; Kats et al., 2010b).

[3] A discussion on the PEG mailing list provides some examples of ordering issues taken from the combined C and Java grammar `http://comments.gmane.org/gmane.comp.parsers.peg.general/1`

Finally, generalized parsers such as GLL and GLR handle ambiguities based on explicit disambiguation. If different interpretations exist for a given input string, and the language engineer has not indicated which one takes precedence, then all interpretations are returned. In general, it is not possible to statically determine whether a context-free grammar is ambiguous or not.[4] By returning all interpretations at runtime this approach ensures that ambiguities never go undetected. Any ambiguities found at runtime can be resolved using runtime disambiguation strategies (e.g. semantic disambiguation), or by extending the grammar with additional disambiguation rules.[5] However, one can never be sure that all such ambiguities have been found.

1.3.2 *Scannerless Parsing*

Traditionally, most language processing tools separate a scanning and parsing stage. Scanners are specified using a set of regular expressions that describe the tokens of a language, while parsers are specified using a grammar that describes how tokens can be combined to form a syntactically correct program. The separation between scanning and parsing introduces restrictions on language definitions that complicates composition, e.g., because the scanner recognizes words either as keyword tokens or as identifiers, regardless of the context. Using scannerless parsing (Salomon and Cormack, 1989, 1995), these issues can be elegantly addressed. Scannerless implementations exist for PEGs and packrat parsing (Ford, 2002; Grimm, 2006), as well as for generalized parsing (Visser, 1997b). Another approach is to use context-aware scanning, where the scanner is still a separate entity but is executed on-demand and uses context information from the parser, as e.g. applied by (Wyk and Schwerdfeger, 2007; Grönniger et al., 2008). The basis of the work in this thesis is the scannerless parsing extension for GLR described in (Visser, 1997b).

1.4 APPROACH

The research conducted in this dissertation has a strong base in constructive research (Crnkovic, 2010). Constructive research takes off from an existing well understood situation, pointing out a practically and theoretically relevant problem. The problem is then solved by constructing a new technique that changes and hopefully improves the status quo. As a last step, the proposed technique is validated by analysing the new status quo. The validation can be analytical, i.e., by reasoning about the properties of the technique, as well as empirical, i.e., by experiments with an implementation of the technique.

[4] This is an undecidable problem (Cantor, 1962; Ginsburg and Ullian, 1966). There are ambiguity checker tools, but they either perform an exhaustive search which may not terminate, or they use approximative methods, at the expense of accuracy. A promising new direction is taken by Basten et al. (2012), where both approaches are combined.

[5] This is another area where tools can help, for example Dr. Ambiguity (Basten and Vinju, 2012) helps language engineers by finding grammar-level explanations of ambiguities and proposing possible solutions to eliminate them.

Engineering research compares to engineering practice in that both disciplines involve the construction of artifacts that address relevant practical problems. However, as Hevner et al. (2004) point out, a key difference is that engineering research provides a clear identification of a contribution to the archival knowledge base of foundations and methodologies. Once the research results are codified in the knowledge base, they become applicable to solve engineering problems that occur in practice. Hevner et al. (2004) provide a framework for good software engineering research. The core of the framework consists of seven guidelines that address design as an artifact, problem relevance, design evaluation, research contributions, research rigor, design as a search process, and research communication. We tried to follow these guidelines in our work.

The broader objective of our research is to construct an environment for developing new languages together with their IDEs. To realize this vision, we had to invent new techniques that address open questions in the area of language and IDE engineering. Examples of such questions are: How to generate scannerless, generalized parsers with support for syntactic error recovery? How to derive textual transformations from abstract transformations? How to reuse the language semantics to implement behavior preservation criteria for refactorings?

To answer these questions, we followed a constructive approach. We first obtained an understanding of the identified problem by analyzing existing solutions described in literature. We then reflected on these solutions, determining their strengths and weaknesses. As a next step, we proposed a new solution to the problem that addresses some of the limitations of the existing solutions. Finally, we validated the proposed solution by a theoretical analysis and by experiments with an implementation. The provided solutions and validations contribute knowledge about how the investigated problems can be understood and solved.

For all proposed solutions, we provide an analytical validation of the technique in the form of a critical discussion. In this discussion, we reason about properties such as correctness, complexity and applicability; compare the solution with other solutions; and seek similarities with other, well-known abstract structures. An example of the latter is our analysis of the layout preservation problem (Chapter 6), which we discuss within the conceptual framework of bi-directional transformations.

A risk of an analytical evaluation is that it tends to focus on solving an idealized problem in an idealized world, thereby abstracting over practical details that may challenge the applicability in a real world setting. To gain insight into realistic problems and to evaluate the proposed solutions against realistic criteria, we supported our research contributions by tool implementations that realize the proposed techniques. We used these implementations to conduct experiments that measure relevant properties of the techniques in a real world setting.

To evaluate our error recovery technique, we measured the quality and performance of the technique against a representative set of test inputs. In

addition, we provided a benchmark comparison with the error recovery implemented by the JDT, a state-of-the-art development environment for the Java language. The experiments are covered in sections 2.6, 3.5 and 4.5. To gain insight into the quality of our layout preservation algorithm, we applied the implemented algorithm to a set of code fragments that cover layout patterns discussed in literature. Again, we use the JDT implementation as a baseline comparison (Section 6.9). To verify the correctness of the name binding preservation criterion, we applied an automated test strategy that uses an inverse oracle (Daniel et al., 2007). The experiment is described in Section 7.6.

All implemented techniques are integrated into Spoofax (Kats et al., 2010a), an open source environment for the development of domain-specific languages. Spoofax is used by researchers to develop DSLs of a realistic, representative scale; notable examples include WebDSL (Groenewegen et al., 2008), Mobl (Hemel and Visser, 2011) and SugarJ (Erdweg et al., 2011a,b). In addition, Spoofax has also been used in education, supporting courses on model-driven software development and compiler construction. From this rich application experience we gained valuable feedback that has been a driving force to improve our solutions. Moreover, the application in practice provides confirmatory evidence of the practical applicability of the techniques. The integration of our techniques in Spoofax is covered in Chapter 5 (error recovery) and the technical report (de Jonge and Visser, 2013) (refactoring).

1.5 ORIGIN OF CHAPTERS

All chapters in this dissertation are directly based on peer-reviewed publications at conferences or in journals on programming languages and software engineering. Each chapter has distinct core contributions. However, the different chapters also contain some redundancy to ensure that they are self-contained.

The author of this thesis is the main contributor of all chapters except Chapter 2, being responsible for most of the effort involved in implementing the approach, performing experiments, and writing the text. Chapter 2, 3, and 5, and Section 4.5 is joint work with Lennart Kats, Emma Söderberg and Eelco Visser. Chapter 2 is a revised version of the OOPSLA 2009 paper on error recovery by Kats et al. (2009a), for which the balance of the work was in favor of Kats. The contribution of the thesis author for this chapter includes the implementation of the backtracking algorithm (Section 2.5), and the experimental evaluation with different rule sets (Section 2.6). Chapter 3 is based on the SLE 2009 paper by de Jonge et al. (2009) for which the balance was in favor of the thesis author. The material of these chapters is also covered by the TOPLAS 2012 paper by de Jonge et al. (2012) for which the thesis author was the main contributor. The TOPLAS paper integrates and updates (Kats et al., 2009a) and (de Jonge et al., 2009), and contains some additional material which is covered in Section 4.5 and Chapter 5 of this dissertation.

- Chapter 2 is an updated and revised version of the OOPSLA 2009 paper *Providing Rapid Feedback in Generated Modular Language Environments*

(Kats, de Jonge, Nilsson-Nyman and Visser, 2009a). The content of this chapter is also covered in de Jonge, Kats, Nilsson-Nyman and Visser (2012).

- Chapter 3 is an updated and revised version of the SLE 2009 paper *Natural and Flexible Error Recovery for Generated Parsers* (de Jonge, Kats, Nilsson-Nyman and Visser, 2009). The content of this chapter is also covered in de Jonge, Kats, Nilsson-Nyman and Visser (2012).
- Chapter 4 is an extended version of the ASE 2012 paper *Automated Evaluation of Syntax Error Recovery* (de Jonge and Visser, 2012b). Section 4.5 contains material from de Jonge, Kats, Nilsson-Nyman and Visser (2012).
- Chapter 5 incorporates material from the TOPLAS 2012 paper *Natural and Flexible Error Recovery for Generated Modular Language Environments* (de Jonge, Kats, Nilsson-Nyman and Visser, 2012).
- Chapter 6 is an extended version of the SLE 2012 paper *An Algorithm for Layout Preservation in Refactoring Transformations* (de Jonge and Visser, 2012a).
- Chapter 7 covers the LDTA 2012 paper *A Language Generic Solution for Name Binding Preservation in Refactorings* (de Jonge and Visser, 2012c).

Part I

Error Recovery for Generated Modular Language Environments

Error Recovery for Scannerless Generalized Parsing

2

ABSTRACT

Integrated development environments (IDEs) increase programmer productivity, providing rapid, interactive feedback based on the syntax and semantics of a language. Key components for the realization of IDE plugins are the language's grammar and generated parser. Unlike conventional parsing algorithms, scannerless generalized LR parsing supports the full set of context-free grammars, which is closed under composition, and hence can parse language embeddings and extensions composed from separate grammar modules. To apply this algorithm in an interactive environment, this chapter introduces a novel error recovery mechanism, which allows it to be used on files with syntax errors – common in interactive editing.

2.1 INTRODUCTION

Integrated Development Environments (IDEs) increase programmer productivity by combining a rich toolset of generic language development tools with services tailored for a specific language. These services provide programmers rapid, interactive feedback based on the syntactic structure and semantics of the language. High expectations with regard to IDE support place a heavy burden on the shoulders of developers of new languages.

One burden in particular for textual languages is the development of a parser. Traditionally, IDEs have often used handtailored parsers. Doing so introduces high development and maintenance costs and reduces flexibility, especially with language extensions and combinations in mind. Parser generators address this problem by automatically generating a parser from a grammar definition. They significantly reduce the development time of the parser and the turnaround time for changing it as a language design evolves. Thus, parser generators are essential for the efficient development of language tools.

The effectiveness and applicability of parser generators in an IDE is determined by their quality and performance. With performance we mean the speed at which the generated parsers consume input tokens, i.e., the time complexity of the parse algorithm. With respect to quality, we distinguish two important criteria, first, the grammar classes that a parser generator supports, secondly, the quality of the syntax error recovery that the generated parsers provide. In this chapter we show how generated parsers can both be general – supporting the full class of context-free languages – and automatically provide support for error recovery. Below we elaborate on these aspects,

describe the challenges in addressing them together, and give an overview of our approach.

Composite languages and SGLR Success of a language, in part, depends on interoperability with other languages and systems. Different languages address different concerns. Language composition is a promising approach for providing integrated support for different concerns. However, compositional languages, such as language extensions and language embeddings, further increase the burden for language engineers, as they now have to provide IDE support for a combination of languages or language elements. Therefore, language development tools must offer support for extensions and combinations of languages. How well a tool can support language composition depends on the underlying language techniques it uses.

A limitation of most conventional parsers is that they only support certain subclasses of the context-free grammars, such as LL(k) grammars or LR(k) grammars, reporting conflicts for grammars outside that grammar class. Such restrictions on grammar classes make it harder to change grammars – requiring refactoring – and prohibit the composition of grammars as only the full class of context-free grammars is closed under composition (Hopcroft and Ullman, 1979). Generalized parsers such as generalized LR support the full class of context-free grammars with strict time complexity guarantees[1]. By using scannerless GLR (SGLR) (Visser, 1997b), even scanner-level composition problems such as reserved keywords are avoided.

The scannerless generalized LR parsing algorithm (SGLR) (Visser, 1997b) supports the modular syntax definition formalism SDF (Visser, 1997c). SDF is closed under composition, e.g., existing grammars can be reused and composed to form new languages. This characteristic benefits the definition of non-trivial grammars, in particular the definition of grammars that are composed from two or more independently developed grammars. SDF is declarative yet expressive, and has been used to specify non-trivial grammars for existing languages such as Java, C, and PHP, as well as domain-specific languages, and embeddings and extensions based on these languages (Bravenboer and Visser, 2004).

Error recovery The parser for a programming language forms the foundation of all language-specific editor services. The parser performs a syntactic analysis (parsing) to construct an abstract syntax tree (AST) that represents the grammatical structure of a program. The AST can be used for *syntactic editor services*, such as syntax highlighting, code folding, and outlining. The AST is further analyzed by a semantic analyzer, allowing for *semantic editor services* such as cross-referencing and checking for semantic errors.

To provide the user with rapid syntactic and semantic feedback, programs are interactively parsed as they are edited. As the user edits a program, it is often in a syntactically invalid state. Parse error recovery techniques can diagnose and report parse errors, and can construct a valid AST for syntac-

[1]Generalized LR (Tomita, 1988) parses LR(1) grammars in linear time and gracefully copes with non-determinism and ambiguity with a polynomial worst-case time complexity.

tically invalid programs (Degano and Priami, 1995). Thus, to successfully apply a parser in an interactive setting, proper parse error recovery is of vital importance.

Challenges The scannerless, generalized nature of SGLR is essential for parsing composite languages, but also introduces challenges for implementing error recovery. We have identified two main challenges. (1) *Scannerless parsing*: scannerless parsing implies that there is no separate scanner for tokenization and that errors cannot be reported in terms of *tokens*, but only in terms of *characters*. This results in error messages about a single erroneous character rather than an unexpected or missing token. Moreover, common error recovery techniques based on token insertion and deletion are ineffective when applied to characters, as many insertions or deletions are required to modify complete keywords, identifiers, or phrases. (2) *Generalized parsing*: A GLR parser processes multiple branches (representing different interpretations of the input) in parallel. Syntax errors can only be detected at the point where the last branch failed, which may not be local to the actual root cause of an error. This makes it difficult to properly identify the offending substring or character. Together, these two challenges make it hard to apply traditional error recovery approaches to scannerless generalized parsers. Parse error recovery for SGLR has been a long-standing open issue (Bravenboer et al., 2006a, Section 8.3.2), which hindered the application of this algorithm in interactive environments.

Approach overview This chapter presents a novel approach to error recovery using SGLR. We base our approach on the principles of island grammars (van Deursen and Kuipers, 1999; Moonen, 2001, 2002) and skeleton grammars (Klusener and Lämmel, 2003), defining new production rules for a grammar to make it more permissive of its inputs. These rules either discard substrings in the input or insert literals (i.e. keywords and braces) as necessary, addressing challenge (1). We identified several idioms for defining such recovery rules. Based on the analysis of an existing grammar, we automatically derive sets of these recovery rules, which makes the approach applicable in a parser generator.

To cope with the added complexity of grammars with recovery rules, we adapt the parser implementation to apply the recovery rules in an on-demand fashion, using a backtracking algorithm. This algorithm explores an increasing, backward search space to find a minimal-cost solution for applying the set of recovery rules. This technique allows us to identify the most likely origin of an error, addressing challenge (2).

We have incorporated the approach in the Spoofax/IMP IDE plugin generator (Kats et al., 2009b), to obtain robust editors for composite languages that can provide feedback to the user in the presence of syntactic errors. We have evaluated the error recovery approach using a grammar composed from Stratego and Java.

Contributions The main contribution of this chapter is a language independent approach to error recovery for SGLR. The approach is based on grammar relaxation, i.e., adding new "recovery" productions to make a grammar more

```
public class Authentication {
  public String getPasswordHash(String user) {
    SQL stm = <| SELECT password FROM Users
                 WHERE name = ${user} |>;
    return database.query(stm);
  }
}
```

Figure 2.1 An extension of Java with SQL queries.

permissive of its inputs. Furthermore, the approach involves an adaptation of the SGLR algorithm to efficiently handle the increased complexity of the grammar.

Outline This chapter starts with a motivating study of composite languages in Section 2.2. In Section 2.3 we discuss the notion of *island grammars* and related techniques, which provide the inspiration for our error recovery approach. In Section 2.4 we show how the ideas of island grammars can be used to transform grammars into permissive grammars. Section 2.5 explains the adaptation of the SGLR algorithm to deal with the combinatorial explosion introduced by permissive grammars. Finally, in Section 2.6 we evaluate the approach, comparing different recovery rule sets.

2.2 COMPOSITE LANGUAGES

Composite languages integrate elements of different language components. We distinguish two classes of composite languages: language extensions and embedded languages. Language extensions extend a base language with new, often domain-specific elements. Language embeddings combine two or more existing languages, allowing one language to be nested in the other.

Examples of language extensions include the addition of traits (Ducasse et al., 2006) or aspects (Kiczales et al., 1997) to object-oriented languages, enhancing their support for adaptation and reuse of code. Other examples include new versions of a language, introducing new features to an existing language, such as Java's enumerations and lambda expressions.

Examples of language embeddings include database query expressions integrated into an existing, general-purpose language such as Java (Bravenboer et al., 2010). Such an embedding both increases the expressiveness of the host language and facilitates static checking of queries. Figure 2.1 illustrates such an embedding. Using a special *quotation* construct, an SQL expression is embedded into Java. In turn, the SQL expression includes an *anti-quotation* of a Java local variable. By supporting the notion of quotations in the language, a compiler can distinguish between the static query and the variable, allowing it to safeguard against injection attacks. In contrast, when using only a basic Java API for SQL queries constructed using strings, the programmer must take care to properly filter any values provided by the user.

Language embeddings are sometimes applied in meta-programming for quotation of their object language (Visser, 2002). Transformation languages

```
webdsl-action-to-java-method:
  |[ action x_action(farg*) { stat* }       ]| ->
  |[ public void x_action(param*) { bstm* } ]|
  with param* := <map(action-arg-to-java)> farg*;
       bstm*  := <statements-to-java> stat*
```

Figure 2.2 Program transformation using embedded object language syntax.

such as Stratego (Bravenboer et al., 2008) and ASF+SDF (van den Brand et al., 2002) allow fragments of a language that undergoes transformation to be embedded in the specification of rewrite rules. Figure 2.2 shows a Stratego rewrite rule that rewrites a fragment of code from a domain-specific language to Java. The rule uses meta-variables (written in *italics*) to match "action" constructs and rewrites them to Java methods with a similar signature. SDF supports meta-variables by reserving identifier names in the context of an embedded code fragment.

2.2.1 *Parsing Composite Languages*

A successful approach to effective realization of composite languages is grammar composition. Grammar composition requires a modular, reusable syntax definition formalism, which allows constituent languages to be defined independently, and then composed to form a whole.

A particularly difficult problem in composing language definitions is composition at the lexical level. Consider again Figure 2.2. In the embedded Java language, `void` is a reserved keyword. For the enclosing Stratego language, however, this name is a perfectly legal identifier. This difference in lexical syntax is essential for a clean and safe composition of languages. It is undesirable that the introduction of a new language embedding or extension invalidates existing, valid programs.

The difficulty in combining languages with a different lexical syntax stems from the traditional separation between scanning and parsing. The scanner recognizes words either as keyword tokens or as identifiers, regardless of the context. In the embedding of Java in Stratego this would imply that `void` becomes a reserved word in Stratego as well. Only using a carefully crafted lexical analysis for the combined language, introducing considerable complexity in the lexical states to be processed, can these differences be reconciled. Using scannerless parsing (Salomon and Cormack, 1989, 1995), these issues can be elegantly addressed (Bravenboer et al., 2006a).

The *Scannerless Generalized-LR* (SGLR) parsing algorithm (Visser, 1997b) realizes scannerless parsing by incorporating the generalized-LR parsing algorithm (Tomita, 1988). GLR supports the full class of context-free grammars, which is closed under composition, unlike subsets of the context-free grammars such as LL(k) or LR(k) (Hopcroft and Ullman, 1979). Instead of rejecting grammars that give rise to shift/reduce and reduce/reduce conflicts in an LR parse table, the GLR algorithm interprets these conflicts by efficiently trying all possible parses of a string in parallel, thus supporting grammars with

ambiguities, or grammars that require more look-ahead than incorporated in the parse table. Hence, the composition of independently developed grammars does not produce a grammar that is not supported by the parser, as is frequently the case with LL or LR based parsers.[2]

Language composition often results in grammars that contain ambiguities. Generalized parsing allows declarative disambiguation of ambiguous interpretations, implemented as a post parse filter on the parse tree, or rather the *parse forest*. A disadvantage of post parse disambiguation is the costs in performance, which can be reduced by applying filters at an earlier stage in the parsing process. Dependent on the particulars of specific disambiguation rules, filters might be implemented during parse table generation, during parsing, or after parsing.

As an alternative to parsing different interpretations in parallel, *backtracking parsers* revisit points of the file that allow multiple interpretations. Backtrack parsing is not generalized parsing since a backtracking parser only explores one possible interpretation at a time, stopping as soon as a successful parse has been found. In the case of ambiguities, alternative parses are hidden, which precludes declarative disambiguation. Note that backtracking parsers can also produce all derivations, but this would require an exponential amount of time.

Non-determinism in grammars can negatively affect parser performance. With traditional backtracking parsers, this can lead to exponential execution time in the worst case scenario. Packrat parsers use a form of backtracking with memoization to parse in linear time (Ford, 2002); but, as with other backtracking parsers, they greedily match the first possible alternative instead of exploring all branches in an ambiguous grammar (Schmitz, 2006). In contrast, GLR parsers explore all branches in parallel and run in polynomial time in the worst case. Furthermore, they have the attractive property that they parse the subclass of LR(1) grammars in linear time. While scannerless parsing tends to introduce additional non-determinism, the implementation of parse filters during parsing rather than as a pure post-parse filter eliminates most of this overhead (Visser, 1997a).

2.2.2 *Defining Composite Languages*

The syntax definition formalism SDF (Heering et al., 1989b; Visser, 1997c) integrates lexical syntax and context-free syntax supported by SGLR as the parsing algorithm. Undesired ambiguities in SDF2 definitions can be resolved using declarative *disambiguation filters* specified for associativity, priorities, follow restrictions, reject, avoid and prefer productions (van den Brand et al., 2002). Implicit disambiguation mechanisms such as 'longest match' are avoided. Other approaches, including PEGs (Ford, 2002), language inheritance in MontiCore (Krahn et al., 2008), and the composite grammars of

[2]Note that Schwerdfeger and Van Wyk (2009b) have shown that for some LR grammars it is possible to statically determine whether they compose. They claim that if you accept some restrictions on the grammars, the composition of the "independently developed grammars" will not produce conflicts.

```
module Java-SQL
imports
  Java
  SQL
exports context-free syntax
  "<|" Query "|>" → Expr    {cons("ToSQL")}
  "${" Expr  "}"  → SqlExpr {cons("FromSQL")}
```

Figure 2.3 Syntax of Java with embedded SQL queries, adapted from (Bravenboer et al., 2010). The 'cons' annotation defines the name of the constructed AST term.

ANTLR (Parr and Fisher, 2011), implicitly disambiguate grammars by forcing an ordering on the alternatives of a production — the first (or last) definition overrides the others. Enforcing explicit disambiguation allows undesired ambiguities to be detected, and explicitly addressed by a developer. This characteristic benefits the definition of non-trivial grammars, in particular the definition of grammars that are composed from two or more independently developed grammars.

SDF has been used to define various composite languages, often based on mainstream languages such as C/C++ (Waddington and Yao, 2007), PHP (Bravenboer et al., 2007), and Java (Bravenboer and Visser, 2004; Kats et al., 2008). The example grammar shown in Figure 2.3 extends Java with embedded SQL queries. It imports both the Java and SQL grammars, adding two new productions that integrate the two. In SDF, grammar productions take the form $p_1 \ldots p_n$ `->` s and specify that a sequence of strings matching symbols p_1 to p_n matches the symbol s. The productions in this particular grammar specify a quotation syntax for SQL queries in Java expressions, and vice versa an anti-quotation syntax for Java expressions inside SQL query expressions. The productions are annotated with the `{cons(`*name*`)}` annotation, which indicates the constructor name used to label these elements when an abstract syntax tree is constructed.

2.3 ISLAND GRAMMARS

In this section we provide some background on island grammars (van Deursen and Kuipers, 1999; Moonen, 2001, 2002), which served as an inspiration for our error recovery technique. At the end of this section, we also discuss the related techniques of skeleton grammars (Klusener and Lämmel, 2003) and noise-skipping or fuzzy parsing (Lavie and Tomita, 1993; Koppler, 1997).

Island grammars combine grammar production rules for the precise analysis of specific language constructs (the "islands") with general rules for skipping over the remainder of an input program ("water"). Parsing with an island grammar results in a partial interpretation of the input, consisting of lists of chuncks that form the constructs of interest. From this result, valuable information can be extracted about specific properties of the input program. Island grammars are commonly applied for reverse engineering of legacy applications, for which no formal grammar may be available, or for which many (vendor-specific) dialects exist (Moonen, 2001).

```
module ExtractCalls
exports
  context-free start-symbols
    Module
  context-free syntax
    Chunk*     → Module {cons("Module")}
    WATER      → Chunk  {cons("WATER")}
    "CALL" Id  → Chunk  {cons("Call")}
  lexical syntax
    [\ \t\n]              → LAYOUT
    ~[\ \t\n]+            → WATER {avoid}
    [a-zA-Z][a-zA-Z0-9]*  → Id
  lexical restrictions
    WATER -/- [A-Za-z0-9]
```

Figure 2.4 An island grammar for extracting calls from a legacy application; adapted from (Moonen, 2001).

Island grammars were originally developed using SDF (van Deursen and Kuipers, 1999; Moonen, 2001). The integration of lexical and context-free productions of SDF allows island grammars to be written in a single, declarative specification that includes both lexical syntax for the definition of water and context-free productions for the islands. Figure 2.4 shows an SDF specification of an island grammar that extracts call statements from COBOL programs. Any other statements in the program are skipped and parsed as water.

The first context-free production of the grammar shown in Figure 2.4 defines the `Module` symbol, which is the start symbol of the grammar. A `Module` is a sequence of chunks. Each `Chunk`, in turn, is parsed either as a patch of `WATER` or as an island, in the form of a `Call` construct. The lexical productions define patterns for layout, water, and identifiers. The layout rule, using the special `LAYOUT` symbol, specifies the kind of layout, e.g., whitespace, used in the language. Layout is ignored by the context-free syntax rules, since their patterns are automatically interleaved with optional layout. The `WATER` symbol is defined as the inverse of the layout pattern, using the `~` negation operator. Together, they define a language that matches *any* given character stream.

The parse tree produced for an island is constrained using disambiguation filters that are part of the original SDF specification (van den Brand et al., 2002). First, the lexical restrictions section specifies a restriction for the `WATER` symbol. This rule ensures that water is always greedily matched, and never followed by any other water character. Second, the `{avoid}` annotation on the `WATER` rule specifies a disambiguation filter for these productions, indicating that the production is to be avoided, e.g., at all times, a non-water `Chunk` is to be preferred. When required, more advanced disambiguation filters can be implemented that, for example, select the tree with the least number of water productions.

The following example illustrates how programs are parsed using an island grammar:

```
  CALL CKOPEN USING filetable, status
```

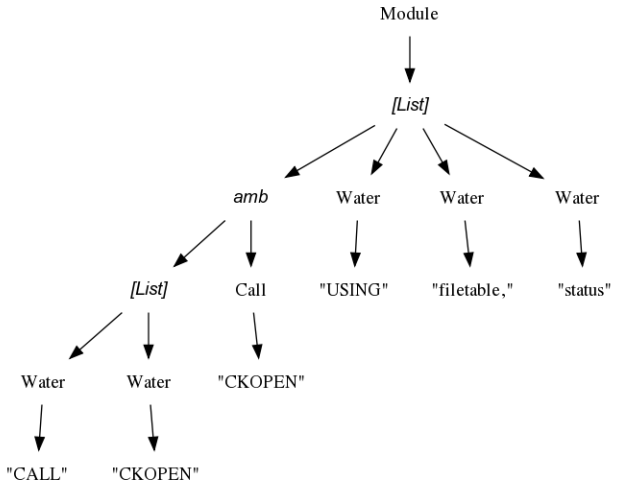


Figure 2.5 The unfiltered abstract syntax tree for a COBOL statement, constructed using the ExtractCalls grammar.

Given this COBOL fragment, a generalized parser can construct a parse tree — or rather a parse *forest* — that includes all valid interpretations of this text. Internally, the parse tree includes the complete character stream, all productions used, and their annotations. In this chapter, we focus on abstract syntax trees (derived from the parse trees) where only the `{cons(name)}` constructor labels appear in the tree. Figure 2.5 shows the complete, ambiguous AST for our example input program. Note in particular the *amb* term, which indicates an ambiguity in the tree: `CALL CKOPEN` in our example can be parsed either as a proper `Call` statement or as `WATER`. Since the latter has an `{avoid}` annotation in its definition, a disambiguation filter can be applied to resolve the ambiguity. Normally, these filters are applied automatically during or after parsing.

Skeleton grammars Similar to island grammars, skeleton grammars Klusener and Lämmel (2003) are tolerant grammars that skip over certain parts of the input that are considered irrelevant. Skeleton grammars are different from island grammars in the sense that they only skip over subtrees. That is, skeleton grammars preserve the context-free structure of the baseline grammar down-to a certain depth in the parse tree; thereby establishing the right syntactic context for the constructs that are considered relevant. For cases where a baseline grammar is available, i.e., a complete grammar for some dialect of a legacy language, Klusener and Lämmel (2003) present a semi-automatic process to derive a skeleton grammar for the selected constructs of interest.

Fuzzy parsing A parser for an island (or skeleton) grammar behaves similar to one that implements a noise-skipping algorithm (Lavie and Tomita, 1993) or a fuzzy parsing strategy (Koppler, 1997). That is, it can skip over any form

```
module Java-15
exports
lexical syntax
  [\ \t\12\r\n]            → LAYOUT
  "\""  StringPart*  "\""  → String
  "/*"  CommentPart* "*/"  → Comment
  Comment                  → LAYOUT
  ...
context-free syntax
  "if" "(" Expr ")" Stm            → Stm {cons("If")}
  "if" "(" Expr ")" Stm "else" Stm → Stm {cons("IfElse"), avoid}
  ...
```

Figure 2.6 Part of the standard Java grammar in SDF; adapted from (Bravenboer et al., 2006a).

of noise in the input file. The main difference is that, using an island grammar, this logic is entirely encapsulated in the grammar definition itself instead of the parsing algorithm.

2.4 PERMISSIVE GRAMMARS

As we have observed in the previous section, there are many similarities between a parser using an island grammar and a noise-skipping parser. In the former case, the water productions of the grammar are used to "fall back" in case an input sentence cannot be parsed, in the latter case, the parser algorithm is adapted to do so. While the technique of island grammars is targeted only towards partial grammar definitions, this observation suggests that the basic principle behind island grammars may be adapted for use in recovery for complete, well-defined grammars.

In the remainder of this section, we illustrate how the notion of productions for defining "water" can be used in regular grammars, and how these principles can be further applied to achieve alternative forms of recovery from syntax errors. We are developing this material in an example-driven way in the sections 2.4.1 to 2.4.3. Then, in Section 2.4.4, we explain how different forms of recovery can be combined. Finally, in Section 2.4.5 we discuss automatic derivation of recovery rules from the grammar, while Section 2.4.6 explains how the set of generated recovery rules can be customized by the language developer.

We focus many of our examples on the familiar Java language. Figure 2.6 shows a part of the SDF definition of the Java language. SDF allows the definition of concrete and abstract syntax in a single framework. The mapping between concrete syntax trees (parse trees) and abstract syntax trees is given by the `{cons(`*name*`)}` annotations. Thus, in the given example, the `{cons("If")}` and `{cons("IfElse")}` annotations specify the name of the constructed abstract syntax terms. Furthermore, the abstract syntax tree does not contain redundant information such as layout between tokens and literals in a production. The `{avoid}` annotation in the second context-free production is used to explicitly avoid the "dangling else problem", a notorious ambiguity that occurs with nested if/then/else statements. Thus, the `{avoid}` annota-

```
module Java-15-Permissive-Chunks
imports Java-15
exports
lexical syntax
  ~[\ \t\12\r\n]+ → WATER {avoid}
lexical restrictions
  WATER -/- ~[\ \t\12\r\n]
context-free syntax
  WATER → Stm {cons("WATER")}
```

Figure 2.7 Chunk-based recovery rules for Java using `avoid`.

```
module Java-15-Permissive-Chunks
imports Java-15
exports
lexical syntax
  ~[\ \t\12\r\n]+ → WATER {recover}
lexical restrictions
  WATER -/- ~[\ \t\12\r\n]
context-free syntax
  WATER → Stm {cons("WATER")}
```

Figure 2.8 Chunk-based recovery rules for Java using `recover`.

tion states that the interpretation of an `IfElse` term with a nested `If` subterm, must be avoided in favor of the alternate interpretation, i.e., an `If` term with a nested `IfElse` subterm. Indeed, Java can be parsed without the use of SGLR, but SGLR has been invaluable for extensions and embeddings based on Java such as those described in (Bravenboer and Visser, 2004; Bravenboer et al., 2006a).

2.4.1 *Chunk-Based Recovery Rules*

Island grammars rely on constructing a grammar based on coarse-grained chunks that can be parsed normally or parsed as water and skipped. This structure is lacking in normal, complete grammars, which tend to have a more hierarchical structure. For example, Java programs consist of one or more classes that each contain methods, which contain statements, etc. Still, it is possible to impose a more chunk-like structure on existing grammars in a coarse-grained fashion; for example, in Java, all statements can be considered as, possibly nested, chunks.

Figure 2.7 extends the standard Java grammar with a coarse-grained chunk structure at the statement level. In this grammar, each `Stm` symbol is considered a "chunk", which can be parsed as either a regular statement or as water, effectively skipping over any noise that may exist within method bodies. Water is defined as any non-empty sequence of non-layout characters. To ensure that water is always greedily matched, a follow restriction is specified (`-/-`), expressing that the WATER symbol is never followed by another water character.

We can extend the grammar of Figure 2.7 to introduce a chunk-like structure at other levels in the hierarchical structure formed by the grammar, e.g. at the method level or at the class level, in order to cope with syntax errors in different places. Doing so leads to a large number of possible interpretations of syntactically invalid (but also syntactically valid) programs. For example, any invalid statement that appears in a method could then be parsed as a "water statement". Alternatively, the entire method could be parsed as a "water method". A preferred interpretation can be picked based on the seize of the regions that are parsed as water.

From Avoid to Recover productions As part of the original SDF specification, the `{avoid}` annotation is used to disambiguate parse trees produced

by grammar productions. An example is the "dangling else" disambiguation shown in Figure 2.6. In Figure 2.7, we use the `{avoid}` annotation on the water production to indicate that preference should be given to parsing statements with regular productions. The key insight of permissive grammars is that this mechanism is sufficient, in principle, to model error recovery.

However, in practice, there are two problems with the use of `{avoid}` for declaring error recovery. First, `{avoid}` is also used in regular disambiguation of grammars. We want to avoid error recover productions *more* than 'normal' `{avoid}` productions. Second, `{avoid}` is implemented as a post-parse filter on the parse forest produced by the parser. This is fine when ambiguities are relatively local and few in number. However, noise-skipping water rules such as those in Figure 2.7 cause massive numbers of ambiguities; each statement can be interpreted as water or as a regular statement, i.e., the parse forest should represent an exponential number of parse trees. While (S)GLR is equipped to deal with ambiguities, their construction has a performance penalty, which is wasteful when there are no errors to recover from.

Thus, we introduced the `{recover}` annotation in SDF to distinguish between the two different concerns of recovery and disambiguation (Figure 2.8). The annotation is similar to `{avoid}`, in that we are interested in parse trees with as few uses of `{recover}` productions as possible. Only in case all remaining branches contain recover productions, a preferred interpretation is selected heuristically by counting all occurrences of the `{recover}` annotation in the ambiguous branches, and selecting the variant with the lowest count. Parse trees produced by the original grammar productions are always preferred over parse trees containing recover productions. Furthermore, `{recover}` branches are disambiguated at runtime, and, to avoid overhead for error-free programs, are only explored when parse errors occur using the regular productions. The runtime support for parsing and disambiguation of recover branches is explained in Section 2.5. Throughout this section we use only the standard, unaltered SDF specification language, adding only the `{recover}` annotation.

Limitations of chunk-based rules While the approach we presented so far can already provide basic syntax error recovery, there are three disadvantages to the recovery rules as presented here. Firstly, the rules are language-specific and are best implemented by an expert of a particular language and its SDF grammar specification. Secondly, the rules are rather coarse-grained in nature; invalid subexpressions in a statement cause the entire statement to be parsed as water. Lastly, the additional productions alter the abstract syntax of the grammar (introducing new `WATER` terminals), causing the parsed result to be unusable for tools that depend on the original structure.

Adapting a grammar to include water productions at different hierarchical levels is a relatively simple yet effective way to selectively skip over "noise" in an input file. In the remainder of this section, we refine this approach, identifying idioms for recovery rules.

Most programming languages feature comments and insignificant whitespace that have no impact on the logical structure of a program. They are generally not considered to be part of the AST. As discussed in Section 2.3, any form of layout, which may include comments, is implicitly interleaved in the patterns of concrete syntax productions. Layout and comments interleave the context-free syntax of a language at a much finer level than the recovery rules we have discussed so far. Consider for example the Java statement

```
if (temp.greaterThan(MAX) /*API change pending*/)
  fridge.startCooling();
```

in which a comment appears in the middle of the statement.

The key idea discussed in this section is to declare water tokens that may occur anywhere that layout may occur. Using this idea, permissive grammars can be defined with noise skipping recovery rules that are language-independent *and* more fine grained than the chunk-based recovery rules described above. To understand how this can be realized, we need to understand the way that SDF realizes 'character-level grammars'.

Intermezzo: layout in SDF In SDF, productions are defined in *lexical syntax* or in *context-free syntax*. Lexical productions are normal context-free grammar productions, i.e., not restricted to regular grammars. The *only* distinction between lexical syntax and context-free syntax is the role of layout. The characters of an identifier (lexical syntax) should not be separated by layout, while layout *may* occur between the sub-phrases of an if-then-else statement, defined in context-free syntax.

The implementation of SDF with scannerless parsing entails that individual characters are the lexical tokens considered by the parser. Therefore, lexical productions and context-free productions are merged into a single context-free grammar with characters as terminals. The result is a character-level grammar that explicitly defines all the places where layout may occur. For example, the `If` production is defined in Kernel-SDF (Visser, 1997c), the underlying core language of SDF, as follows[3]:

```
syntax
  "if" LAYOUT? "(" LAYOUT? Expr LAYOUT? ")" LAYOUT? Stm  →
  Stm {cons("If")}
```

Thus, optional layout is interleaved with the regular elements of the construct. It is not included in the construction of abstract syntax trees from parse trees. Since writing productions in this explicit form is tedious, SDF produces them through a grammar transformation, so that, instead of the explicit rule above, one can write the `If` production as in Figure 2.6:

```
context-free syntax
  "if" "(" Expr ")" Stm → Stm {cons("If")}
```

Water as layout We can use the notion of interleaving context-free productions with optional layout in order to define *deletion recovery rules*, which form

[3]We have slightly simplified the notation that is used for non-terminals in Kernel-SDF.

```
module Java-15-Permissive-Deletions
imports Java-15
exports
lexical syntax
  [A-Za-z0-9\_]+             → DELETEWORD {recover}
  ~[A-Za-z0-9\_\ \t\12\r\n] → DELETESEP  {recover}
  DELETEWORD                  → LAYOUT    {cons("DELETE")}
  DELETESEP                   → LAYOUT    {cons("DELETE")}
lexical restrictions
  DELETEWORD -/- [A-Za-z0-9\_]
```

Figure 2.9 Deletion recovery rules.

a variation of the water recovery rules we have shown so far. Consider Figure 2.9, which combines elements of the comment definition of Figure 2.6 and the chunk-based recovery rules from Figure 2.8. It introduces additional productions into the grammar for layout, which interleaves the context-free syntax patterns. As such, it skips noise on a much finer grained level than our previous grammar incarnation. To separate patches of deleted characters into small chunks, each associated with its own significant `{recover}` annotation, we distinguish between `DELETEWORD` and `DELETESEP` tokens. The production for the `DELETEWORD` token allows to skip over identifier strings, while the production for the `DELETESEP` token allows to skip over special characters that are neither part of identifiers nor whitespace characters. The latter production is defined as an inverse pattern, using the negation operator (`~`). This distinction ensures that large strings, consisting of multiple words and special characters, are counted towards a higher recovery cost.

As an example input, consider a programmer who is in the process of introducing a conditional clause to a statement:

```
if (temp.greaterThan(MAX) // missing )
  fridge.startCooling();
```

Still missing the closing bracket, the standard SGLR parser would report an error near the missing character, and would stop parsing. Using the adapted grammar, a parse forest is constructed that considers the different interpretations, taking into account the new deletion recovery rule. Based on the number of `{recover}` annotations, the following would be the preferred interpretation:

```
if (temp.greaterThan)
  fridge.startCooling();
```

In the resulting fragment both the opening `(` and the identifier `MAX` are discarded, giving a total cost of 2 recoveries. The previous, chunk-based incarnation of our grammar would simply discard the entire `if` clause. While not yet ideal, the new version maintains a larger part of the input. Since it is based on the `LAYOUT` symbol, it also does not introduce new "water" or "deletion" terms into the AST. For reporting errors, the original parse tree, which *does* contain "deletion" terms, can be inspected instead.

The adapted grammar of Figure 2.9 no longer depends on hand-picking particular symbols at different granularities to introduce deletion recovery

```
module Java-15-Permissive-LiteralInsertions
imports Java-15
exports
lexical syntax
   → ")" {cons("INSERT"), recover}
   → "]" {cons("INSERT"), recover}
   → "}" {cons("INSERT"), recover}
   → ">" {cons("INSERT"), recover}
   → ";" {cons("INSERT"), recover}
```

Figure 2.10 Insertion recovery rules for literal symbols.

rules. Therefore, it is effectively language-independent, and can be automatically constructed using only the `LAYOUT` definition of the grammar.

2.4.3 *Insertion Recovery Rules*

So far, we have focused our efforts on recovery by deletion of erroneous substrings. However, in an interactive environment, most parsing errors may well be caused by *missing substrings* instead. Consider again our previous example:

```
if (temp.greaterThan(MAX) // missing )
  fridge.startCooling();
```

Our use case for this has been that the programmer was still editing the phrase, and did not yet add the missing closing bracket. Discarding the opening `(` and the `MAX` identifier allowed us to parse most of the statement and the surrounding file, reporting an error near the missing bracket. Still, a better recovery would be to insert the missing `)`.

One way to accommodate for insertion based recovery is by the introduction of a new rule to the syntax to make the closing bracket optional:

```
context-free syntax
  "if" "(" Expr Stm → Stm {cons("If"), recover}
```

This strategy, however, is rather specific for a single production, and would significantly increase the size of the grammar if we applied it to all productions. A better approach would be to *insert* the particular literal into the parse stream.

Literal insertion SDF allows us to simulate literal insertion using separate productions that virtually insert literal symbols. For example, the lexical syntax section in Figure 2.10 defines a number of basic *literal-insertion recovery rules*, each inserting a closing bracket or other literal that ends a production pattern. This approach builds on the fact that literals such as `")"` are in fact non-terminals that are defined with a production in Kernel-SDF:

```
syntax
  [\41] → ")"
```

Thus, the character 41, which corresponds to a closing brace in ASCII, reduces to the nonterminal ")". A literal-insertion rule extends the defini-

tion of a literal non-terminal, effectively making it optional by indicating that they may match the empty string.[4] Just as in our previous examples, `{recover}` ensures these productions are deferred. The constructor annotation `{cons("INSERT")}` is used as a labeling mechanism for error reporting for the inserted literals. As the `INSERT` constructor is defined in lexical syntax, it is not used in the resulting AST.

Insertion rules for opening brackets In addition to insertions of closing brackets in the grammar, we can also add rules to insert opening brackets. These literals start a new scope or context. This is particularly important for composed languages, where a single starting bracket can indicate a transition into a different sublanguage, such as the `|[` and `<|` brackets of Figure 2.1 and Figure 2.2. Consider for example a syntax error caused by a missing opening bracket in the SQL query of the former figure:

```
SQL stm = // missing <|
    SELECT password FROM Users WHERE name = ${user}
|>;
```

Without an insertion rule for the `<|` opening bracket, the entire SQL fragment could only be recognized as (severely syntactically incorrect) Java code. Thus, it is essential to have insertions for such brackets:

```
lexical syntax
   → "<|" {cons("INSERT"), recover}
```

On literals, identifiers, and reserved words Literal-insertion rules can also be used for literals that are not *reserved words*. This is an important property when considering composed languages since, in many cases, some literals in one sublanguage may not be reserved words in another. As an example, we discuss the insertion rule for the `end` literal in the combined Stratego-Java language.

In Stratego, the literal `end` is used as the closing token of the `if ... then ... else ... end` construct. To recover from incomplete `if-then-else` constructs, a good insertion rule is:

```
lexical syntax
   → "end" {cons("INSERT"), recover}
```

In Java, the string `end` is not a reserved word and is a perfectly legal *identifier*. In Java, identifiers are defined as follows:[5]

```
lexical syntax
  [A-Za-z\_\$][A-Za-z0-9\_\$]* → ID
```

This lexical rule would match a string `end`. Still, the recovery rule will strictly be used to insert the literal `end`, and never an identifier with the name "end".

[4] Insertion rules work best for languages that enforce greedy matching on layout and identifiers, otherwise, these rules introduce a large number of ambiguities that may significantly decrease the performance during recovery.

[5] In fact this production is a simplified version of the actual production. Java allows many other (Unicode) letters and numbers to appear in identifiers.

```
module Java-15-Permissive-LexicalInsertions
imports Java-15
exports
lexical syntax
  INSERTSTARTQ StringPart* "\n" → String {cons("INSERTEND")}
  "\""                          → INSERTSTARTQ  {recover}
  INSERTSTARTC CommentPart* EOF → Comment {cons("INSERTEND")}
  "/*"                          → INSERTSTARTC {recover}
```

Figure 2.11 Insertion recovery rules for string and comment closings.

The reason why the parser can make this distinction is that the literal `end` itself is defined as an ordinary symbol when normalized to kernel syntax:

```
syntax
  [\101] [\110] [\100] → "end"
```

The reason that SDF allows this production to be defined in this fashion is that in the SGLR algorithm, the parser only operates on characters, and the `end` literal has no special meaning other than a grouping of character matches.

The literal-insertion recovery rule simply adds an additional derivation for the `"end"` symbol, providing the parser with an additional way to parse it, namely by matching the empty string. As such, the rule does not change how identifiers (`ID`) are parsed, namely by matching the pattern at the left hand side of the production rule for the `ID` symbol. With a naive recovery strategy that inserts tokens into the stream, identifiers (e.g., `end` in Java) could be inserted in place of keywords. With our approach, these effects are avoided since the insertion recovery rules only apply when a literal is expected.

Insertion rules for string and comment closings Figure 2.11 specifies recovery rules for terminating the productions of the `String` and `Comment` symbols, first seen in Figure 2.6. Both rules have a `{recover}` annotation on their starting literal. Alternatively, the annotation could be placed on the complete production:

```
lexical syntax
  "\"" StringPart* "\n" → String {cons("INSERTEND"),recover}
```

However, the given formulation is beneficial for the runtime behavior of our adapted parser implementation, ensuring that the annotation is considered before construction of the starting literal. The recovery rules for string literals and comments match either at the end of a line, or at the end of the file as appropriate, depending on whether newline characters are allowed in the original, non-recovering productions. An alternative approach would have been to add a literal insertion production for the quote and comment terminator literals. However, by only allowing the strings and comments to be terminated at the ending of lines and the end of file, the number of different possible interpretations is severely reduced, thus reducing the overall runtime complexity of the recovery.

Insertion rules for lexical symbols Insertion rules can also be used to insert lexical symbols such as identifiers. However, lexical symbols do have a representation in the AST, therefore, their insertion requires the introduction of placeholder terms that represent a missing code construct, for example a `NULL()` term. Since placeholder terms alter the abstract syntax of the grammar, their introduction adds to the complexity of tools that process the AST. However, for certain use cases such as content completion in an IDE, lexical insertion can be useful. We revisit the topic in Chapter 5 (Section 5.4) which discusses the integration of our recovery technique with the IDE services that depend on it.

2.4.4 *Combining Different Rule Sets*

The deletion recovery rules of Section 2.4.2 and the insertion rules of Section 2.4.3 can be combined to form a unified recovery mechanism that allows both discarding and insertion of substrings:

```
module Java-15-Permissive
imports
  Java-15-Permissive-Deletions
  Java-15-Permissive-LiteralInsertions
  Java-15-Permissive-LexicalInsertions
```

Together, the two strategies maintain a fine balance between discarding and inserting substrings. Since the recovery strategy assigns additional costs for each recover rule application, insertion or deletion of a single or a few token strings will generally be preferred over inserting or discarding multiple token strings. This ensures that most of the original (or intended) user input is preserved.

2.4.5 *Automatic Derivation*

Automatically deriving recovery rules helps to maintain a valid, up-to-date recovery rule set as languages evolve and are extended or embedded into other languages. Particularly, as languages are changed, all recovery rules that are no longer applicable are automatically removed from the grammar and new recovery rules are added. Thus, automatic derivation helps to maintain language independence by providing a language-parametric approach towards the introduction of recovery rules.

SDF specifications are fully declarative, which allows automated analysis and transformation of a grammar specification. We formulate a set of heuristic rules for the generation of recovery rules based on different production patterns. These rules are applied in a top-down traversal to transform the original grammar into a permissive grammar. The heuristics in this section focus on insertion recovery rules, since these are language specific. The deletion recovery rules are general applicable and added to the transformed grammar without further analysis. The heuristics discussed in this section are based on our experience with different grammars.

So far, we only focused on a particular kind of literals for insertion into the grammar, such as brackets, keywords, and string literals. Still, we need not restrict ourselves to only these particular literals. In principle, any literal in the grammar is eligible for use in an insertion recovery rule. However, for many literals, automatic insertion can lead to unintuitive results in the feedback presented to the user. For example, in the Java language "synchronized" is an optional modifier at the beginning of a class declaration. We don't want the editor to suggest to insert a "synchronized" keyword. In those cases, discarding some substrings instead may be a safer alternative. The decision whether to consider particular keywords for insertion may depend on their semantic meaning and importance (Degano and Priami, 1995). To take this into account, expert feedback on a grammar is needed.

Since we have aimed at maintaining language independence of the approach, our main focus is on more generic, structure-based properties of the grammar. We have identified four different general *classes of literals* that commonly occur in grammars:

- Closing brackets and terminating literals for context-free productions.
- Opening brackets and starting literals for context-free productions.
- Closing literals that terminate lexical productions where no newlines are allowed (such as most string literals).
- Closing literals that terminate lexical productions where newlines are allowed (such as block comments).

Each has its own particular kind of insertion rule, and each follows its own particular definition pattern. We base our generic, language-independent recovery technique on these categories.

By grammar analysis, we derive recovery rules for insertions of the categories mentioned above. With respect to the first and second category, we only derive rules for opening and closing terminals that appear in a balanced fashion with another literal (or a number of other literals). Insertions of literals that are not balanced with another literal can lead to undesired results, since such constructs do not form a clear nesting structure. Furthermore, we exclude lexical productions that define strings and comments, for which we only derive more restrictive insertion rules given by the third and fourth category.

Insertion rules for the first category, *closing bracket* and *terminating literal insertions*, are added based on the following criteria. First, we only consider context-free productions. Second, the first and last symbols of the pattern of such a production must be a literal, e.g., the closing literal appears in a balanced fashion. Finally, the last literal is not used as the starting literal of any other production. The main characteristic of the second category is that it is based on starting literals in context-free productions. We only consider a literal a starting literal if it only ever appears as the first part of a production pattern in all rules of the grammar. For the third category, we only consider productions with identical starting and end literals where no newlines

```
module Java-15
...
context-free syntax
 "{" BlockStm* "}"          → Block {cons("Block")}
 "(" Expr ")"               → Expr  {bracket}
 "while" "(" Expr ")" Stm → Stm   {cons("While")}
 ...
 "void" "." "class"                → ClassLiteral {cons("Void")}
 (Anno | ClassMod)* "class" Id ... → ClassHead    {cons("ClassHead")}
```

Figure 2.12 A selection of context-free productions that appear in the Java grammar.

are allowed in between. Finally, for the fourth category we derive rules for matching starting and ending literals in `LAYOUT` productions. Note that we found that some grammars (notably the Java grammar of (Bravenboer et al., 2006a)) use kernel syntax for `LAYOUT` productions to more precisely control how comments are parsed. Thus, we consider both lexical and kernel syntax for the comment-terminating rules.

As an example, consider the context-free productions of Figure 2.12. Looking at the first production, and using the heuristic rules above, we can recognize that `}` qualifies as a closing literal. Likewise, `)` satisfies the conditions for closing literals we have set. By programmatically analyzing the grammar in this fashion, we collected the closing literal insertion rules of Figure 2.10 which are a subset of the complete set of closing literal insertion rules for Java. From the productions of Figure 2.12 we can further derive the `{` and `(` opening literals. In particular, the `while` keyword is not considered for deriving an opening literal insertion rule, since it is not used in conjunction with a closing literal in its defining production.

No set of heuristic rules is perfect. For any kind of heuristic, an example can be constructed where it fails. We have encountered a number of anomalies that arose from our heuristic rules. For example, based on our heuristic rules, the Java `class` keyword is recognized as a closing literal[6], which follows from the "void" class literal production of Figure 2.12, and from the fact that the `class` keyword is never used as a starting literal of any production. In practice, we have found that these anomalies are relatively rare and in most cases harmless.

We evaluated our set of heuristic rules using the Java, Java-SQL, Stratego, Stratego-Java and WebDSL grammars, as outlined in Section 4.5. For these grammars, a total number of respectively 19 (Java), 43 (Java-SQL), 37 (Stratego), 47 (Stratego-Java) and 32 (WebDSL) insertion rules were generated, along with a constant number of deletion recovery rules as outlined in Figure 2.9. The complete set of derived rules is available from (Permissive, 2011).

[6]Note that for narrative reasons, we did not include an insertion rule for this keyword in Figure 2.10.

Using automatically derived rules may not always lead to the best possible recovery for a particular language. Different language constructs have different semantic meanings and importance. Different languages also may have different points where programmers often make mistakes. Therefore a good error recovery mechanism is not only *language independent*, but is also *flexible* (Degano and Priami, 1995). That is, it allows grammar engineers to use their experience with a language to improve recovery capabilities. Our system, while remaining within the realm of the standard SDF grammar specification formalism, delivers both of these properties.

Language engineers can add their own recovery rules using SDF productions similar to those shown earlier in this section. For example, a common "rookie" mistake in Stratego-Java is to use `[|` brackets `|]` instead of `|[` brackets `]|`. This may be recovered from by standard deletion and insertion rules. However, the cost of such a recovery is rather high, since it would involve two deletions and two insertions. Other alternatives, less close to the original intention of the programmer, might be preferred by the recovery mechanism. Based on this observation, a grammar engineer can add *substitution recovery rules* to the grammar:

```
lexical syntax
  "[|" → "|[" {recover, cons("INSERT")}
  "|]" → "]|" {recover, cons("INSERT")}
```

These rules substitute any occurrence of badly constructed embedding brackets with the correct alternative, at the cost of only a single recovery. Similarly, grammar engineers may add recovery rules for specific keywords, operators, or even placeholder identifiers as they see fit to further improve the result of the recovery strategy.

Besides composition, SDF also provides a mechanism for subtraction of languages. The `{reject}` disambiguation annotation filters all derivations for a particular set of symbols (van den Brand et al., 2002). Using this filter, it is possible to disable some of the automatically derived recovery rules. Consider for example the insertion rule for the `class` keyword, which arose as an anomaly from the heuristic rules of the previous subsection. Rather than directly removing it from the generated grammar, we can disable it by extending the grammar with a new rule that disables the `class` insertion rule.

```
lexical syntax
   → "class" {reject}
```

It is good practice to separate the generated recovery rules from the customized recovery rules. This way, the generated grammar does not have to be adapted and maintained by hand. A separate grammar module can import the generated definitions, while adding new, handwritten definitions. SDF allows modular composition of grammar definitions.

```
i = f ( x ) + 1  ;
i = f ( x   + 1 );
i = f ( x )      ;
i = f (       1 );
i =   ( x ) + 1  ;
i =   ( x   + 1 );
i =     x   + 1  ;
i = f            ;
i =   ( x )      ;
i =     x        ;
i =           1  ;
    f ( x   + 1 );
    f ( x )      ;
    f (       1 );
                 ;
```

Figure 2.13 Multiple interpretations of `i=f(x)+1;` with insertion recovery rules (underlined) and deletion recovery rules.

2.5 PARSING PERMISSIVE GRAMMARS

When all recovery rules are taken into account, permissive grammars provide many different interpretations of the same code fragment. As an example, Figure 2.13 shows many possible interpretations of the string `i=f(x)+1;`. The alternative interpretations are obtained by applying recover productions for inserting parentheses or removing text parts. This small code fragment illustrates the explosion in the number of ambiguous interpretations when using a permissive grammar. The option of inserting opening brackets results in even more possible interpretations, since bracket pairs can be added around each expression that occurs in the program text.

Conceptually, the use of grammar productions to specify how to recover from syntax errors provides an attractive mechanism to parse erroneous fragments. All possible interpretations of the fragment are explored in parallel, using a generalized parser. Any alternative that does not lead to a valid interpretation is simply discarded, while the remaining branches are filtered by disambiguation rules applied by a post processor on the created parse forest. However, from a practical point of view, the extra interpretations created by recover productions negatively affect time and space requirements. With a generalized parser, all interpretations are explored in parallel, which significantly increases the workload for the parser, even if there are no errors to recover from.

In this section we address the performance problems introduced by the multiple recover interpretations. We extend the SGLR algorithm with a selective form of backtracking that is only applied when actually encountering a parsing error. The performance problems during normal parsing are simply avoided by ignoring the recover productions. Figure 2.14 empirically illustrates the need for a backtracking approach, showing parse times measured for different permissive grammars with and without backtracking.

parsed characters	C	D	DC	DC + btr
0	0	0	0	0
10	2	2	12	0
20	2	3	18	0
30	2	7	55	0
40	2	27	92	1
50	3	38	134	1
60	3	59	226	1
70	5	106	870	1
80	21	204	47646	1

Figure 2.14 Parse times in millisecondes for correct prefixes of a Java program. The numbers show the results for different permissive grammars (D=Deletions, C=Insert Closings), using the original implementation (C, D, DC), and using the implementation with support for backtracking (DC + btr).

2.5.1 *Backtracking*

As it is not practical to consider all recovery interpretations in parallel with the normal grammar productions, we need a different strategy to efficiently parse with permissive grammars. As an alternative to parsing different interpretations in parallel, *backtracking parsers* revisit points of the file that allow multiple interpretations, i.e., the *choice points*. Backtrack parsing is not a correct implementation of generalized parsing, since a backtracking parser only produces a single possible parse. However, when applied to error recovery, this is not problematic. For typical cases, parsing only a single interpretation at a time suffices; ultimately, only one recovery solution is needed.

To minimize the overhead of recovery rules, we introduce a selective form of backtracking to (S)GLR parsing that is only used for the concern of error recovery. We ignore all recover productions during normal parsing, and employ backtracking to apply the recovery rules only once an error is detected. Backtracking parsers exhibit exponential behavior in the worst case (Johnstone et al., 2004). For pathological cases with repetitive backtracking, the parser is aborted, and a secondary, non-correcting, recovery technique is applied.

2.5.2 *Choice Points*

A parser that supports error recovery typically operates by consuming tokens (or characters) until an erroneous token is detected. At the point of detection of an error, the recovery mechanism is activated. A major problem for error recovery techniques is the difference between the actual location of the error and the point of detection (Degano and Priami, 1995). Consider for example the erroneous code fragment in Figure 2.15. The superfluous closing bracket (underlined) after the `foo();` statement is obviously intended as a closing bracket for the `if` construct. However, since the `if` construct misses an

```
void methodX() {
   if (true)
      foo();
   }
   int i = 0;
   while (i < 8)
      i=bar(i);
}
```

Figure 2.15 The superfluous closing bracket causes a parse failure at the `while` keyword.

opening bracket, the closing bracket is misinterpreted as closing the method instead of the `if` construct. At that point, the parser simply continues, interpreting the remaining statements as class-body declarations. Consequently, the parser fails at the reserved `while` keyword, which can only occur inside a method body. More precisely, with a scannerless parser, it fails at the unexpected space after the characters `w-h-i-l-e`; the character cannot be shifted and all branches (interpretations at that point) are discarded.

In order to properly recover from a parse failure, the text that precedes the point of failure must be reinterpreted using a correcting recovery technique. Using backtracking, this text is inspected in reverse order, starting at the point of detection, gradually moving backwards to the start of the input file. Using a reverse order helps maintain efficiency, since the actual error is most likely near the failure location.

As generalized LR parsers process different interpretations in parallel, they use a more complicated stack structure than regular LR parsers. Instead of a single, linear stack, they use a graph-structured stack (GSS) that efficiently stores the different interpretation branches, which are discarded as input tokens or characters are shifted (Tomita, 1988). All discarded branches must be restored in case the old state is revisited, which poses a challenge for applying backtracking.

To make it possible to resume parsing from a previous location, the complete stack structure for that location is stored in a choice point. We found that it is prohibitive (in terms of performance) to maintain the complete stack state for each shifted character. To minimize the overhead introduced, we only selectively record the stack structure. Lines have meaning in the structure of programs as units of editing. Typically, parse errors are clustered in the line being edited. We base our heuristic for storing choice points on this intuition. In the current implementation, we create one backtracking choice point for each line of the input file.

2.5.3 *Search Heuristic*

A parse failure indicates that one or more syntax errors reside in the prefix of the program before the failure location. Since it is unlikely that the parser can consume many more tokens after a syntax error, these errors are typically located near the failure location. To recover from multiple errors, multiple corrections are sometimes required. To recover from syntax errors efficiently,

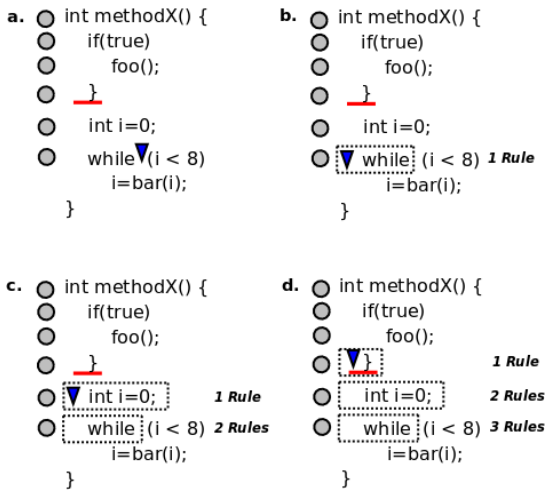


Figure 2.16 Applying error recovery rules with backtracking. The initial point of failure and the start of the recovery search space is indicated by a triangle. The entire search space is indicated using dashed lines, where the numbers to the side indicate the number of recovery rules that can be applied at that line.

we implement a heuristic that expands the search space with respect to the area that is covered and with respect to the number of corrections (recovery rule applications) that are made.

Figure 2.16 illustrates how the search heuristic is applied to recover the Java fragment of Figure 2.15. The algorithm iteratively explores the input stream in reverse order, starting at the nearest choice point. With each iteration of the algorithm, different candidate recoveries are explored in parallel for a restricted area of the file (the dotted lines in Figure 2.16) and for a restricted number of recovery rule applications (indicated at the right of the dotted lines in bold). For each following iteration the size of the area increases with one line, and the branches explored in this area have one more recovery rule application when compared to the previous iteration.

Figure 2.16a shows the parse failure after the `while` keyword. The point of failure is indicated by the triangle. The actual error, at the closing bracket after the `if` statement, is underlined. The figure shows the different choice points that have been stored during parsing using circles in the left margin.

The first iteration of the algorithm (Figure 2.16b) focuses on the line where the parser failed. The parser is reset to the choice point at the start of the line, and enters recovery mode. At this point, only candidate recoveries that use one recover production are considered; alternative interpretations formed by a second recover production are cut off. Their exploration is postponed until the next iteration. In this example scenario, the first iteration does not lead to a valid solution.

For the next iteration, in Figure 2.16c, the search space is expanded with

respect to the size of the inspected area and the number of applied recovery rules. The new search space consists of the line that precedes the point of detection, plus the error detection line where the recovery candidates with two corrections are considered, resuming the interpretations that were previously cut off.

In Figure 2.16d, the search space is again expanded with the preceding line. This time, a valid recovery is found: the application of a deletion recovery rule that discards the closing bracket leads to a valid interpretation of the erroneous code fragment. Once the original line where the error was detected can be successfully parsed, normal parsing continues.

2.5.4 *Algorithm*

The implementation of the recovery algorithm requires a number of (relatively minor) modifications of the SGLR algorithm used for normal parsing. First, productions marked with the `{recover}` attribute are ignored during normal parsing. Second, a choice point is stored at each newline character. Finally, if all branches are discarded and no accepting state is reached, the `Recover` function is called. Once the recovery is successful, normal parsing resumes with the newly constructed stack structure.

Figure 6.12 shows the recovery algorithm in pseudo code. The `Recover` function controls the iterative search process described in Section 2.5.3. The function starts with some initial configuration (line 2–3), initializing the variable `candidates`, and selecting the last inserted choice point. The choice points are then visited in reverse order (line 4–7), until a valid interpretation (non-empty stack structure) is found (line 7).

For each choice point that is visited, the `ParseCandidates` function is called. The `ParseCandidates` function has a twofold purpose (line 16, 17): first, it tries to construct a valid interpretation (line 16) by exploring candidate recover branches; second, it collects new candidate recover branches (line 17) the exploration of which is postponed until the next iteration. Candidate recover branches are recover interpretations of a prefix of the program that are not yet fully explored. The `ParseCandidates` function reparses the fragment that starts at the choice point location and ends at the accept location (line 19–27). We heuristically set the `ACCEPT_INTERVAL` on two more lines and at least twenty more characters being parsed after the failure location. For each character of this fragment, previously cut off candidates are merged into the stack structure (line 23) so that they are included in the parsing (line 24); while new candidates are collected by applying recover productions on the stack structure (line 24, 26, 32).

The main idea, implemented in line 23-26 and the `ParseCharacter` function (line 29–33), is to postpone the exploration of branches that require multiple recover productions, thereby implementing the expanding search space heuristic described in Section 2.5.3.

After the algorithm completes and finds a non-empty set of stacks for the parser, it enters an optional disambiguation stage. In case more than one

```
RECOVER(choicePoints, failureOffset)
1  ▷ Constructs a recovery stack for the parse input after the failure location
2  candidates ← {}
3  cp ← Last inserted choicepoint
4  do
5     (stacks, candidates) ← PARSECANDIDATES(candidates, cp, failureOffset)
6     cp ← Preceding choicepoint (or cp if none)
7  until |stacks| > 0
8  return stacks

PARSECANDIDATES(candidates, choicePoint, failureOffset)
 9  ▷ Parses in parallel previously collected candidate recover branches,
10     while cutting off and collecting new recover candidates
11  ▷ Input:
12     candidates - Unexplored recover branches created in the previous loop
13     choicePoint - The start configuration for the parser
14     failureOffset - Location were the parser originally failed
15  ▷ Output:
16     stacks - recovered stacks at the accept location
17     newCandidates - new unexplored recover branches
18
19  stacks ← choicePoint.stacks
20  offset ← choicePoint.offset
21  newCandidates ← {}
22  do
23     stacks ← stacks ∪ { c | c ∈ candidates ∧ c.offset = offset}
24     (stacks, recoverStacks) ← PARSECHARACTER(stacks, offset, true)
25     offset = offset +1
26     newCandidates ← newCandidatess ∪ recoverStacks
27  until offset = (failureOffset + ACCEPT_INTERVAL)
28  return (stacks, newCandidates)

PARSECHARACTER(stacks, offset, inRecoverMode)
29  ▷ Parses the input character at the given offset.
30  ▷ Output:
31     parseStacks - stacks created by applying normal grammar productions
32     recoverStacks - stacks created by applying recover productions
33  return (parseStacks, recoverStacks)
```

Figure 2.17 A backtracking algorithm to apply recovery rules.

valid recovery is found, stacks with the lowest recovery costs are preferred. These costs are calculated as the sum of the cost of all recovery rules applied to construct the stack. We employ a heuristic that weighs the application of a deletion recovery rule as twice the cost of the application of an insertion recovery rule, which accounts for the intuition that it is more common that a program fragment is incomplete during editing than that a text fragment was not intended and therefore should be deleted. Ambiguities obtained by application of a recovery rule annotated with `{reject}` form a special case. The reject ambiguity filter removes the stack created by the corresponding rule from the GSS, thereby effectively disabling the rule.

2.6 EVALUATION

In their comparative study, Degano and Priami (Degano and Priami, 1995) set out a number of criteria for good error recovery strategies, which we follow in the evaluation of our approach. We distinguish between aspects that impact users, and aspects that are relevant for developers of a language. There are two main criteria with respect to the end user experience:

- **Quality of recovery**: The recovered program should be as close as possible to the program intended by the programmer. Since the AST is used for syntactic and semantic editor services in the IDE, the quality of the reconstructed AST is of great importance for the feedback presented to the user.

- **Performance**: For interactive usage, the error recovery mechanism must not disturb the work flow of the user. We measure the recovery time for syntax errors in a file.

Important criteria for developers of a language or an IDE (plugin) are:

- **Language independence**: An error recovery algorithm should be independent of a particular language, i.e., it should be usable with any given grammar, without introducing a prohibitive amount of work.

- **Flexibility**: The approach must be easily customizable to the insights of language engineers.

- **Transparency**: It should be clear why a particular recovery is presented. The language engineer should have insight into how the recovery works for a given grammar.

This evaluation focuses on selecting the optimal recovery rule set and evaluating the strengths and weaknesses of the technique. The results shown in this section should be considered as preliminary; a more extensive evaluation is provided in Chapter 4 (Section 4.5), which evaluates the effectiveness of the permissive grammar technique when applied in combination with other recovery techniques for SGLR, presented in Chapter 3. In this extensive evaluation we apply the combined techniques to multiple languages, compare

the approach to the quality standard set by the JDT parser, and measure the performance on files of different sizes, with and without syntax errors.

In the remainder of this section we describe our experimental setup, experimentally select an effective set of recovery rules, and discuss the quality and performance results for this set. We will also argue that our approach satisfies the quality criteria defined for language developers.

2.6.1 *Experimental Setup*

We focus our evaluation on the Stratego-Java language, which is a complex language composed from Stratego and Java. We choose the Stratego-Java language since composed languages form an important use case for scannerless generalized parsing. Our extended evaluation in Chapter 4 covers experiments that involve other languages as well.

We follow the evaluation method proposed in Chapter 4. First, we generate a large set of test inputs from a small set of correct base files. The generated test inputs are modified versions of the base files, representing typical editing scenarios as identified in an empirical study on editing behavior. Together with each test input, an oracle interpretation is automatically generated that represents the intended interpretation of the test input. In total, we generated 158 test cases from 5 correct base files. The base files are taken from the *Dryad compiler*, an open compiler for the Java platform (Kats et al., 2008), and the *WebDSL compiler*, a compiler that generates Java code for applications written in WebDSL (Groenewegen et al., 2008).

To measure the quality of a recovery, we calculate the tree edit distance (Chawathe et al., 1996) between the recovered interpretation and the intended interpretation. The scales for the figures we show are calibrated such that they roughly correspond to the human assesment criteria proposed by Pennello and DeRemer (1978). That is, "no diff" corresponds to the *excellent* qualification, a "small diff" (1–25 tree edits) roughly corresponds to the *good* qualification, and a "large diff" (26+ tree edits) approximately corresponds to the *poor* qualification. Figure 4.14 in Chapter 4, Section 4.4.3 illustrates how these thresholds were determined by plotting measured diff values against their corresponding values obtained after human judgement.

To measure the performance, we compute the extra time it takes to recover from one or more errors (the recovery time), by subtracting the parse time of the oracle file from the parse time of the erroneous file. For all performance measures included in this section, an average, collected after three runs, is used. We set a time limit of 5 seconds to cut off recoveries that take an (almost) infinite time to complete. All measuring is done on a "pre-heated" JVM running on a laptop with an Intel(R) Core(TM) 2 Duo CPU P8600, 2.40GHz processor, 4 GB Memory.

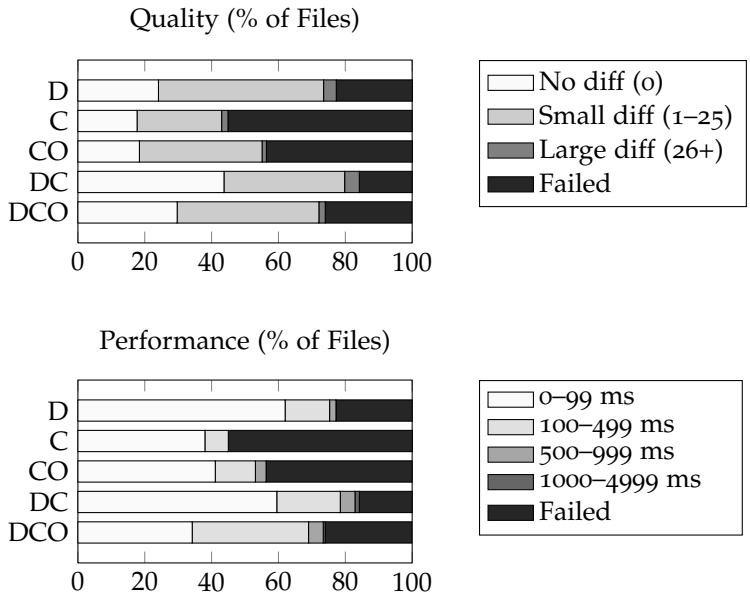


Figure 2.18 Quality and performance (recovery times) using a permissive grammar with different recovery rule sets for Stratego-Java. **D** - Deletions, **C** - Insertion of closing brackets, **O** - Insertion of opening brackets.

2.6.2 *Comparing Different Rule Sets*

In this experiment we focus on selecting the most effective recovery rule set for a permissive grammar with respect to quality and performance. Following Section 2.4, we evaluate three recovery rule sets in isolation and in combination – *Deletions* (D), *insertion of Closing brackets* (C), and *insertion of Open brackets* (O). The rule sets are automatically derived based on the heuristics described in Section 2.4.5. The results from the experiment are shown in Figure 2.18. The figure includes results for D, C, CO, DC and DCO for a Stratego-Java grammar. The remaining combinations, O and DO, are excluded since it is arguably more important to insert closing brackets than to insert open brackets in an interactive editing scenario.

The results show that the insertion of closing brackets (C) and the application of deletion rules (D) both contribute to the quality of a recovery. Combined together (DC) they further improve recovery results. The insertion of opening brackets (O) slightly improves the recovery quality for insertion-only grammars, which follows from comparing C to CO. However, when all rules are combined (DCO), the recovery quality decreases in comparison with the DC grammar. This slightly unexpected result is partly explained by the fact that the insertion rules for opening brackets prove to be too costly with respect to performance, which leads to failures because of exceeding of the time limit set. A second explanation is that the combined rule set (DCO) allows many

creative recoveries that often do not correspond to the human intended recoveries. We conclude that DC seems to be the best trade off between Quality and Performance, providing adequate recoveries in 80% of the cases.

2.6.3 *Pathological Cases*

When inspecting the results for the DC-grammar (Figure 2.18), we notice that there are certain pathological cases where the complete recovery rule set takes too long to find a proper recovery (15.8% failures, given a time limit of 5.0 seconds) or where the recovery leads to objectionable parse times (0.6% in the interval $1.0 - 5.0$ seconds). Furthermore, manual inspection of the large diff results revealed that in some cases a poor repair was chosen with many spurieus errors spread out all over the input file.

We speculate that these pathological cases arise from a combination of syntax errors that can only be resolved by a multitude of recovery operations and/or the presence of particularly liberal productions in the base grammar that prevent early detection of the error, or spurieus error due to a poor recovery. An example of the latter are flat structures such as block comments or multi-line strings. After the opening of the block comment (/*), the parser accepts all characters until the block comment is ended (*/) or the end of the file is reached. As a consequence, a missing block comment ending is typically detected at a large distance from the error location.

A good error recovery approach maintains a fine balance between response time and the quality of a recovery. Successful approaches typically combine multiple strategies, using a secondary strategy if the first does not suffice (Degano and Priami, 1995). The secondary strategy should focus purely on performance, not quality of recovery. In order to still recover from pathological cases, we introduce a fallback recovery strategy in Chapter 3.

2.6.4 *Language Independence, Flexibility and Transparency*

The permissive grammars we evaluated in this section were automatically constructed using the heuristic rules described in Section 2.4.5, showing that the approach works independently of a particular language. Yet by using derived recovery rules, specified as normal SDF productions, transparency is maintained.

In Section 2.4.6 we discussed the customizability of derived grammars; additional recovery rules can be added and undesired rules can be removed to improve the recovery quality. Based on the results of a test set as we constructed for our evaluation, permissive grammars can be manually tuned in order to improve the results.

2.7 RELATED WORK

Recovery techniques for LR parsers The problem of handling syntax errors during parsing has been widely studied (Lévy, 1971; Mauney and Fischer, 1988;

Pai and Kieburtz, 1980; Barnard and Holt, 1982; Tai, 1978; Fischer et al., 1980; Degano and Priami, 1995; McKenzie et al., 1995; Corchuelo et al., 2002). Error recovery techniques can be divided into *correcting* and *non-correcting* techniques. While correcting techniques try to repair syntax errors by diagnosing their cause, non-correcting techniques in contrast recover from errors by skipping (possible large) fragments of the input. Non-correcting techniques are mainly used as a fallback mechanism in case the implemented correcting technique fails.

Correcting recovery methods for LR parsers typically attempt to insert or delete tokens nearby the location of an error, until parsing can resume (Tai, 1978; McKenzie et al., 1995; Cerecke, 2002). There may be several possible corrections of an error which means a choice has to be made. One approach applied by Tai (1978) is to assign a cost (a minimum correction distance) to each possible correction and then choose the correction with the least cost.

We implemented a correcting recovery technique for scannerless generalized LR parsing, based on recovery rules that simulate token insertion or deletion. To minimize the performance costs of applying recovery rules, we employ a search heuristic based on backtracking. The search heuristic determines the order in which recoveries are considered based on the number of applied corrections and their distance relative to the parse failure location. The search heuristic is likely to find an (approximately) minimum cost recovery in most cases.

Recovery techniques for composite languages Using SGLR parsing, our approach can be used to parse composed languages and languages with a complex lexical syntax. In related work, only a study by Valkering (2007), based on substring parsing (Rekers and Koorn, 1991), offered a partial approach to error recovery with SGLR parsing. To report syntactic errors, Valkering inspects the stack of the parser to determine the possible strings that can occur at that point. Providing good feedback this way is non-trivial since scannerless parsing does not employ tokens; often it is only possible to report a set of expected *characters* instead. Furthermore, these error reports are still biased with respect to the location of errors; because of the scannerless, generalized nature of the parser, the point of failure rarely is a good indication of the actual location of a syntactic error. Using substring parsing and artificial reduce actions, Valkering's approach could construct a set of partial, often ambiguous, parse trees, whereas our approach constructs a single, well-formed parse tree.

Lavie and Tomita (1993) developed GLR*, a noise skipping algorithm for context-free grammars. Based on traditional GLR with a scanner, their parser determines the maximal subset of all possible interpretations of a file by systematically skipping selected tokens. The parse result with the fewest skipped words is then used as the preferred interpretation. In principle, the GLR* algorithm could be adapted to be scannerless, skipping characters rather than tokens. However, doing so would lead to an explosion in the number of interpretations. In our approach, we restrict these by using backtracking to only selectively consider the alternative interpretations, and using deletion recov-

ery rules that skip over chunks of characters. Furthermore, our approach supports insertions in addition to discarding noise and provides more extensive support for reporting errors.

Composed languages are also supported by parsing expression grammars (PEGs) (Ford, 2002; Grimm, 2006). PEGs lack the declarative disambiguation facilities (Visser, 1997c) that SDF provides for SGLR. Instead, they avoid ambiguities using greedy matching and enforcing an explicit ordering of productions. To our knowledge, no automated form of error recovery has been defined for PEGs. However, based on the ordering property of PEGS, a "catch all" clause is sometimes added to a grammar, which is used if no other production succeeds. Such a clause can skip erroneous content up to a specific point (such as a newline) but does not offer the flexibility of our approach. Furthermore, existing work on error recovery using parser combinators (Swierstra and Duponcheel, 1996) may be a promising direction for recovery in PEGs.

IDE support for composite languages We integrated our recovery approach into Spoofax (Kats et al., 2009b), a language development environment that combines the construction of languages and editor services for these languages. Using SDF and (J)SGLR[7], Spoofax has the distinguishing feature that it offers full support for language embeddings and extensions, composed from separate grammar modules.

Related projects, also based on SDF and SGLR, respectively SGLL, are the Meta-Environment (van den Brand et al., 2002, 2007) and the Rascal metaprogramming language (Klint et al., 2009). The Meta-Environment currently does not employ interactive parsing, and only parses files after a "save" action from the user. Using the traditional SGLR implementation, it also does not provide error recovery. Rascal offers a platform for language development, integrating with Eclipse through the IMP framework (Charles et al., 2007, 2009). SGLL parsers defined with Rascal are used interactively in Eclipse, but syntax error recovery is not yet supported.

Another language development environment is MontiCore (Krahn et al., 2007, 2008). Based on ANTLR (Parr and Quong, 1995), it uses traditional LL(k) parsing. As such, MontiCore offers only limited support for language composition and modular definition of languages. Combining grammars can cause conflicts at the context-free or lexical grammar level. For example, any keyword introduced in one part of the language is automatically recognized by the scanner as a keyword in another part.

MontiCore supports a restricted form of embedded languages through runtime switching to a different scanner and parser for certain tokens. Using the standard error recovery mechanism of ANTLR, it can provide error recovery for the constituent languages. However, recovery from errors at the edges of the embedded fragments (such as missing quotation brackets), is more difficult using this approach. This issue is not addressed in the papers on MontiCore (Krahn et al., 2007, 2008). In contrast to MontiCore, our approach

[7] `http://strategoxt.org/Stratego/JSGLR/`

is based on scannerless generalized-LR parsing, which supports the full set of context-free grammars, and allows composition of grammars without any restrictions.

Tolerant grammars and fuzzy parsing The basic principles of our permissive grammars are based on the water productions used in island grammars (van Deursen and Kuipers, 1999; Moonen, 2001) and skeleton grammars (Klusener and Lämmel, 2003), also indicated with the collective term *tolerant grammars* (Klusener and Lämmel, 2003). Tolerant grammars are partial grammars that contain only a subset of the baseline grammar's productions, and are more permissive in nature. They have traditionally been used for different reverse and re-engineering tasks. Unlike our permissive grammars, tolerant grammars are not aimed at application in an interactive environment. They do not support the notion of reporting errors and are limited to skipping content. Our approach supports recovery rules that insert missing literals and provides an extensive set of error reporting capabilities. Also, in case of correct input, parsing with a permissive grammar gives exactly the same result as parsing with the baseline grammar.

A related technique is fuzzy parsing, as defined and engineered in (Koppler, 1997). As with tolerant grammars, fuzzy parsers aim at constructing a partial interpretation of the input program by skipping over unparsable fragments. However, they do not implement a purely grammar based approach since they apply a lexical criterion to switch between context-free and ignore mode. Beside application to the domain of reverse and re-engineering tasks, fuzzy parsing is also successfully applied to the domain of IDE development. In (Bischofberger, 1992; Sametinger and Schiffer, 1995) a C++ program development environment is described that employs a fuzzy parser to extract structural information from program files. By skipping over method and function bodies, the parser benefits from increased performance and robustness.

2.8 CONCLUSION

Scannerless, generalized parsers support the full set of context-free grammars, which is closed under composition. With a grammar formalism such as SDF, they can be used for declarative specification and composition of syntax definitions. Error recovery for scannerless, generalized parsers has previously been identified as an open issue. In this chapter, we presented a flexible, language-independent recovery technique to resolve this issue.

The three pillars of our work have been to use standard SDF productions to specify error recovery rules; to automatically generate such error recovery rules from SDF grammars; and to adapt the SGLR parser to efficiently parse files without syntax errors and to gracefully cope with errors locally.

We evaluated our approach using Stratego-Java, a non-trivial language composed from a Stratego and a Java grammar. The results show that, in the majority of cases, our approach achieves adequate recoveries in an acceptable time span. A good error recovery strategy maintains a fine balance between response time and the quality of a recovery. Based on our test set, we

have recognized that there are certain pathological cases where the complete recovery rule set takes too long to find a proper recovery. We address this issue in the next chapter which presents a secondary recovery technique that can be used as a fallback strategy to recover from pathological cases.

An Indentation Based Technique for Locating Parse Errors

3

ABSTRACT

In Chapter 2 we introduced a recovery technique for SGLR based on grammar relaxation. This chapter focuses on two open issues that we identified for this technique. The first is the quality of corrections, which is sometimes lacking since a global, linguistic solution is not aware of the structure of the program expressed by the layout. The second is the performance of the recover algorithm; for some problematic cases an adequate recovery solution cannot be find in an acceptable timespan. To address these issues, this chapter introduces a regional recovery technique that uses layout to select regions of code that enclose syntax errors. The selected regions can be analyzed in detail by a correcting technique, or discarded if no recovery is found within an acceptable time span.

3.1 INTRODUCTION

Integrated Development Environments (IDEs) heavily depend on the parser to determine the grammatical structure of an input program. Given the grammar definition of a language, the parser constructs an abstract syntax tree (AST) of the program. The AST is then further analyzed by the semantic analyzer which adds static semantic information to the AST. The result of the analysis is used by editor services to provide feedback about the syntactic and semantic structure of the program. IDEs parse a file as it is typed in, making incomplete programs and syntax errors the common case rather than the exceptional one. Using error recovery, a parser can still construct a partial abstract syntax tree, allowing the IDE to provide interactive feedback and to report all syntactic errors that occur in the program.

In Chapter 2 we introduced an approach to error recovery for scannerless generalized LR parsing based on grammar relaxation; after analysis of the original grammar, a set of recovery rules that simulate token insertion or deletion is automatically derived. The recovery rules are applied in an on-demand fashion, using a backtracking algorithm. Starting from the parse failure location, this algorithm explores an increasing, backward search space to find a (presumable) minimal-cost solution for applying recovery rules. The backtracking technique allows us to identify the most likely origin of an error, thus providing recovery suggestions that local recovery methods can not.

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 program-

mer would suggest them. 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, or for errors that are detected at a large distance of their actual location, it is not always feasible to provide good suggestions in an acceptable time span.

To address these problems, the present chapter proposes an approach to identify the region in which the parse errors are located. 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 to the original error. This addresses the first problem. Using a smaller search space also helps performance, thereby addressing the second problem. To further help performance, we add a form of "panic mode" (Degano and Priami, 1995); if no solution of applying the recovery rules is found within an acceptable time span, the entire region is skipped and marked as erroneous. This way, the parser can still continue to report other errors and construct a partial AST.

We select erroneous regions based on indentation usage. Indentation typically follows the logical nesting structure of a program, therefore, we can use indentation to partition files into nested blocks that represent code constructs. Code constructs such as statements and methods form free standing blocks, e.g., they can be omitted without influencing the syntactic interpretation of other blocks. It follows that erroneous free standing blocks can simply be skipped, providing a coarse recovery that allows the parser to continue. By subsequently discarding blocks nearby the failure location, we identify the region that contains syntax errors.

We have implemented a staged recovery approach for SGLR that combines the regional recovery technique with the permissive grammar technique described in Chapter 2. The evaluation shows that the combined approach solves most of the quality and performance issues that we noticed for the permissive grammar technique.

Contributions The contribution of this chapter is a regional recovery technique that uses layout to select regions of code that enclose syntax errors. The selected regions can be analyzed in detail by a correcting technique, or discarded if no recovery is found within an acceptable time span. The technique is language independent and can be implemented for different parsing formalisms.

Outline We begin this chapter with a background on error recovery in Section 3.2. Section 3.3 summarizes permissive grammars and backtracking, and discusses its limitations. The limitations are addressed in Section 3.4, which presents region-selection; a layout-sensitive technique to detect discardable, erroneous regions. Finally, Section 3.5 evaluates the application of both recovery techniques together and in isolation.

3.2 PARSE ERROR RECOVERY

Parse error handling encompasses two concerns: error recovery and error reporting. Recovery from parse errors allows the parser to continue the analysis of the source code after the detection of a syntax error. The resulting parse tree represents the corrected input, allowing further analysis of the source code at the semantic level. The quality of the recovered parse tree is decisive for the quality of the syntactic and semantic editor services that depend on it.

The traditional use case of error recovery has been to report multiple errors in a file, thus reducing the typical "compile-fix, compile-fix" cycle into "compile-fix-fix". Error reporting, by itself, has an important role in giving feedback to the user. An error handling technique should accurately report all syntactic errors without introducing spurious errors. This requires accurate diagnosis of errors. A faulty correction may leave the parser in a state that will cause spurious syntactic errors to be reported later.

3.2.1 *Correcting and Non-Correcting Techniques*

Recovery techniques can be divided into correcting and non-correcting techniques. Correcting techniques typically attempt to repair the input string by inserting or deleting tokens until parsing can resume (Tai, 1978; McKenzie et al., 1995; Cerecke, 2002). Based on the modifications to the input string, an error message can be reported to the programmer that indicates the exact location of the error and provides a suggestion for correction. Good error messages reflect the intention of the programmer. When an error is misdiagnosed, the error message issued for it tends to be misleading.

Contrary to correcting techniques, non-correcting techniques do not diagnose the cause of an error, but instead try to recover from errors by skipping parts of the input (Degano and Priami, 1995). Error messages based on non-correcting recoveries tend to be less precise, reporting only the ignored part of the text and/or the location of the token or character that caused the parse failure.

Successful recovery mechanisms often combine more than one technique (Degano and Priami, 1995). For example, a non-correcting technique such as panic mode (Degano and Priami, 1995) is often used as a fall back method if correcting attempts fail. Burke and Fisher (1987) present a recovery approach based on three phases of recovery. The first phase looks for simple correction by the insertion or deletion of a single token. If this does not lead to a recovery, one or more open scopes are closed by inserting scope closing tokens. The last phase consists of discarding tokens that surround the parse failure location.

3.2.2 *Local, Global and Regional Techniques*

A parser that supports error recovery typically operates by consuming tokens (or characters) until an erroneous token is found. At the point of detection of an error, the recovery mechanism is activated. Simple, local approaches to er-

ror recovery will then attempt to adjust the input at the point where the error was detected, so that at least one more original symbol can be parsed (Degano and Priami, 1995). This approach works well in some cases, but in other cases local techniques choose a poor repair that leads to further problems as the parser continues ("spurious errors").

Spurious errors are the result of one of the major problems in error recovery: the difference between the point of detection and the actual location of an error in the source program (Degano and Priami, 1995). In contrast to local methods, global recovery methods examine the entire program and make a minimum of changes to repair all syntax errors (Aho and Peterson, 1972; Lyon, 1974). While these methods give an adequate repair in the majority of cases, they are not efficient.

An alternative approach to local or global recovery is to consider only the direct context of the error, by identifying the region of code in which the error resides (Lévy, 1971; Mauney and Fischer, 1988; Pai and Kieburtz, 1980). Using regions for error recovery has three main advantages. Firstly, they improve the efficiency of a recover algorithm by reducing the search space for corrections. Secondly, by constraining the recovery suggestions to a particular part of the file, they avoid suggestions based on spurious errors that are spread out all over the file. And thirdly, they can be used as a secondary recovery strategy (Degano and Priami, 1995), i.e., erroneous regions can be discarded entirely if a detailed analysis of the region does not provide a better recovery solution.

3.3 PERMISSIVE GRAMMARS AND BACKTRACKING

In Chapter 2 we introduced a recovery technique for scannerless generalized parsing based on the idea to extend grammars with additional "recover" productions to make them more permissive of their inputs. Recover productions are written just as any other production, except that they are annotated with `{recover}`, meaning that they are only applied when recovery is required.

To recover from missing literals, *insertion recovery rules* are defined that simulate token insertion. For example, the production below is used to recover from a missing `}` literal. The production specifies that the empty string (hence the empty left-hand side) can be parsed instead of the closing `}` literal as a possible recovery.

```
→ "}" {recover, cons("INSERT")}
```

In addition to insertion recovery rules, Chapter 2 also defines *deletion recovery rules*, which are lexical "catch-all" production rules to discard unparsable substrings, distinguishing "words" and "separators". As an example, we show the production rule below to skip over "words" by parsing them as layout.

```
[A-Za-z0-9\_]+          → DELETE_WORD {recover}
DELETE_WORD             → LAYOUT {cons("DELETE")}
```

```
SQL stm = // missing <|
    SELECT password
    FROM Users
    WHERE name = ${user}
|>;
```

```
SQL stm;
    SELECT password;
    FROM Users;
    WHERE name = user
;
```

Figure 3.1 Inserting the missing <| token offers an adequate recovery (left fragment). However, if this insertion is not supported, the entire SQL fragment will be parsed as severely incorrect Java code (right fragment).

Recover productions allow for a high-level, grammar-oriented technique of customizing a recovery strategy (Aho and Peterson, 1972; Graham et al., 1979). Because the language engineer must design them a priori, they have sometimes been criticized for being language-dependent (Degano and Priami, 1995). We addressed this issue in Section 2.4.5 by introducing a technique to derive recovery rules from the original grammar.

To cope with the added complexity of grammars with recovery rules, we adapted the parser implementation to apply the recovery rules in an on-demand fashion, using a backtracking algorithm (Section 2.5). This algorithm explores an unbounded, backward search space to, heuristically, find a minimal-cost solution for applying the set of recovery rules.

3.3.1 *Limitations*

Relying on the increasing search space of permissive grammars and backtracking, it is not always feasible to provide good recovery suggestions in an acceptable time span. Problems can arise when the recovery requires many combined corrections, or when the distance between the error location and the detection location is exceptionally large.

The "multiple corrections problem" can occur when multiple errors are tightly clustered, or when no suitable correction is at hand for a particular error. In general, a valid parse can be found, but at the risk of a high performance cost, and potentially resulting in a complex network of recovery suggestions that do not lead to useful feedback for programmers. Figure 3.1 provides an example in wich an entire SQL fragment is parsed as severely incorrect Java code.

The "distance problem" is seen in practice for block comments and other flat structures such as (multi-line) strings. In case a block comment is not properly closed, the error is typically detected only after parsing many more lines, when the next block comment is closed, or at the end of the file. The optimal recovery requires backtracking to the user intended block comment ending. In practice, the algorithm will give priority to an artificial recovery near the detection location, or the extensive backtracking leads to an unacceptable overhead in time. Figure 3.2 shows an unclosed block comment in Java as an example.

To address these problems, the next section introduces an approach to identify the *region* in which the actual error is situated. The erroneous region is

```
class X {
  int i;
  public void foo() {
    if (i > 5 /* max ... ) {
      bar();
    }
  }

  /*Does bar*/
  private void bar() {}
}
```

```
class X {
  int i;
  public void foo() {
    if (i > 5 /* max ... ) {
      bar();
    }
  }

  /*Does bar*/
                  ) {}}
}
```

Figure 3.2 The unclosed block comment error `/* max ...` is only detected at the `private` keyword (left fragment). The late detection of the error results in a poor repair near the failure location (right fragment).

used to constrain the search space used by the backtracking technique for applying recovery rules. By constraining the recovery suggestions to a particular part of the file, *region selection* improves the efficiency as well as the quality of the recovery, avoiding suggestions that are spread out all over the file.

In some cases it is 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. As a second application of the regional approach, *region skipping* is used as a fallback recovery strategy that discards the erroneous region entirely in case a detailed analysis of the region does not lead to a satisfactory recovery.

3.4 LAYOUT-SENSITIVE REGION SELECTION

In this section we describe a regional recovery technique that uses layout to select regions of code that enclose syntax errors. The selected regions can be analyzed in detail by a correcting technique, or discarded if no correction is found within an acceptable time span. The technique is language independent and can be implemented for different parsing formalisms.

3.4.1 *Nested Structures*

Language constructs such as statements and methods are elements of list structures. List elements form free standing blocks, in the sense that they can be omitted without influencing the syntactic interpretation of other blocks. It follows that erroneous free standing blocks can simply be skipped, providing a coarse recovery that allows the parser to continue. We conclude that these blocks form suitable regions for regional error recovery.

Indentation typically follows the logical nesting structure of a program, as illustrated in Figure 3.3. The relation between constructs can be deduced from the layout. An indentation shift to the right indicates a *parent-child* relation, whereas the same indentation indicates a *sibling* relation. The region selection technique inspects the parent and sibling structures near the parse failure location to detect the erroneous region.

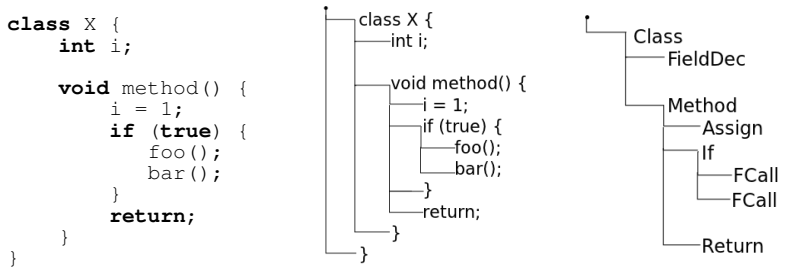


Figure 3.3 Indentation closely resembles the hierarchical structure of a program.

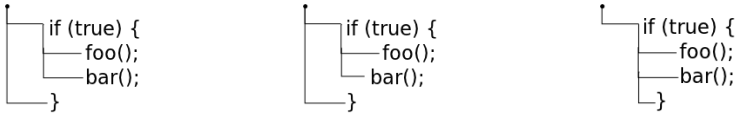


Figure 3.4 Parent child relations between lines with consistent layout (left) and inconsistent layout (mid, right). `if(true){` is the parent line of the siblings `foo();` and `bar();` (left, mid, right), and the inconsistently indented `}` (right).

Indentation usage is not enforced by the language definition. Proper use of layout is a convention, being part of good coding practice. We generally assume that most programmers apply layout conventions, which is reinforced by the application of automatic formatters. Furthermore we assume that indentation follows the logical nesting structure. However, we should keep in mind the possibility of inconsistent indentation usage which decreases the quality of the results. The second assumption we make is that programs contain free standing blocks, i.e., that skipping a region still yields a valid program. Most programming languages seem to meet this assumption. If both assumptions are met, layout-sensitive region selection can improve the quality and performance of a correcting technique, and offer a fallback recovery technique in case the correcting technique fails.

3.4.2 *Indentation-based Partitioning*

We view the source text as a tree-structured collection of lines, whereby the parent-child relation between lines are determined by indentation shifts. Thus, given a line l, line p is the *parent* of l if and only if l is strictly more indented than p, and line l succeeds line p, and no lines exist between l and p that have less indentation than l. Lines with the same parent are *siblings* of each other. Figure 3.4 illustrates the parent-child relation for some small code fragments. The line `if(true){` in the left fragment is the parent of the sibling lines `foo();` and `bar();`. The mid and right fragment illustrate how the parent-child relation applies in case of inconsistent indentation; by definition, child lines are more indented than their parent, however, the sibling lines in these fragments do not all have the same indent value.

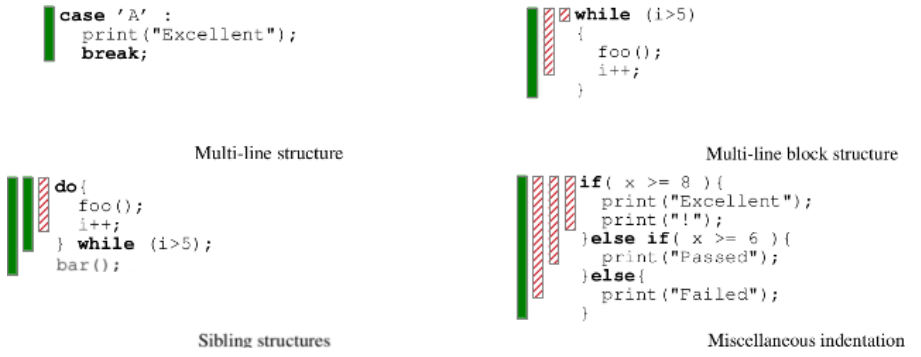


Figure 3.5 Multi-line Java constructs with various indentation patterns. The solid bars indicate layout regions that correspond to code regions, the hatched bars indicate layout regions that are in fact unwished artifacts.

A parent-child relation between two lines is a strong indication that the code constructs associated to these lines are also in parent-child relation. Similarly, a sibling relation between two lines indicates that either their associated code constructs are siblings as well, or that both lines belong to the same multi-line construct. Figure 3.5 provides some examples of multi-line constructs with various indentation patterns. For all constructs in the figure it holds that a parent-child relation between two lines reflects a parent-child relation between the code constructs associated to these lines. The shown constructs are different with respect to the number of siblings (of the first line) that are part of the construct. Another type of multi-line constructs are constructs that wrap over to the subsequent, more indented line. In that case, a parent child relation exists between two lines that actually belong to the same construct. This is an example of a small inconsistency that is not harmful to the overall approach.

We decompose a code fragment into candidate regions, based on the assumption that parent-child relations between lines reflect parent-child relations between the associated constructs, e.g., if a line is contained in a region then its child lines are also contained in that region. Unfortunately, indentation alone does not provide sufficient information to demarcate regions exactly. The main limitation is the ambiguous interpretation of sibling lines, which, by assumption, either belong to the same code construct or to separate constructs that are siblings. Given a single line, we construct multiple indentation-based regions: the smallest region consist of the line plus its child lines, the alternate regions are obtained by subsequently including sibling lines, including their children.

The bars in Figure 3.5 show the different regions that are constructed for the first line of the given fragments. Only the regions corresponding to the solid bars represent actual code constructs or (sub)lists of code constructs. The other bars are unwanted artifacts that, based on indentation alone, cannot be distinguished from real regions. Notice that most of these ambiguities could

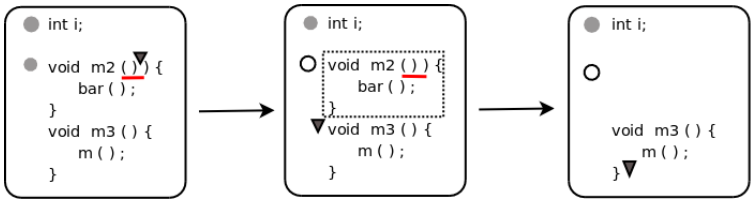


Figure 3.6 A candidate region is validated and successfully discarded.

be solved by using language-specific information, for example about the use of curly braces in Java; lines that start with a curly brace are most likely to be part of the region being constructed. An interesting approach would be to extract this type of information from the grammar so that it can be used in the region detection heuristic. However, in the current implementation we opted for a language-independent algorithm.

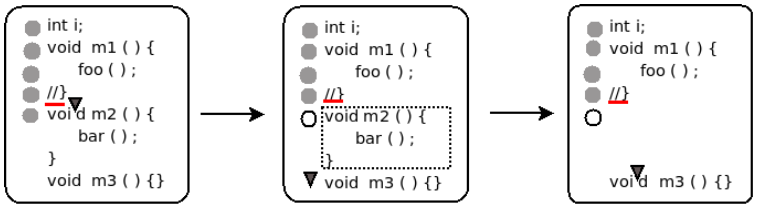
3.4.3 *Region Selection*

We follow an iterative process to select an appropriate region that encloses a syntax error. In each iteration, a different *candidate region* is considered. This candidate is then *validated* and either accepted as erroneous or rejected; in case of a rejected candidate, another candidate is considered.

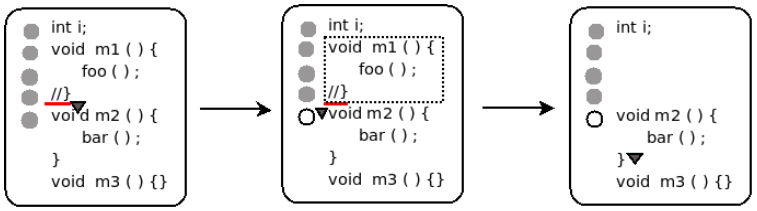
The selection of candidate regions faces two challenges: First, the start line of the erroneous code construct is not known, second, multiple unsuitable regions are constructed because of the ambiguous interpretation of sibling lines. We adopt a pragmatic approach, subsequently selecting candidate regions for a different start line location with a different number of sibling lines. We start with validating small regions near the failure location, then we continue with validating regions of increased size as well as regions that are located further away from the failure location. More details are provided in Section 3.4.4 that describes the region selection algorithm.

A region is validated as erroneous in case discarding of that region solves the syntax error, e.g., parsing continues after the original failure location. The region validation criterion should balance the risk of evaluating a syntactically correct candidate region as erroneous, and the risk of evaluating an erroneous candidate region as syntactically correct. Both cases lead to large regions and/or spurious syntax errors, which should be avoided. The underlying problem are multiple errors; it is not possible to distinguish a secondary parse failure from a genuine syntax error that happens to be close-by. We address the issue of multiple syntax errors by implementing a heuristic accept criterion. The criterion considers a candidate region as erroneous if discarding results in two more lines of code parsed correctly. The criterion is established after some experimentation and has shown good practical results.

We show example scenarios in Figure 3.6 and 3.7. The underlined text reveals the syntax errors, the boxed fragments are the selected candidate re-



(a) A candidate region is rejected.



(b) An alternative candidate region is validated and successfully discarded.

Figure 3.7 Iterative search for a valid region.

gions, the triangles indicate the location of the parser, while the circles in the left marge represent choice points that store the parser configuration at that location. The choice points are used for backtracking to a previous configuration. Figure 3.6 shows a syntax error and the point of detection, indicated by a triangle (left). A candidate region is selected based on the alignment of the `void` keyword and the closing bracket (middle figure), and validated by discarding the region. Since the parsing of the remainder of the fragment is successful (right), the region is accepted as erroneous. Figure 3.7a shows an example where a candidate region is rejected. Based on the point of detection, an obvious candidate region is the `m2` method (middle), which is discarded (right). However, the attempt to parse the succeeding construct leads to a premature parse failure (right), therefore the region is rejected. In Figure 3.7b an alternative candidate region is selected. This region is validated as erroneous since discarding solves the parse error.

3.4.4 *Algorithm*

Figure 3.8 illustrates the region selection procedure applied to a small code fragment with a parse failure at the marked line. The vertical bars represent the regions that are subsequently visited by increasing the backtracking distance (`backwardIndex`) and the region size (`siblingCount`). The right most bar represents the parent region visited in a recursion step. We will now discuss the implementation of the algorithm in more detail.

Figure 3.9 shows the region selection algorithm in pseudo-code. The function `SelectErroneousRegion` takes as input the failure line and returns as output the erroneous region described by its start line and end line. The

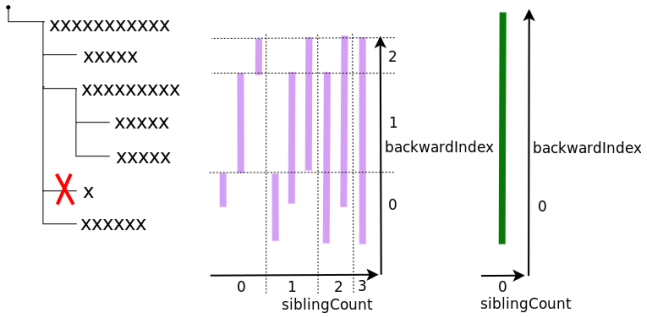


Figure 3.8 Candidate regions subsequently tested for the indented code fragment at the left. Candidate regions are selected by backtracking (*backwardIndex*) and by extending the number of sibling lines that are contained in the region (*siblingCount*). Finally, the parent line is visited in the recursion step.

nested `for` loops (line 4, 5) implement the iterative search process described in Section 3.4.3. The iteration starts with the smallest region (`siblingCount=0`) that can be constructed for the failure line (`backwardIndex=0`). In the first iteration (line 5), regions are selected that start at increasing distance from the failure location. The second iteration (line 4) increases the size of the selected regions. The iteration stops in case a selected region is validated as erroneous (lines 9-11). If no erroneous region is found, the search process continues by recursively visiting the parent of the failure line (line 14). For performance reasons, we restrict the maximum size of the visited regions (line 4) and the maximum number of backtracked lines (line 5). Good practical results were obtained with a maximum size of 5 sibling lines and 5 backtracking steps.

3.4.5 *Practical Considerations*

Tabs and spaces A practical concern for parsing techniques that take indentation into account is the mixed use of tabs and spaces. The indentation level in this case depends on the tab width, i.e., the mapping from tab characters to space characters which may be configurable in the editor. Currently, our technique assumes a fixed tab width of 4 space characters. An obvious improvement would be to make the tab width configurable and dependent on the editor setting when used in an interactive environment.

Separators and operators Region selection works for structures that form free standing blocks in the grammar, e.g., list elements and optional elements such as the `else` block in an `if-else` statement. A practical consideration are separators and operators that may reside between language constructs. For example, the constructs `FAILED` and `score <= 8` in the Java fragment of Figure 3.10 can only be discarded if the separator (`,`), respectively the operator (`&&`) that connects these constructs with their preceding constructs are discarded as well. To address this issue, we have extended the region selection schema

```
SelectErroneousRegion(failureLine)
 1  ▷ Input: Line where the parse failure occurs (or a parent of this line)
 2  ▷ Output: Region that contains the error
 3
 4  for siblingCount in 0 to MAX_NUMBER_OF_SIBLING_LINES
 5      for backwardIndex in 0 to MAX_BACKWARD_INDEX
 6          startLine ← GetPrecedingSibLine(failureLine, backwardIndex)
 7          siblingLine ← GetFollowingSibLine(startLine, siblingCount)
 8          endLine ← GetLastDescendantLine(siblingLine)
 9          if TrySkipRegion(startLine, endLine) then
10              return (startLine, endLine) ▷ erroneous region
11          end
12      end
13  end
14  return SelectErroneousRegion(GetParentLine(failureLine))

TrySkipRegion(startline, endline)
17  ▷ Output: true iff discarding the region startline ... endline
      lets parsing continue after the failure location

GetPrecedingSibLine(line, bwCount)
18  ▷ Output: Sibling line that preceeds line by bwCount siblings

GetFollowingSibLine(line, fwCount)
19  ▷ Output: Sibling line that succeeds line by fwCount siblings

GetLastDescendantLine(line)
20  ▷ Output: Last descendant line of line, or line if no descendants exist

GetParentLine(line)
21  ▷ Output: Parent line of line
```

Figure 3.9 Algorithm to select a discardable region that contains the syntax error.

with a candidate region consisting of the original region plus the lexical token at the end of the preceding sibling line.

Multi-line comments and strings The selection procedure can generally select erroneous regions that are located at a reasonable distance from the failure location. However, if the distance between the error and the failure location is too large, the region selection schema fails to locate the error. A particularly problematic case commonly seen in practice are unclosed flat structures such as block comments or multi-line strings. Due to the liberal nature of these structures, a missing closing symbol typically causes parse failures far from

```
public enum Grade {
  EXCELLENT ,
  PASSED ,
  FAILED
}

Grade getGrade(){
  ...
  if(
    6 <= score &&
    score <= 8
  ) return Grade.PASSED;
  ...
}
```

Figure 3.10 Separators and operators must be included in the candidate region of the adjacent construct.

```
class X {
  int i;
  public void foo() {
    if (i > 5 /* max ...  ){
      bar();
    }
  }
  ...
  ...
  /*Does bar*/
  private void bar() {}
}
```

Figure 3.11 Unclosed block comments are generally detected at a large distance of the error location, which makes it challenging to detect the erroneous region.

the actual location of the error. An example is shown in Figure 3.11 where the parser fails at the `private` keyword.

We describe how we address this issue for SGLR, which is a scannerless generalized LR parser, e.g., it does not employ a separate scanner for tokenization. After the opening of the block comment (`/*`), the parser accepts all characters until the block comment is ended (`*/`) or the end of the file is reached. As a consequence, a missing block comment ending is typically detected at a large distance from the error location. The stack structure of the parser in these scenarios is characterized by a reduction that involves many characters starting from the characters that open the flat construct (`/*`). If this stack structure is recognized, a candidate region is selected from the start of the reduction, making it possible to cope with flat multi-line structures such as block comments for which errors may cause a parse failure far from the actual error location.

3.5 EVALUATION

In this section we evaluate the techniques discussed in 3.3 and Section 3.4, together and in isolation. The evaluation has the objective to select the optimal combination of recovery techniques. The results shown in this section should be considered as preliminary; a more extensive evaluation is provided in Chapter 4, Section 4.5, which evaluates the effectiveness of the selected combination of techniques for different scenarios. That is, we apply the combined technique to multiple languages, compare the approach to the quality standard set by the JDT parser, and measure the performance on files of different sizes, with and without syntax errors.

3.5.1 *Experimental Setup*

We focus our evaluation on the Stratego-Java language, which is a complex language composed from Stratego and Java. We choose the Stratego-Java language since composed languages form an important use case for scannerless generalized parsing. Our extended evaluation in Chapter 4 covers experiments that involve other languages as well.

We follow the evaluation method proposed in Chapter 4. First, we generate a large set of test inputs from a small set of correct base files. The generated test inputs are modified versions of the base files, representing typical editing scenarios as identified in an empirical study on editing behavior. Together with each test input, an oracle interpretation is automatically generated that represents the intended interpretation of the test input. In total, we generated 158 test cases from 5 correct base files. The base files are taken from the *Dryad compiler*, an open compiler for the Java platform (Kats et al., 2008), and the *WebDSL compiler*, a compiler that generates Java code for applications written in WebDSL (Groenewegen et al., 2008).

To measure the quality of a recovery, we calculate the tree edit distance (Chawathe et al., 1996) between the recovered interpretation and the intended interpretation. The scales for the figures we show are calibrated such that they roughly correspond to the human assesment criteria proposed by Pennello and DeRemer (1978). That is, "no diff" corresponds to the *excellent* qualification, a "small diff" (1–25 tree edits) roughly corresponds to the *good* qualification, and a "large diff" (26+ tree edits) approximately corresponds to the *poor* qualification. Figure 4.14 in Chapter 4, Section 4.4.3 illustrates how these thresholds were determined by plotting measured diff values against their corresponding values obtained after human judgement.

To measure the performance, we compute the extra time it takes to recover from one or more errors (the recovery time), by subtracting the parse time of the oracle file from the parse time of the erroneous file. For all performance measures included in this section, an average, collected after three runs, is used. We set a time limit of 5 seconds to cut off recoveries that take an (almost) infinite time to complete. All measuring is done on a "pre-heated" JVM running on a laptop with an Intel(R) Core(TM) 2 Duo CPU P8600, 2.40GHz processor, 4 GB Memory.

3.5.2 *Comparing Different Combinations of Techniques*

In this experiment, we focus on selecting the best parser configuration combining the different recovery techniques presented in this chapter: the permissive grammars and backtracking approach of Section 2.4 (PG), and the region selection technique of Section 3.4 (RS), which can be applied as a fall back recovery technique (RR) by skipping the selected region. We apply the permissive grammar technique (PG) using the WC recovery rule set that was selected as the optimal rule set in a previous experiment described in Sec-

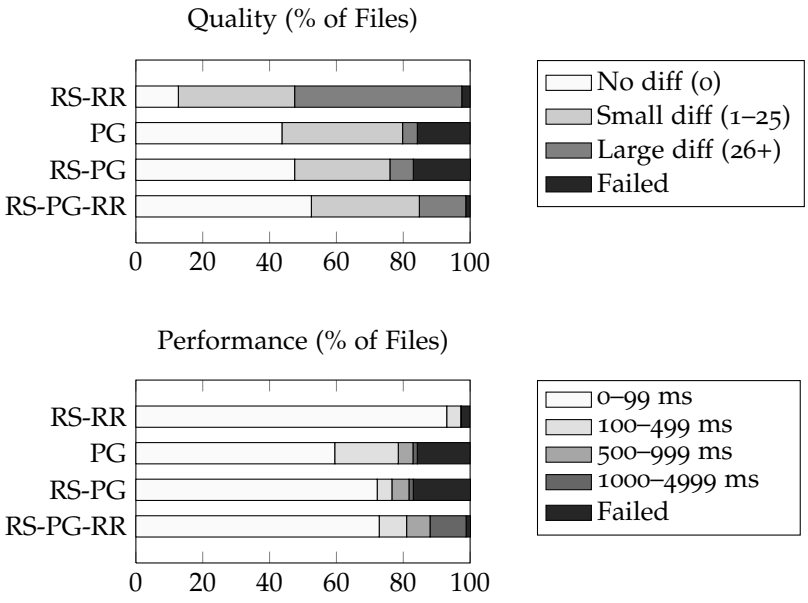


Figure 3.12 Quality and performance (recovery times) using combinations of techniques for Stratego-Java. **RR** - Region selection and recovery, **PG** - Permissive grammars, **RS** - Region selection.

tion 2.6.2. We set a time limit of 1 second for applying recovery rules (PG), when this technique is applied in combination with the fallback strategy (RR).

We first applied the techniques in isolation; regional recovery by skipping regions (RS-RR), and parsing with permissive grammars (PG). We then evaluate the approaches together; first parsing with permissive grammars applied to a selected region (RS-PG), then adding region recovery (RR) as a fallback recovery technique (RS-PG-RR). The results from the experiment are shown in Figure 3.12.

Figure 3.12 (Performance) shows the performance results for the different combinations of techniques. The results show that region recovery (RS-RR) gives good performance in all cases, and that region selection (RS) positively affects the performance of the permissive grammar technique (RS-PG versus PG). Since all techniques give reasonable performance, we focus on quality to find the best combination of techniques.

Considering the Quality part of Figure 3.12 and the results of PG, we see that it has a relatively large number of failed recoveries (17%), but regardless of this fact it still leads to reasonable recoveries (small diffs) in the majority of cases (80%). For regional recovery (RS-RR), the situation is exactly the opposite. As expected, skipping a whole region in most cases does not lead to the optimal recovery. However, the skipping technique does provide a robust mechanism, leading to a successful parse in most cases (97%). Restricting PG to a selected erroneous region (RS-PG) only has a marginal effect on the

recover quality and performance. However, combining all techniques (RS-PG-RR), improves the robustness (99%), as well as the precision (85% small or no diff) compared to both individual techniques.

Based on our evaluation we conclude that the combined approach of RS-PG-RR is the most effective configuration. We revisit the evaluation of this combined approach and provide a further discussion in Section 4.5.

3.6 RELATED WORK

In Chapter 2 we introduced error recovery for SGLR, based on recover productions that can be automatically derived from a grammar. The present chapter refines this work, improving the quality and performance of the technique by constraining the application of recovery rules to coarse-grained regions that were identified as erroneous. The coarse-grained regions can be discarded as a fallback recovery in case the application of recovery rules does not lead to a valid parse. The region detection technique uses indentation to locate parse errors in the source text.

Error recovery for LR The problem of handling syntax errors during parsing has been widely studied (Lévy, 1971; Mauney and Fischer, 1988; Pai and Kieburtz, 1980; Barnard and Holt, 1982; Tai, 1978; Fischer et al., 1980; Degano and Priami, 1995; McKenzie et al., 1995; Corchuelo et al., 2002). We focus on LR parsing for which there are several different error recovery techniques (Degano and Priami, 1995). These techniques can be divided into *correcting* and *non-correcting* techniques.

The most common non-correcting technique is *panic mode*. On detection of an error, the input is discarded until a synchronization token is reached. When a synchronizing token is reached, states are popped from the stack until the state at the top enables the resumption of the parsing process. Panic mode does not provide a proper diagnosis of the error and may skip large fragments of an input. Our layout-sensitive regional recovery algorithm can be used in a similar fashion, but selects discardable regions based on indentation. The advantage of our technique is that it is language independent and that it respects the grammatical structure of the intended program expressed by the use of indentation.

Successful recovery mechanisms often combine more than one technique (Degano and Priami, 1995). For example, panic mode is often used as a fall back method if correction attempts fail. Burke and Fisher (1987) present a correcting method based on three phases of recovery. The first phase looks for simple correction by the insertion or deletion of a single token. If this does not lead to a recovery, one or more open scopes are closed. The last phase consists of discarding tokens that surround the parse failure location. We implemented similar phases in our approach, however, in our work we take indentation into account, for the secondary region skip recovery technique. In addition, by starting with region selection, the performance as well as the quality of our correcting technique, i.e., the permissive grammar approach of Chapter 2, is improved.

Regional recovery Regional error recovery methods (Lévy, 1971; Mauney and Fischer, 1988; Pai and Kieburtz, 1980; Barnard and Holt, 1982) select a region that encloses the point of detection of an error. Typically, these regions are selected based on nearby marker tokens (also called fiducial tokens (Pai and Kieburtz, 1980), or synchronizing symbols (Barnard and Holt, 1982)), which are language-dependent. In our approach, we assign regions based on layout instead. Layout-sensitive regional recovery requires no language-specific configuration, which makes the technique effective for a variety of languages. Similar to the fiducial tokens approach, it depends on the assumption that languages have recognizable (token or layout) structures that serve for the identification of regions.

Barnard and Holt (1982) present an hierarchic error repair approach using *phases* corresponding to lists of lines. For instance, a phase may be a set of declarations that must appear together. These phases are similar to our regions, with the difference that , in our case, the regions are constructed based on indentation. Both approaches have some kind of local repair within phases or regions, and may skip parts of the input.

Indentation-based recovery The approach of using layout information for partitioning files has been inspired by the technique of bridge parsing (Nilsson-Nyman et al., 2009). Bridge parsing is a supplementary recovery technique that uses indentation to insert missing scope closing tokens. In contrast to our technique, bridge parsing is a correcting technique that is limited to the recovery of unclosed scope structures.

3.7 CONCLUSION

In Chapter 2 we identified two open issues with our recovery technique for SGLR based on grammar relaxation. The first is the quality of corrections, which is sometimes lacking since a global, linguistic solution is not aware of the structure of the program expressed by the layout. The second is the performance of the recover algorithm; for some problematic cases an adequate recovery solution cannot be find in an acceptable timespan.

In this chapter, we addressed both issues by introducing a region selection technique that can be combined with the permissive grammar technique. Source code has a hierarchical structure that is reflected in the use of indentation. We have shown that this property can be exploited to partition files and detect erroneous regions; these regions can be analyzed in detail by a correcting technique, or discarded as a fallback recovery strategy.

We evaluated our approach by comparing different combinations of techniques. The evaluation shows that constraining the search space for recovery rule applications to erroneous regions improves the quality and performance of the permissive grammar approach discussed in Chapter 2. Furthermore, discarding erroneous regions as a fallback recovery helps to cope with pathological cases not easily addressed with only permissive grammars and backtracking.

Automated Evaluation of Parse Error Recovery Techniques

4

ABSTRACT

Evaluation of parse error recovery techniques is an open problem, since objective standards and methods to measure the quality of recovery results are missing. This chapter proposes an automated technique for recovery evaluation that offers a solution for two main problems in this area. First, a representative test set is generated by a mutation based fuzzing technique that applies knowledge about common syntax errors. Secondly, the quality of the recovery results is automatically measured using an oracle-based evaluation technique.

We apply the evaluation technique to evaluate the quality and performance of the recovery approach for SGLR that we presented in Chapter 2 and 3. The evaluation results show that the approach works for different languages, that the technique is scalable with respect to performance, and that the recovery quality is on par with the standard set by the JDT parser.

4.1 INTRODUCTION

Integrated development environments (IDEs) increase programmer productivity by combining generic language development tools with services tailored for a specific language. Language-specific services require as input a structured representation of the source code in the form of an abstract syntax tree (AST) constructed by the parser.

To provide rapid syntactic and semantic feedback, IDEs interactively parse programs as they are edited. The parser runs in the background with each key press or after a small delay passes. As the user edits a program, it is often in a syntactically invalid state. Parse error recovery techniques can diagnose and report parse errors, and can construct ASTs for syntactically invalid programs (Degano and Priami, 1995). Thus, to successfully apply a parser in an interactive setting, proper parse error recovery is essential.

Evaluation of error recovery techniques is an open problem in the area of parsing technology. An objective and automated evaluation method is essential to do benchmark comparisons between existing techniques, and to detect regression in recovery quality due to adaptations of the parser implementation. Currently, objective standards and methods for performing thorough evaluations are not yet defined. We identified two challenges: first, the recovery technique must be evaluated against a representative set of test inputs, secondly, the recovery outputs must be automatically evaluated against

a quality metric. The aim of this chapter is to provide an automated method for evaluating error-recovery techniques; and to apply this method to evaluate the recovery technique described in Chapter 2 and 3.

Test data The first challenge for recovery evaluation is to obtain a representative test suite. Evaluations in the literature often use manually constructed test suites based on assumptions about which kind of errors are the most common (Horning, 1976; Kats et al., 2009a; Nilsson-Nyman et al., 2009). The lack of empirical evidence for these assumptions raises the question how representative the test cases are, and how well the technique works in general. Furthermore, manually constructed test suites tend to be biased because in many cases the same assumptions about edit behavior are used in the recovery algorithm as well as in the test set that measures its quality. Many test inputs need to be constructed to obtain a test set that is statistically significant. Thus, manual construction of test inputs is a tedious task that easily introduces a selection bias.

A better option for obtaining representative test data is to construct a test suite based on collected data, an approach which is taken in (Pennello and DeRemer, 1978) and (Ripley and Druseikis, 1978). However, collecting empirical data requires administration effort and may be impossible for new languages that are not used in practice yet. Furthermore, empirical data does not easily provide insight into what types of syntax errors are evaluated. That is, the collected syntax errors must be categorized manually to evaluate how the recovery algorithm performs on different types of syntax errors. A final problem is automated evaluation of the quality of the parser output, which is not easily implemented on collected data.

As an alternative to collecting edit scenarios in practice, this chapter investigates the idea of generating edit scenarios. The core of our technique is a general framework for iterative generation of syntactically incorrect files. To ensure that the generated files are realistic program files, the generator uses a mutation based fuzzing technique that generates a large set of erroneous files from a small set of correct input files taken from real projects. To ensure that the generated errors are realistic, the generator implements knowledge about editing behavior of real users, which was retrieved from an empirical study. The edit behavior is implemented by *error generation rules* that specify how to construct an erroneous fragment from a syntactically correct fragment. The implicit assumption is that these rules implement edit behavior that is generic for different languages. The generation framework provides extension points were compiler testers can hook in custom error generation rules to test the recovery of syntax errors that are specific for a given language.

Quality measurement The second challenge for recovery evaluation is to provide a systematic method to measure the quality of the parser output, e.g., the recovered AST which represents a speculative interpretation of the program.

Human judgement is decisive with respect to the quality of recovery results. For this reason, Pennello and DeRemer (Pennello and DeRemer, 1978) introduce human criteria to categorize recovery results. A recovery is rated

excellent if it is the one a human reader would make, *good* if it results in a reasonable program without spurious or missed errors, and *poor* if it introduces spurious errors or if excessive token deletion occurs. The Pennello and DeRemer criteria represent the state of the art evaluation method for syntactic error recovery applied in, amongst others, (Pennello and DeRemer, 1978; Pai and Kieburtz, 1980; Degano and Priami, 1995; Corchuelo et al., 2002). Though human criteria most accurately measure recovery quality, application of these criteria requires manual inspection of the parse results which makes the evaluation subjective and inapplicable in an automated setting. Another disadvantage is the lack of precision, only three quality criteria are distinguished.

Oracle-based approaches form an alternative to manual inspection. First, the intended program is constructed manually. Then, the recovered program is compared to the intended program using a diff based metric on either the ASTs or the textual representations obtained after pretty printing. An oracle-based evaluation method is applied in (de Jonge et al., 2009) and (Nilsson-Nyman et al., 2009). The former uses textual diffs on pretty-printed ASTs, while the latter uses tree alignment distance (Jiang et al., 1994) as a metric of how close a recovered interpretation is to the intended interpretation of a program.

A problem with these approaches is the limited automation. Differential oracle approaches allow automated evaluation, but the intended files must be specified manually which requires considerable effort for large test suites. Furthermore, the intended recovery may be specified after inspecting the recovery suggestion in the editor, which causes a bias towards the technique implemented in the editor.

To address the concern of automation, we extend the error generator so that it generates erroneous files together with their oracle interpretations. The oracle interpretations follow the interpretation of the base file, except for the affected code structures. For these structures, an *oracle generation rule* is implemented that specifies how the intended interpretation is constructed from the original structure and the erroneous syntax elements. By default, the original structure itself is taken as an oracle interpretation; which works well in case only literal tokens such as delimiters and operators are modified by the error generation rule. Compiler testers can overwrite this behavior by specifying a custom oracle generation rule. Custom oracle rules can handle cases in which the original structure is not applicable as an oracle interpretation for the erroneous fragment; for example because non-literal tokens such as identifiers or numbers are removed from the original syntax. We demonstrate an oracle rule that can repair an incomplete syntactic structure by inserting missing syntax elements, and an oracle rule that can select an appropriate sub construct as a replacement for the broken original construct.

The remaining problem is to define a suitable metric between recovered programs and their intended oracle programs. We compared four differential oracle metrics based on their accuracy and on qualitative aspects such as applicability and comprehensibility. We concluded that all evaluated metrics accurately reflect recovery quality. For practical reasons we chose tree-edit

distance as the preferred metric that we use in our recovery evaluations.

Application We applied the automated evaluation method to study the quality and performance of the error recovery technique described in Chapter 2 and 3. The evaluation shows that the technique works for different grammars, that it has a low performance overhead, and that it provides good or excellent recovery quality in the majority of cases.

Contributions This chapter provides the following contributions:

- A preliminary statistical study on syntax errors that occur during interactive editing.
- A test input generation technique that generates realistic edit scenarios specified by error generation rules.
- A fully automatic technique to measure the quality of recovery results for generated edit scenarios.
- A comparison of different metrics for recover quality measurement.
- An extensive evaluation of the recovery technique that we implemented for SGLR.

We start this chapter with a statistical analysis of syntax error that occur during interactive editing (Section 4.2). The input generation technique is discussed in Section 4.3, while automatic quality measurement is the topic of Section 4.4. Section 4.5 covers the experiments we did to evaluate the quality and performance of the recovery approach that we implemented for SGLR parsing.

4.2 UNDERSTANDING EDIT BEHAVIOR

We did an empirical study to gain insight into edit scenarios that occur during interactive editing. The final objective was to implement these scenarios in an error generation tool used for the generation of test inputs for error recovery. With this objective in mind, we focus on the following research questions:

- How are syntax errors distributed in the file? Do multiple errors occur in clusters or in isolation? What is the size of the fragments that contain clustered syntax errors?
- What kinds of errors occur during interactive editing? Can we classify these errors in general categories?

4.2.1 *Experimental Design*

For the analysis of syntax errors we examined edit data for three different languages: Stratego (Bravenboer et al., 2008), a transformation language used for code generation and program analyses; SDF (Luttik and Visser, 1997),

a declarative syntax definition language; and WebDSL (Groenewegen et al., 2008), a domain-specific language for web development. We chose these languages since they are considerably different from each other, covering important characteristics of respectively transformation, declarative and imperative languages.

To study edit behavior in Stratego, we collected edit scenarios from two groups; a group of students that took part in a graduate course on compiler construction and a group of researchers that work in this field. Contrary to the researchers, the students had only limited experience with the Stratego language. In total, we collected 43,984 Stratego files (program snapshots) from 22 students and 13,427 Stratego files from 5 experienced researchers. To evaluate the impact of familiarity with a language on the syntactic errors being made, the data was examined from the two groups separately. It turned out that the unfamiliarity with the language had only a minor effect on the edit behavior we observed. For WebDSL and SDF we did not have a large user group, we analysed respectively 936 WebDSL files from 2 programmers and 103 SDF files from 3 programmers.

The edit data was collected during interactive editing in an Eclipse based IDE. The support offered by the IDE included main editor services such as syntax highlighting, content completion, automatic bracket completion, auto-indent, and syntax error recovery and reporting. The edit scenarios were created by copying the file being edited, capturing snapshots between individual keystrokes applied by the programmer. We filtered structurally equivalent scenarios that only differ at the character level.

4.2.2 *Distribution of Syntax Errors*

We did a statistical analysis of the collected edit data to obtain knowledge about how syntax errors are distributed in a file. To process the large amount of collected data, we looked for an automated method. In general, it is difficult to automatically analyze the distribution of syntax errors, since this requires a technique to locate syntax errors in the source code; which is one of the main challenges for parse error recovery techniques. To overcome this difficulty, we locate syntax errors in the source code by identifying *edit fragments*. Edit sequences consist of syntactically correct files and syntactically incorrect files that are interleaved. We presume that all syntax errors are located in the edit fragments, i.e., the code fragments that are modified in between two subsequent correct files.

To gain insight into the distribution of errors in a file, we analyzed edit fragments for the scenarios we collected for Stratego, SDF and WebDSL. To identify these fragments, we applied a textual diff algorithm on the edit scenarios and their preceding correct scenario. We counted the number of edited fragments per file, and we measured the size of these fragments in terms of lines of code. The results are discussed below.

Unrelated syntax errors occur when programmers continue editing the next fragment while the previous edited fragment still contains syntax errors. We

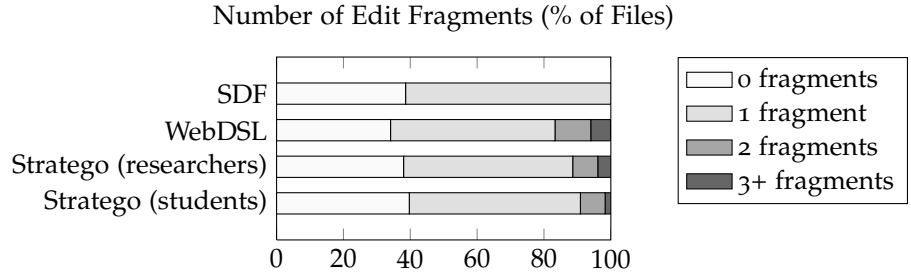


Figure 4.1 Number of edited fragments per file.

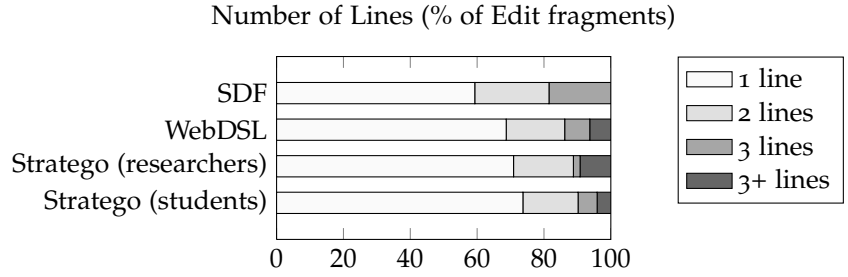


Figure 4.2 Number of lines per fragment being edited.

analysed the possible occurrence of unrelated errors by counting the number of edited fragments per file in the collected scenarios. The results in Figure 4.1 show that programmers typically edit code fragments one by one, solving all syntax errors before editing the next fragment. We conclude that unrelated errors only occur in a minority of cases.

From the previous discussion we conclude that syntax errors are typically clustered in relatively small fragments. The logical follow-up question is: what are the sizes of these fragments? We measured the size of the edited fragments, Figure 4.2 shows the result. The figure shows that in most cases the fragments are small, consisting of only one or two lines of code.

4.2.3 *Classification of Syntax Errors*

We manually inspected erroneous files to gain insight into the kind of errors that occur in the fragment being edited. We inspected 50 randomly selected files for each test group (Stratego students, Stratego researchers, SDF and WebDSL). The results are summarized in Figure 4.3. The categories are explained below. From Figure 4.3 we conclude that most errors are not language specific, and that incomplete constructs and random errors are the most frequently occurring syntax errors.

- *Incomplete constructs*, language constructs that miss one or more symbols at the suffix, e.g. an incomplete `for` loop as in `for (x = 1; x`.

	Stratego S.	Stratego R.	SDF	WebDSL
Incomplete Construct	24	18	22	28
Random errors	17	22	8	11
Scope errors	0	0	0	1
String Literal	3	3	18	3
Comments	2	4	2	4
Large Region	2	3	0	3
Language-specific	2	0	0	0

Figure 4.3 Classification of edit scenarios for different languages. The numbers indicate the number of errors that fall in a given category for a given test group.

- *Random errors*, constructs that contain one or more token errors, e.g., *missing*, *incorrect* or *superfluous* symbols.
- *Scope errors*, constructs with missing or superfluous scope opening or closing symbols.
- *String or comment errors*, block comments or string literals that are not properly closed, e.g., `/*...*`.
- *Large erroneous regions*, severely incorrect code fragments that cover multiple lines.
- *Language specific errors*, errors that are specific for a particular language.

4.3 GENERATION OF SYNTAX ERRORS

To test the quality of an error recovery technique, parser developers can manually write erroneous input programs containing different kind of syntax errors. However, to draw a conclusion that is statistically significant, the developer must extend the test set so that it becomes sufficiently large and diversified. Variation points are: the error type, the construct that is broken, the syntactic context of the broken construct, and the layout of the input file nearby the syntax error.

It is quite tedious to manually write a large number of invalid input programs that cover various instances of a specific error type. As an alternative, our error generation framework allows the tester to write a generator that automates the creation of test files that contain syntax errors of the given type. Error generators are composed from error generation rules and error seeding strategies. The error generation rules specify how to construct an erroneous fragment from a syntactically correct fragment; the error seeding strategies control the application of the error generation rules, e.g., they select the code constructs that will be broken in the generated test files. A generator for files with multiple errors can be defined by combining generators for single error files.

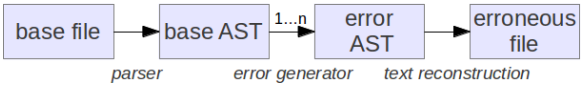


Figure 4.4 Error generators specify how to generate multiple erroneous files from a single base file.

Figure 4.4 shows the work flow for test case generation implemented by the generation framework. First, the parser constructs the AST of the base file. Then the generator is applied which constructs a large set of ASTs that represent syntactically erroneous variations of the base program. Finally, the texts of the test files are reconstructed from these ASTs by a text reconstruction algorithm that preserves the original layout (de Jonge and Visser, 2012a). Error generation rules may by accident generate modified code fragments that are in fact syntactically correct. As a sanity check, all generated files that are accidentally correct are filtered out by parsing them with error recovery turned off.

4.3.1 *Error Generation Rules*

Error generation rules are applied to transform abstract syntax terms into string fragments that represent syntactically erroneous constructs. The error generation rules operate on the concrete and abstract syntax elements that are associated to the abstract term, i.e., its abstract child terms plus the associated literals and layout tokens. An example is given in Figure 4.5, syntax elements. Applying modifications to a list of concrete and abstract child elements rather than a list of tokens or characters offers refined control over the effect of the rule. For example, error generation rules can specify insert, delete or update operations on the literals associated to a code construct, without affecting the literals in its child constructs. If necessary, the abstract child constructs can be further expanded to their associated syntax elements. As an example, Figure 4.5 illustrates the application of an error generation rule on an `if` construct. The given rule removes the closing bracket of the `if` condition from the list of syntax elements.

The generation framework predefines a set of primitive generation rules that form the building blocks for more complex rules. Primitive error generation rules introduce simple errors by applying insertion, deletion, or replacement operations on the syntax elements associated with a code structure. For example, the following generation rule drops the last syntax element of a code construct.

```
DROPLASTELEMENT(syntaxElements)
1  if LENGTH(syntaxElements) > 0 then
2      syntaxElements ← remove element at LENGTH(syntaxElements) − 1
3  return syntaxElements
```

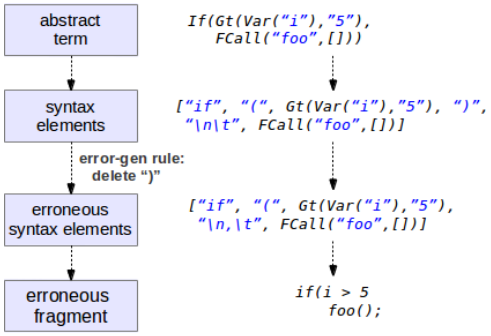


Figure 4.5 Error generation rules create erroneous constructs by modifying the syntax elements of correct constructs.

```
if(i > 5) {          if(i > 5) {          if(i
  foo(i);            if(i > 5)            if(i)
}                    if(i > 5             if(
if(i > 5) {          if(i >               if()
  foo(i);            if(i > )             if
```

Figure 4.6 Incomplete construct: prefixes of an `if` statement.

By composing primitive generation rules, complex clustered errors can be generated. For example, iterative application of the rule `dropLastElement` generates incomplete constructs that miss n symbols at the suffix.

GENERATEINCOMPLETION(*syntaxElements*, n)

1 **for** i **in** 0 **to** n
2 *syntaxElements* ← DROPLASTELEMENT(*syntaxElements*)
3 **return** *syntaxElements*

A more advanced example is provided by nested incomplete construct errors. By applying the `generateIncompletion` rule twice, first to the construct and then to the last abstract child construct in the resulting list of syntax elements, an incomplete construct is created that resides in an incomplete context, for example "`if(i >`". A final example is provided by the (optional) support for automated bracket completion offered by most commonly used IDEs. We simulate this edit scenario by generating test cases with added closing brackets for all unclosed opening brackets in the incomplete prefix of a construct, for example "`if(i > )`". The test cases are generated by composing the `generateIncompletion` rule with a rule that repairs the bracket structure. We omit the pseudo code for the last two examples. Figure 4.6 shows prefixes of an `if` statement that can be constructed from the incompletion rules described in this section.

4.3.2 *Error Seeding Strategies*

In principle, error generation rules could be applied iteratively to all terms in the abstract syntax tree, generating test files each time that the rule application succeeds. However, the resulting explosion of test files increases evaluation time substantially without yielding significant new information about the quality of the recovery technique. As an alternative to exhaustive application, we let the tester specify an error seeding strategy that determines to which terms an error generation rule is applied. Typically, constraints are specified on the sort and/or the size of the selected terms, and, to put a limit on the size of the test suite, a maximum is set or a coverage criterion is implemented. For example, we predefined a coverage criterion that states that all types of constructs that appear in the abstract syntax tree must occur exactly once in the list of selected terms.

4.3.3 *Predefined Generators*

The remaining challenge for test input generation is to implement error generators that cover common syntax errors. From the preliminary study covered in Section 4.2 we conclude that most syntax errors are editing related and generic for different languages. We implemented reusable generators for these scenarios, i.e., all scenarios in Figure 4.3 except the category of language-specific errors. In addition, we implemented a generator that generates erroneous files containing a combination of errors of the identified scenarios. Only a few errors were related to error prone constructs in a particular language. We leave it to an expert of a language to implement custom error generators for language-specific errors.

4.4 AUTOMATED QUALITY MEASUREMENT

An important problem in automated generation of test inputs is automated checking of the outputs, also known as the oracle problem. To judge the quality of a recovery of a test input with a generated error, it is not sufficient to simply compare it to the original, correct program, as essential information may have gone missing in the erroneous input. Therefore, an oracle generation technique is needed to construct the intended recovery for the generated error file. In addition, a quality metric is needed to measure how closely the actual recovery follows the intended recovery.

We extend the test case generation framework with a differential oracle technique that automates quality measurement of recovery outputs. Figure 4.7 shows the work flow for the evaluation framework. The workflow combines the generation of erroneous test inputs with the generation of oracle ASTs that represent their optimal recovery. As a final step, the recovered ASTs for the test inputs are compared to their corresponding oracle AST using a differential oracle metric. The generation of oracle ASTs forms the topic of

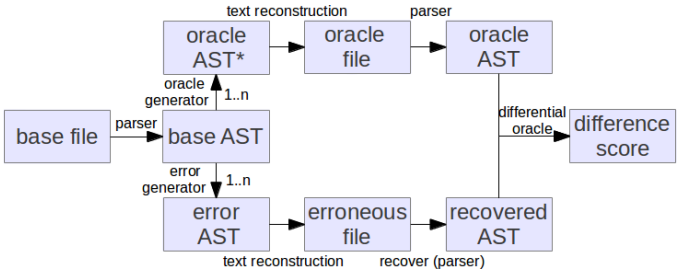


Figure 4.7 Workflow for automated evaluation of error recovery techniques.

Section 4.4.1, while various differential oracle metrics are discussed and compared in respectively Section 4.4.2 and Section 4.4.3.

4.4.1 *Oracle Construction*

The quality of the recovered AST is given by its closeness to the AST that represents the intended program, also called the oracle AST. We automate the construction of oracle ASTs for generated error files. First, we assume that all unaffected code constructs keep their original interpretation. Thus, the constructed oracle AST follows the base AST except for the affected terms. Secondly, we try to construct an oracle interpretation for the affected terms, either by taking the original term as a default oracle interpretation, or by applying a custom oracle generation rule. Only in case both approaches fail, we fall back on manual inspection to construct an appropriate oracle interpretation. Below we explain the construction of oracle interpretations in more detail.

An obvious candidate oracle interpretation for an affected term is the original term from which the affected term was created. The original term most likely provides a suitable recovery for errors that are constructed by deletion, insertion or replacement of literal tokens such as brackets, semicolons, separators and operators. For example, the original construct `while(true){foo();}` provides the preferred recover interpretation for the generated erroneous construct `while(true){foo();` that is created from it by removing the closing parenthesis (`}`). We use the original term as a default oracle term in case no custom oracle rule is specified for a given error generation rule.

Oracle generation rules complement error generation rules by specifying how an oracle interpretation for the affected term is constructed from the original term and the erroneous syntax elements (see Figure 4.5). Oracle rules can handle cases in which the original term is not by definition the preferred recover interpretation; for example because non-literal tokens such as identifiers or numbers are removed from the original syntax by the error generation rule. In this case, the recovery technique can not be expected to reproduce exactly the same syntax as the original file. We discuss two examples.

```
trySelectOracleSubterm(originalTerm, errorElems)
1  subTermCandidate ← null
2  for each elem in errorElems
3     if subTermCandidate == null and elem, originalTerm are same sort then
4           subTermCandidate ← elem
5     else if not elem is literal token then
6           return null
7  return subTermCandidate
```

Figure 4.8 Oracle generation rule that selects a (unique) subterm of the same grammatical sort as the original term.

Figure 4.8 shows an oracle generation rule that can be applied to create an oracle interpretation for broken constructs that contain a sub construct of the same grammatical sort. For example, the rule constructs the numeric (sub)expression `Num(2)` as an oracle for the broken syntax elements `[Num(2), "+"]`, given the original expression term `Plus(Num(2), Num(4))`. The rule selects the appropriate subterm from the erroneous syntax elements (line 3, 4), or fails in case such a subterm does not exist or in case at least one of the (discarded) syntax elements is not a literal token (line 5, 6). In the latter case, the oracle is rejected because alternate, possibly more appropriate interpretations may exist.

Figure 4.9 shows an oracle generation rule that can be applied to create a suitable oracle interpretation for incomplete construct errors. For example, given the original term `while(true){ foo(); }` and the generated incomplete construct `while(true`, the rule constructs the oracle interpretation `while(true){}`. The rule operates by comparing the original syntax elements, `"while", "(", True(), ")", "{", [FunCall("foo")], "}"`, with the erroneous syntax elements `"while", "(", True()`. The result is a list of oracle syntax elements that will be transformed into a proper AST term after text reconstruction and parsing (Figure 4.7). The constructed oracle syntax elements consist of all original elements that are preserved in the the erroneous elements (line 7), plus all literal tokens that are missing (line 10), plus empty list terms that serve as a default for missing list terms (line 12). The rule fails in case other syntax elements are missing (line 14). The given rule can be extended to include support for optional terms and terms for which a suitable default term exists.

In some exceptional cases, error generation rules may generate errors for which it is not feasible to determine a suitable recovery using oracle generation rules. For these cases, manual inspection is required to determine the intended recovery. We apply manual inspection for all cases in which the specified oracle generation rule fails. For cases that are known in advance to be problematic, an oracle generation rule can be specified that always results in a failure.

```
tryConstructOracleCompletion(originalTerm, errorElems)
 1  originalElems ← syntax elements of originalTerm
 2  i ← 0
 3  j ← 0
 4  oracleElems ← []
 5  while i < length(originalElems)
 6      if j < length(errorElems) and originalElems[i] == errorElems[j] then
 7          add originalElems[i] to oracleElems at length(oracleElems)
 8          j ← j+1
 9      else if originalElems[i] is literal token then
10          add originalElems[i] to oracleElems at length(oracleElems)
11      else if originalElems[i] is list term then
12          add empty list term to oracleElems at length(oracleElems)
13      else
14          return null
15      i ← i+1
16  return oracleElems
```

Figure 4.9 Oracle generation rule that compensates for deleted literal tokens and deleted list terms.

The optimal recover interpretation of an erroneous construct can be ambiguous or questionable. For example, the broken arithmetic expression `1 2` has many reasonable recover interpretations, such as: `1 + 2`, `1 * 2`, `1`, and `2`. Selecting only one of these interpretations as the intended interpretation can cause a small inaccuracy in the measured quality. A possible solution is to allow the specification of multiple ambiguous interpretations or the specification of a placeholder interpretation that serves as a wildcard. The disadvantage of this approach is that it violates the well-formedness of the oracle AST which may complicate further processing by differential oracle techniques. For this reason we do not support placeholder oracle terms or ambiguous oracle terms in our framework.

4.4.2 *Quality Metrics*

The remaining problem is to define a suitable metric between recovered programs and their intended oracle programs. Below we discuss different metrics schematically shown in Figure 4.10. In addition, we discuss human criteria introduced by Pennello and DeRemer (Pennello and DeRemer, 1978) as the state-of-the-art, non-automated, quality metric.

Human criteria Human judgement is decisive with respect to the quality of recovery results. For this reason, Pennello and DeRemer (Pennello and DeRemer, 1978) introduce human criteria to rate recovery outputs. A recovery is rated *excellent* if it is the one a human reader would make, *good* if it results

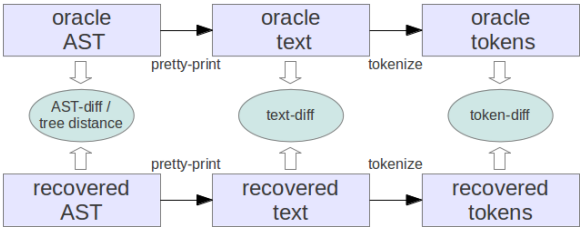


Figure 4.10 Possible metrics on different program representations.

in a reasonable program without spurious or missed errors, and *poor* if it introduces spurious errors or if excessive token deletion occurs. Human criteria represent the state of the art evaluation method for syntactic error recovery, applied in, amongst others, (Pennello and DeRemer, 1978; Pai and Kieburtz, 1980; Degano and Priami, 1995; Corchuelo et al., 2002).

AST diff A simple AST-based metric can be defined by taking the size of the textual AST diff. First, the ASTs of both the recovered program and the intended program are printed to text, formatted so that nested structures appear on separate lines. Then, the printed ASTs are compared by counting the number of lines that are different in the recovered AST compared to the intended AST (the "diff"). The diff size obviously depends on the AST formatter and the settings of the textual diff function. For the results in this section we use the ATerm formatter (van den Brand et al., 2000), and the Unix diff utility with the settings: "-w, -y, –suppress-common-lines, –width=100".

Tree distance An alternative AST-based metric is given by the tree edit distance (Chawathe et al., 1996). Given the edit operations term insertion, term deletion, term move and label update, and given a cost function defined on these operations, the tree edit distance is defined as follows. An edit script between T1 and T2 is a sequence of edit operations that turns T1 into T2. The cost of an edit script is given as the weighted sum over the edit operations contained in the script. The tree edit distance between T1 and T2 is defined as the cost of the minimum-cost edit script. An algorithm to calculate tree edit distances is described in (Chawathe et al., 1996). The tree edit distances in this chapter are calculated based on a cost function that assigns cost 1 to each edit operation.

Token diff A metric can be based on concrete syntax by taking the diff on the token sequences that represent the recovered, respectively the intended program. The token diff counts the number of (non-layout) token insertions and deletions required to turn the recovered token sequence into the intended token sequence. The token sequences can be obtained via pretty-printing and tokenization (Figure 4.10).

Textual diff Assumed that we have a human-friendly pretty printer, a concrete syntax metric can also be based on the textual representations obtained

after pretty printing (Figure 4.10). The textual diff counts the number of lines that are different between the recovered and intended program text. These lines typically correspond to reasonably fine-grained constructs such as statements. The size of the textual diff depends on the pretty-print function and the settings of the textual diff utility. For the results in this section we pretty-print Java programs using the `pp-java` function that is part of the java front end (Bravenboer, 2008) defined in the Stratego/XT framework (Visser, 2004). Furthermore, we use the Unix diff utility with the settings: "-w, -y, –suppress-common-lines, –width=100".

4.4.3 *Comparison of Metrics*

In this section we compare the different metrics based on their accuracy and on qualitative criteria such as applicability and comprehensibility.

Correlation and Accuracy

The Pearson product-moment correlation coefficient (Malgady and Krebs, 1986) is widely used to measure the linear dependence between two variables X and Y. The correlation coefficient ranges from -1 to 1. A correlation equal to zero indicates that no relationship exists between the variables. A correlation of $+1.00$ or -1.00 indicates that the relationship between the variables is perfectly described by a linear equation; all data points lying on a line for which Y increases ($+1.00$), respectively decreases (-1.00), as X increases. More general, a positive correlation coefficient means that X_i and Y_i tend to be simultaneously greater than, or simultaneously less than, their respective means; while a negative correlation coefficient means that X_i and Y_i tend to lie on opposite sides of their respective means.

We measured the correlation between the different metrics based on the Pearson correlation coefficient. For this, we applied the quality metrics to a set of 150 randomly generated Java test cases that cover the different error categories identified in Section 4.2. We translated the human criteria into a numeric scale, i.e., 0 = Excellent, 1 = Good, 2 = Poor. The human criteria results were obtained after manual inspection by two researchers who assessed the recovery qualities independently from each other. Both researchers were familiar with the Pennello and DeRemer criteria. Although the researchers both reported some corner cases, it turned out that the judgements were exactly the same for the 150 inspected recoveries.

The correlation matrix of Figure 4.11 shows the correlation between the different metrics. The results indicate that there is a strong correlation between the tree edit distance, ast diff, and token diff metrics. Figure 4.12 illustrates one of these correlations by plotting the measured values in a scatter diagram. The area of the plotted circles corresponds to the number of data points for the given value.

The correlations that involve human criteria and/or the textual diff metric are somewhat less pronounced. A likely reason is that these metrics have a lower resolution, i.e., they provide a more coarse grained categorization.

	tree distance	ast diff	token diff	textual diff	human criteria
tree distance	1.00	0.95	0.94	0.54	0.71
ast diff	0.95	1.00	0.94	0.60	0.72
token diff	0.94	0.94	1.00	0.51	0.69
textual diff	0.54	0.60	0.51	1.00	0.76
human criteria	0.71	0.72	0.69	0.76	1.00

Figure 4.11 Correlation matrix for different metrics on recovery quality.

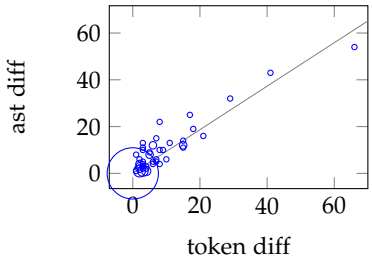


Figure 4.12 A scatter diagram that illustrates the correlation between the ast diff and token diff metrics. The area of the plotted circles corresponds to the number of data points for the given value.

To test this presumption, we also calculated the correlation coefficients after calibrating all metrics to a scale *no diff*, *small diff* and *large diff* so that these categories roughly correspond to the human criteria *excellent*, *good* and *poor*. The results shown in Figure 4.13 indicate a strong correlation between all compared calibrated metrics. Figure 4.14 illustrates how the thresholds were determined by plotting the measured diff values against the corresponding values that followed from human judgement.

The accuracy of a measurement technique indicates the proximity of measured values to 'true values'. A strong correlation between measured values and true values indicates that the measured values are an accurate predictor for true values. Since human criteria are decisive with respect to recovery quality, we use the Pennello and DeRemer criteria to determine the 'true values'. From the results in Figure 4.13 for human criteria we conclude that all metrics have a high prediction accuracy, that is, a high diff is a good predictor of a poor recovery and the same for the other categories.

Qualitative Criteria

Although human criteria are decisive with respect to recover quality, application of these criteria requires manual inspection of the parse results which makes the evaluation subjective and inapplicable in an automated setting. Another disadvantage is the low resolution; only three quality levels are distin-

	tree distance	ast diff	token diff	textual diff	human criteria
tree distance	1.00	0.98	0.96	0.93	0.96
ast diff	0.98	1.00	0.99	0.91	0.97
token diff	0.96	0.99	1.00	0.93	0.98
textual diff	0.93	0.91	0.93	1.00	0.92
human criteria	0.96	0.97	0.98	0.92	1.00

Figure 4.13 Correlation matrix for different metrics on recovery quality. All diff metrics are calibrated to a scale *no diff*, *small diff* and *large diff* so that these categories roughly correspond to the human criteria *excellent*, *good* and *poor*.

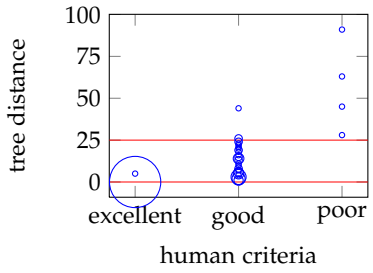


Figure 4.14 Scatter diagram used to determine the thresholds for 'no diff' and 'small diff' for the tree distance metric.

guished. A more refined classification is possible but at the risk of increasing the workload as well as the subjectivity.

AST-based metrics (AST diff and tree distance) reflect recovery quality in the sense that they assign a larger penalty to recoveries that for a larger part of the program provide an inferior interpretation. Empirical evaluation shows a strong correspondence with human judgement (Figure 4.13). A disadvantage of AST-based metrics however is that these metrics are not transparant to humans who are typically more familiar with concrete syntax than with abstract syntax. A related problem is that AST-based metrics depend on the particular abstract interpretation defined for a language. That is, a verbose abstract syntax will lead to higher difference scores than a more concise abstract syntax. The dependence on the abstract interpretation makes it harder to compare parsers for the same language that construct a different abstract representation. To avoid systematic errors, the parser outputs must be translated into the same abstract interpretation; for example via pretty-printing and reparsing. Unfortunately, this may require effort in case the recovered ASTs are not well-formed or in case a pretty-print function is not implemented.

Concrete syntax better reflects human intuition and is independent from the particular abstract interpretation defined for a language. Empirical evaluation shows that metrics based on concrete syntax accurately reflect human

judgement (token diff and textual diff in Figure 4.13). The diff metric based on the token stream has a higher measurement resolution since it can assign a larger penalty to recoveries for which multiple deviations appear on what would be a single line of pretty-printed code. An additional advantage is that the token diff does not depend on the particular layouting that is provided by the pretty-printer. A possible limitation of metrics based on concrete syntax is that they require additional compiler tooling in the form of a human-friendly pretty-printer for textual diffs, and a pretty-printer and tokenizer for token-based diffs.

From this discussion we conclude that all discussed differential oracle metrics are suitable to measure recovery quality. The token-based metric has the advantage that it is understandable for humans, that it does not depend on the particular abstract interpretation constructed by the parser, and that it does not depend on a particular formatting function. However, a practical disadvantage is that additional language tooling that is needed to construct token sequences from abstract syntax trees. From the AST-based metrics, the tree edit distance seems preferable over the AST diff count since it does not depend on a particular formatter and diff function.

In the next section we evaluate the SGLR recovery approach presented in Chapter 2 and 3, using the proposed automated evaluation method. Considering the scale of this study, we use the tree distance metric for comparison, which has adequate accuracy and is most practical since not all the evaluated languages have suitable pretty-printers and tokenizers.

4.5 EVALUATION OF ERROR RECOVERY FOR SGLR

In Chapter 2 and 3 we presented a new, multi-stage recovery approach for scannerless generalized parsing. The approach combines the following techniques. First a region selection technique is applied to detect the erroneous region (RS). In the second stage, the selected region is parsed with a permissive grammar, i.e., a grammar that has been extended with automatically generated recovery rules (PG). In case the permissive grammar technique fails, the erroneous region is skipped as a fallback recovery strategy (RR). In this section we evaluate the approach , focusing on the following properties:

- **Quality**: What is the quality of the recovered AST?
- **Performance**: What is the difference in parsing time between erroneous and correct inputs?
- **Scalability**: Does the approach scale up to large files?

In the remainder of this section we describe our experimental setup, and discuss the results of experiments that evaluate different aspects of our technique.

4.5.1 *Setup*

In this section we describe our experimental setup; we explain how we construct a realistic test set, and how we measure recovery quality and performance. The experimental setup follows the evaluation method proposed in the first part of this chapter.

Syntax Error Seeding

The development of representative syntax error benchmarks is a challenging task, and should be automated in order to minimize the selection bias. Following the approach outlined in Section 4.3, we generate a reasonably large set of syntactically incorrect files from a smaller set of correct base files. We seed syntax errors at random locations in the base files, using a set of rules that cover different types of common editing errors. These rules were established after a statistical analysis of collected edit data for different languages, as described in Section 4.2.

Test Oracle

To measure the quality of a recovery, we compare the AST obtained by parsing the erroneous file against the oracle AST that was constructed a priori by the test generator. In some cases, the generated error is too complex to be solved automatically. For these cases we construct the expected recovery manually. For example, for a "for" loop with an *Incomplete construct* error, such as `for (x = 1; x`, we complete the construct with the minimal amount of symbols possible, which results in the expected construct `for (x = 1; x; );`.

Measuring Quality

We use two methods to measure the quality of the recovery results. First, we do a manual inspection of the pretty-printed results, following the quality criteria of Pennello and DeRemer (1978). Following these criteria, an *excellent* recovery is one that is exactly the same as the intended program, a *good* recovery is one that results in a reasonable program without spurious or missed errors, and a *poor* recovery is a recovery that introduces spurious errors or involves excessive token deletion. The manual assessment of the parse results was done by two researchers, independently from each other. Both researchers were familiar with the Pennello and DeRemer criteria as formulated above.

Since human criteria form an evaluation method that is arguably subjective, as a second method, we also do an automated comparison of the recovered and intended abstract syntax trees. We follow the tree distance metric discussed in Section 4.4.2. The advantage of this approach is that it is objective. Furthermore, since the comparison can be automated, it can be applied to larger test sets.

The scales for the figures we show are calibrated such that "no diff" corresponds to the *excellent* qualification, a "small diff" (1–25 tree edits) roughly corresponds to the *good* qualification, and a "large diff" (26+ tree edits) approximately corresponds to the *poor* qualification. After a selection of recovery

techniques and recovery rule sets, we show both metrics together in a comprehensive benchmark in Section 4.5.2.

Measuring Performance

To compare the performance of the presented recovery technique under different configurations, we measure the additional time spent for error recovery. That is, we compute the extra time it takes to recover from one or more errors (the recovery time) by subtracting the parse time of the syntactically correct oracle file from the parse time of the erroneous file. To evaluate the scalability of the technique, we compare the parse times for erroneous and correct files of different sizes in the interval 1,000–15,000 LOC.

For all performance measures included in this section, we collected average parse times constructed from three individual parses per file. All measuring is done on a "pre-heated" JVM running on a laptop with an Intel(R) Core(TM) 2 Duo CPU P8600, 2.40GHz processor, 4 GB Memory.

Test Sets

To evaluate quality and performance of the suggested recovery techniques we use a test set of programs written in WebDSL, Stratego-Java, Java-SQL and Java, based on the following projects:

- *YellowGrass*: A web-based issue tracker written in the WebDSL language.[1]
- *The Dryad compiler*: An open compiler for the Java platform (Kats et al., 2008) written using Stratego-Java.
- *The StringBorg project*: A tool and grammar suite that defines different embedded languages (Bravenboer et al., 2010), providing Java-SQL code.
- *JSGLR*: A Java implementation of the SGLR parser algorithm.[2]

We selected five representative base files from each project, and generated test files using the error seeding technique. In total, we generated 176 Stratego-Java test cases, 190 WebDSL test cases, 195 Java-SQL test cases, and 329 Java test cases. In addition, for testing of scalability, we constructed a test set consisting of 28 erroneous Stratego-Java files of increasing size in the interval of 1,000–15,000 LOC.

Parser Configuration

We performed initial experiments to select the best configuration of recovery rule sets and recovery techniques. Section 2.6 covers the comparison of different automatically generated recovery rule sets composed from: deletion rules (D), insertion rules for closing symbols (C), and insertion rules for opening symbols (O). The comparison shows that the DC rule set provides the best results. Section 3.5 covers the comparison of different combinations of

[1] `http://www.yellowgrass.org/`.
[2] `http://strategoxt.org/Stratego/JSGLR/`.

techniques, e.g., region selection (RS), permissive grammars (PG), and region recovery (RR). The comparison shows that RS-PG-RR provides the optimal combination. All results in this section are obtained with the DC rule set and the RS-PG-RR configuration. We set a time limit of 500 milli-seconds for applying recovery rules (PG), and an overall time limit of 5 seconds to cut off recoveries that take an (almost) infinite time to complete.

4.5.2 *Experiments*

In this section we present the results of the experiments we performed to evaluate different aspects of our technique.

Overall Benchmark

As an overall benchmark, we compare the quality of our techniques to the parser used by Eclipse's Java Development Tools (JDT). To ensure that all the results are obtained in a reasonable time span, we set a recover time limit of 1 second. It should be noted that, while our approach uses fully automatically derived recovery specifications, the JDT parser in contrast, uses specialized, handwritten recovery rules and methods. We use the JDT parser with statement-level recovery enabled, following the guidelines given by Kuhn and Thomann (2006).

Both Eclipse and our approach apply an additional recovery technique in the scenario of content completion (see Chapter 5). Both techniques use specific completion recovery rules that require the completion request (cursor) location as additional information, also, these rules construct special completion terms that may not represent valid Java syntax. We did not include these techniques in this general benchmark section since they specifically target the use case of content completion and do not work in other scenarios.

Figure 4.15 shows the quality results acquired for the Java test set by applying the criteria of Pennello and DeRemer (1978) and by using the tree-edit distance metric. To ensure that the tree-edit distance is calculated on the same AST format, we translated all SGLR abstract syntax trees into abstract syntax trees in the JDT format.

The results show that the SGLR recovery, using different steps and granularity, is in particular successful in avoiding large diffs, thereby providing more precise recoveries compared to the JDT parser. The JDT parser on the other hand managed to construct an excellent recovery in 67% of the cases, which is a bit better than the 62% of the SGLR parser. The SGLR parser failed to construct an AST in less than 1% of the cases, while the JDT parser constructed an AST in all cases. However, manual inspection revealed that in many large diff cases only a very small part of the original file is reconstructed, for example, only the import lines or the import lines plus the class declaration whereby all declarations in the body are skipped. We conclude that our automatically derived recovery technique is at least on par with practical standards.

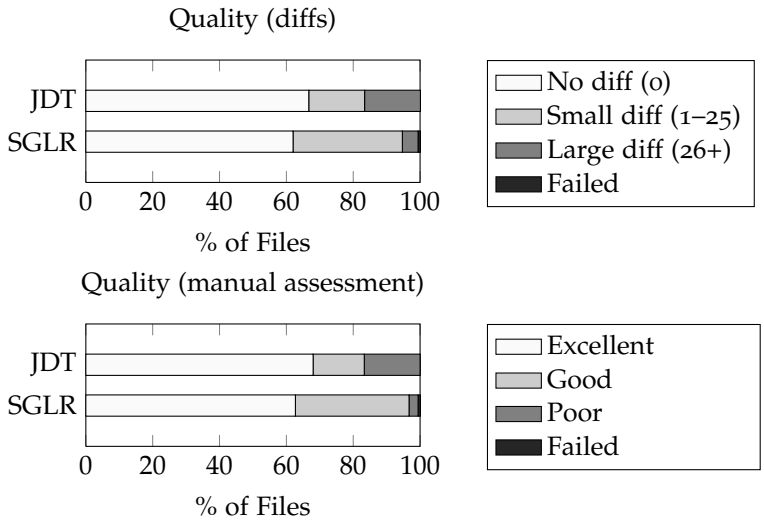


Figure 4.15 Recover quality of the SGLR parser compared to the JDT parser.

Cross-language Quality and Performance

In this experiment we test the applicability of our approach to different languages. For simplicity and to ensure a clear cross-language comparison, we focus only on syntax errors that do not require manual reconstruction of the expected result, i.e., *random errors*, *scope errors* and *string or comment errors*. This allows for a fully automated comparison of erroneous and intended parser outputs. The results of the experiment are shown in Figure 4.16. The figure shows good results and performance across the different languages. From the diagram it follows that the quality of the recoveries varies for the different test sets. For example, the recoveries obtained for Java-SQL are better than the ones for Stratego-Java. Differences like these are both hard to explain and predict, and depend on the characteristics of a particular language, or language combination, as well as the test programs used.

Performance and Scalability

In this experiment we focus on the performance of our approach. We want to study the scalability of our recovery technique and the potential performance drawbacks of adding recovery rules to a grammar, i.e., the effect of increasing the size of the grammar. We use the Stratego-Java language throughout this experiment with the RS-PG-RR recovery configuration and the DC grammar.

To test scalability, we constructed a test set consisting of files of different size in the interval $1,000 - 15,000$ LOC, obtained by duplicating 500-line fragments from a base file in the Stratego-Java test set. For each test file, the same number of syntax errors were added, scattered in such a way that clustering of errors does not occur. We measure parse times as a function of input size, both for syntactically correct files and for files that contain syntax errors. The

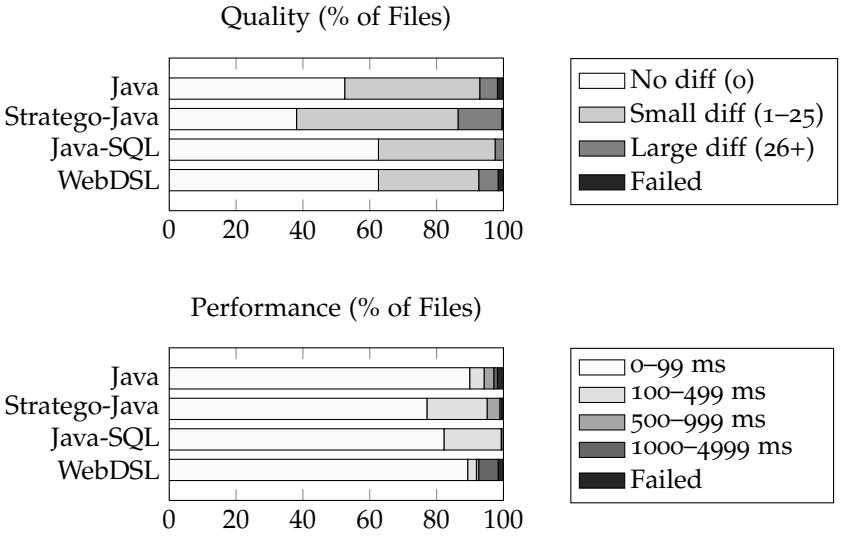


Figure 4.16 Quality and performance (recovery times) for different languages.

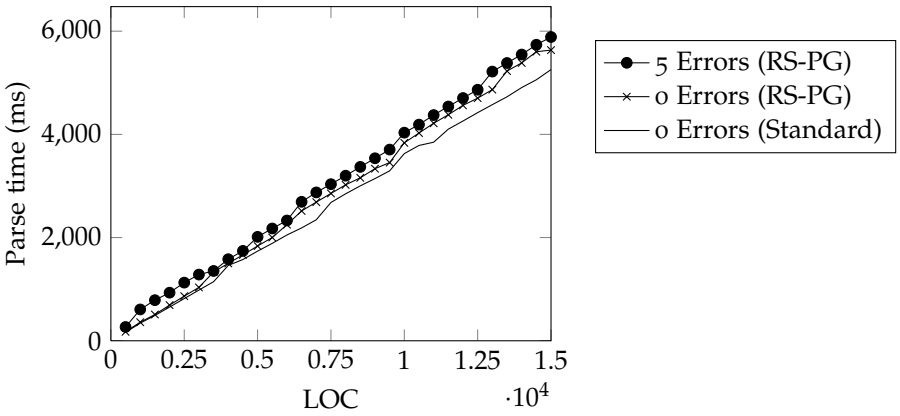


Figure 4.17 Parse times for files of different length with and without errors. The files are written in the Stratego-Java language and parsed with the RS-PG recovery configuration.

results, plotted in Figure 4.17, show that parse times increase *linearly* with the size of the input, both for correct and for incorrect files. Furthermore, the extra time required to recover from an error (recovery time) is independent of the file size, which follows from the fact that both lines in the figure have the same coefficient.

As an additional experiment we studied the performance drawbacks in the increased size of a permissive grammar. The extra recover productions added to a grammar to make it more permissive also increase the size of that grammar, which may negatively affect parse times of syntactically correct inputs.

We measured this effect by comparing parse times of the syntactically correct files in the test set, using the standard grammar and the DC permissive grammar. The results show that the permissive grammar has only a small negative effect on parse times of syntactically correct files. The effect of modifying the parser implementation to support backtracking was also measured, but no performance decrease was found. We consider the small negative performance effect on parsing syntactically correct files acceptable since it does not significantly affect the user experience for files of reasonable size.

4.5.3 *Summary*

In this section we evaluated the effectiveness of our recovery approach based on the combined application of region selection, permissive grammars, and region recovery. The permissive grammar technique was introduced in Chapter 2 and the region selection and region recovery techniques were introduced in Chapter 3.

To measure the quality of our recovery approach, we did a baseline comparison with the Eclipse JDT parser, considered the industry standard for Java parsing. Our results showed that the recovery quality holds up to the standard set by the JDT.

We showed that the approach is suitable for different languages. We tested the approach on Java, on the composite of the declarative query language SQL and Java in the form of Java-SQL, on the Stratego transformation language with embedded Java expressions, and on the domain-specific WebDSL language. With this variety of languages and language compositions, we showed that the approach is language independent.

We evaluated the performance of the approach by measuring the time required to recover from parse errors. The results show that the recovery time is independent of file size, and that the recovery gives no substantial performance overhead in the majority of the cases.

We also measured the overhead time introduced by the approach for parsing correct fragments of code. The results show that the additional recovery productions of the permissive grammars only give a small overhead compared to the base grammars, while the overhead of the extended parser infrastructure is negligible. Finally, we measured parse times as a function of input sizes. Parse times increase linearly with the size of the input, both for correct and for incorrect files.

4.6 DISCUSSION

In this chapter we introduced a method for automatic evaluation of parse error recovery techniques. We applied this method to evaluate the approach described in Chapter 2 and 3. The automated evaluation method made it possible to do an extensive evaluation that considers multiple parser configurations and multiple languages. In this section we discuss the validity of our experimental results.

We identify some threats to external validity. External validity refers to the question whether we can generalize the results obtained in the experiment to other, similar situations. We base our results on generated syntax errors. A possible threat to validity is that they do not accurately reflect the syntax errors experienced "in the wild". To gain confidence in how representative the generated syntax errors are, we base them on edit scenarios identified in an empirical study on editing behavior. This study is based on three different programming languages and covers edit scenarios from multiple programmers. Still, the study could be improved by including more 'real world' languages, and by including multiple editing environments. We leave this as future work.

We identify some threats to construct validity. Construct validity refers to whether an experiment measures the desired construct adequately. We study the quality of parse error recovery, defined as the quality of the recovered ASTs. To measure the quality of a recovered AST, we calculate the tree-edit distance between the recovered AST and a predefined oracle AST. We identify two threats to validity with respect to this quality measurement method. First, multiple alternative oracle ASTs may exist that represent recoveries that are as good as the given oracle AST. The effect of this threat should be small: at worst, this may result in some recoveries being judged as "good" when they are in fact "excellent". Second, the tree-edit distance metric may not accurately reflect the quality of the recovered AST. To gain confidence in our quality metric, we compared different oracle based metrics and a metric based on human assessment. The comparison shows a high correlation between all those metrics.

An important use case of parse error recovery is to support editor services in an interactive environment. In our evaluation, we have focused on the quality and performance of error recovery. If the ultimate goal of an error recovery implementation is to provide the best possible basis for editor services, a final threat to validity is the assumption that the quality of error recovery indeed determines the quality of editor services. An additional user study could be performed to measure the direct effect on editor services.

4.7 RELATED WORK

Error recovery evaluation Recovery techniques in literature use test sets that are either manually constructed (de Jonge et al., 2009; Nilsson-Nyman et al., 2009), or composed from practical data (Pennello and DeRemer, 1978; Ripley and Druseikis, 1978). According to our knowledge, test generation techniques have not yet been applied to recovery evaluation.

Human criteria (Pennello and DeRemer, 1978) and differential oracle techniques (de Jonge et al., 2009; Nilsson-Nyman et al., 2009) form the state of the art methods to measure the quality of recovery results. We accomplished to apply a differential oracle technique in a full automatic setting.

Analysis of syntax errors To gain insight into syntax errors that are actually made during editing, we investigated edit behavior by applying a statistical

analysis of collected edit data. Previous studies on syntax errors (Litecky and Davis, 1976; Ripley and Druseikis, 1978) focus on on-demand compilation. These studies report that most syntax errors consisted of a single missing or erroneous token. In contrast, we studied syntax errors that occurred during background compilation as applied in modern IDEs. Comparison of the results show that syntax errors that occur during background compilation are more complex and often involve multiple errors clustered in a small fragment.

Fuzzing Fuzzing is a popular method used in software testing. The technique of fuzzing consists of sending a large number of different inputs to a program to test its robustness against unpredictable or unvalid data. The inputs are normally constructed in one of two fashions, generation or mutation. Generation approaches produce test data based on a specification of the set of input data. Alternatively, mutation-based approaches modify valid input data collected from practical data. The modifications may be random or heuristic and are possibly guided by stochastic parameters. Mutation and generation-based fuzzing techniques are compared in (Miller and Peterson, 2007). We use a mutation based fuzzing technique to generate testcases for syntactic error recovery.

Compiler testing Previous work on compiler testing (Kossatchev and Posypkin, 2005) primarily focuses on the generation of valid input programs. Based on a language specification, valid sentences are generated that form positive test cases for the parser implementation. A parser for a language must not only accept all valid sentences but also reject all invalid inputs. Only a few papers address the concern of negative test cases, a generation based approach is discussed in (Zelenov and Zelenova, 2005). Generation based techniques are good at constructing small input fragments in a controlled manner. The problem these techniques address is to meet a suitable coverage criterion by a (preferable small) set of generated testcases (Harm and Lämmel, 2000; Zelenov and Zelenova, 2005).

In contrast, the evaluation of error recovery techniques exclusively targets negative test cases. Furthermore, the generated negative test cases must be realistic error scenarios instead of small input fragments that are 'likely to reveal implementation errors'. Generation-based techniques construct testcases starting from a formal specification. It hardly seems possible to formally specify what a realistic input fragment is that contains realistic syntax errors. For this reason, we consider a mutation based fuzzing technique more appropriate for the generation of error recovery test inputs.

Papers on compiler testing generally consider the parser as a language recognizer that outputs `true` in case a sequence of characters belongs to a given formal language and `false` otherwise. Other features of the parser implementation such as the constructed abstract syntax tree and the reported error messages are ignored. In contrast, we focus on the quality of the constructed AST which we evaluate using an oracle based technique.

(Regehr et al., 2012) presents an approach to automate test case reduction for C programs. That is, given a program that triggers a bug in the C com-

piler, an automated test case reducer searches for ever-smaller inputs that still trigger the same bug. The most successful reduction strategy that the authors present applies a generic fixpoint computation that invokes pluggable, compiler-like transformations that implement reduction operations. Though the paper focuses on a different problem in automated compiler testing, the provided solution is similar to our solution in the sense that it applies pluggable transformations to modify a given input program.

4.8 CONCLUSION

Automated evaluation of parse error recovery techniques has been an open problem in the area of compiler testing. In this chapter we introduced a method for fully automated recovery evaluation; the method combines a mutation-based fuzzing technique that generates realistic test inputs, with an oracle-based evaluation technique that measures the quality of the outputs.

We applied the evaluation method to evaluate the recovery approach that we implemented for SGLR. Based on an extensive evaluation, we conclude that the SGLR recovery approach has a low performance overhead and provides good or excellent recovery solutions in the majority of cases. Comparison with the JDT parser shows that the recover quality holds up to practical standards.

Automated evaluation makes it feasible to do a benchmark comparison between different techniques. As future work we intend to extend the benchmark comparison with different parsers used in common IDEs. Furthermore, we plan to extend the empirical foundation of our recovery evaluation method.

Integrating Error Recovery in the Spoofax Language Workbench

5

ABSTRACT

Spoofax is a development environment for textual languages that combines the construction of languages and editor services. Using SDF and (J)SGLR, Spoofax has the distinguishing feature that it supports language embeddings and extensions composed from separate grammar modules. Parse error recovery is essential for interactive editing. In Chapter 2 and 3 we presented three parse error recovery techniques, that we combined in a multi-staged recovery approach for SGLR. The present chapter describes the integration of this approach in Spoofax and presents general techniques for the implementation of an IDE based on scannerless generalized parsing.

5.1 INTRODUCTION

While IDEs for languages have been constructed and used for several decades, only recently did they become significantly more sophisticated and indispensable for the productivity of software developers. In early 2001, IntelliJ IDEA (Saunders et al., 2006) revolutionized the IDE landscape, setting a new standard for highly interactive and language-specific IDE support for textual languages. Since then, providing good IDE support for new languages has become mandatory, posing a significant challenge for language engineers.

To lower the threshold of creating new languages and developing IDEs for these languages, *language workbenches* have been developed that combine the construction of languages and language-specific editors. Language workbenches improve the productivity of language engineers by providing specialized languages, frameworks, and language engineering tools. Examples of language workbenches for textual languages include EMFText (Heidenreich et al., 2009), MontiCore (Krahn et al., 2008; Grönniger et al., 2008), TCS (Jouault et al., 2006), Xtext (Efftinge and Voelter, 2006), ASF+SDF Meta-Environment (Klint, 1993; van den Brand et al., 2001), Rascal (van der Storm, 2011; Klint et al., 2009), and Spoofax (Kats and Visser, 2010).

The central artifact that language engineers define in a language workbench is the grammar of a language, which is used to generate a parser. The parser constructs an abstract representation of the source program that provides a basis for all interactive editor services. Traditionally, IDEs used handwritten parsers or only did a lexical analysis of source code for syntax highlighting in real-time. By using a generated parser that runs in the background every time the source code changes, modern IDEs have access to more accurate and

more up-to-date information, but they also crucially depend on the parser's performance and its support for error recovery.

Scannerless generalized parsers support the full class of context-free grammars, thereby allowing a natural expression of the intended syntax and offering support for language extension and composition (Visser, 1997b). In Chapter 2 and 3 we presented a language-independent recovery approach for scannerless generalized parsing. The approach combines a layout sensitive technique to select erroneous regions and a recovery technique based on grammar relaxation with automatically generated recovery rules. Evaluation of the approach (Section 4.5) showed that it has a low performance overhead and provides good or excellent recovery quality in the majority of cases. In the present chapter we describe the integration of the approach in Spoofax (Kats and Visser, 2010), an environment for developing languages and accompanying IDE support.

Ultimately, error recovery provides a speculative interpretation of the intended program, which may not always be the desired interpretation. As such, it is both unavoidable and not uncommon that editor services operate on inaccurate or incomplete information due to imperfectly recovered syntax errors. Experience with modern IDEs (e.g. Eclipse, Visual Studio) shows that this is not a problem in itself, as programmers are shown both syntactic and semantic errors directly in the editor. In the worst case, the provided feedback may be not optimal. Still, there are a number of editor services that inherently require some interaction with the recovery strategy. In this chapter we present general techniques for the implementation of editor services that interact with the parse error recovery technique.

Where other editor services should behave robustly in case of incomplete or syntactically incorrect programs, the content completion service is almost exclusively targeted towards incomplete programs. Essential for the completion service is the interpretation of the syntactic structure near the cursor location, i.e., the location where the completion is requested. In the current chapter we develop an additional recovery technique to interpret the local context of the completion request location.

Outline We first give an overview of the recovery techniques that we developed for SGLR (Section 5.2). Then, in Section 5.3, we describe the implementation of these techniques in the Spoofax language workbench. Next, in Section 5.4, we present general techniques to handle the interaction of error recovery with editor services. Finally, in Section 5.5 we evaluate the additional recovery technique that we developed to improve the provision of semantic content completion suggestions.

5.2 OVERVIEW RECOVERY APPROACH

This section gives an overview of the multi-stage recovery approach we implemented for SGLR. We summarize the individual techniques presented in Chapter 2 and 3, and describe their integration.

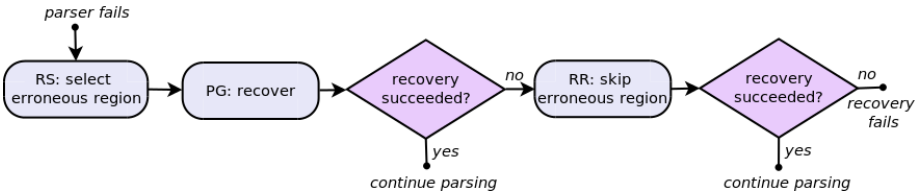


Figure 5.1 Overview integrated recovery approach implemented in JSGLR.

Permissive grammars and backtracking In Chapter 2 we introduced permissive grammars (Section 2.4), a recovery technique for scannerless generalized LR parsing based on grammar relaxation; after analysis of the original grammar, a set of recovery rules that simulate token insertion or deletion is automatically derived. The recovery rules are applied in an on-demand fashion, using a backtracking algorithm (Section 2.5). Starting from the parse failure location, this algorithm explores an increasing, backward search space to find a (presumable) minimal-cost solution for applying recovery rules.

Region selection Region selection, introduced in Section 3.4, is a regional recovery technique that uses layout to select regions of code that enclose syntax errors. The selected regions can be analyzed in detail by a correcting technique, or discarded if no recovery is found within an acceptable time span.

Integration of techniques We combine the different techniques in a multi-stage recovery approach, illustrated in Figure 5.1. Region selection is applied first to detect the erroneous region. In case region selection fails, the whole file is selected as erroneous region. In the second stage, the permissive grammar technique is applied, where backtracking is restricted to the erroneous region. In case the permissive grammar technique fails, the erroneous region is skipped as a fallback recovery strategy.

5.3 IMPLEMENTATION

We implemented our approach in Spoofax (Kats and Visser, 2010), which is a language development environment that combines the construction of languages and editor services. Using SDF (Heering et al., 1989b; Visser, 1995) and (J)SGLR[1], Spoofax has the distinguishing feature that it supports language embeddings and extensions composed from separate grammar modules. In this section we give an overview of the general system and we discuss the adaptations we made for error recovery.

Figure 5.2 gives a general overview of the tool chain that handles parsing in Spoofax with integrated support for error recovery. Given a grammar definition in SDF, the *make-permissive* tool generates a permissive version of this grammar, for which a parse table is constructed by *sdf2table*. This parse table

[1]Java implementation of the SGLR parsing algorithm. `http://strategoxt.org/Stratego/JSGLR/`

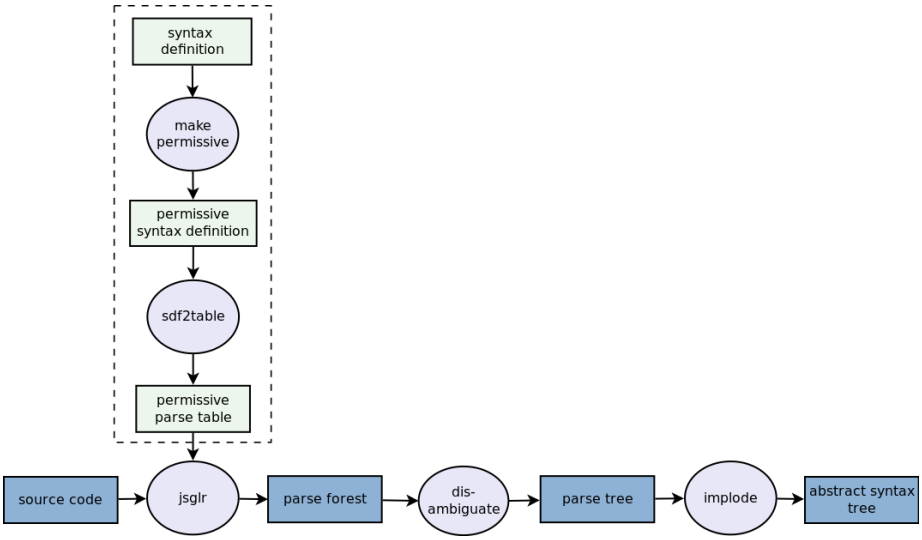


Figure 5.2 Overview tool chain. Make-permissive generates a permissive version of the original grammar, for which a parse table is constructed by sdf2tbl. The (permissive) parse table is used by JSGLR to construct a parse tree for a (possible erroneous) input file, which is then disambiguated and imploded into an AST.

is used by the *JSGLR* parser, which constructs a parse tree for a (possible erroneous) input file. The parse tree is first disambiguated by applying post-parse filters, and then imploded into an bastract syntax tree.

The *make-permissive* tool was added to the tool chain specifically for the concern of error recovery. The tool implements a grammar-to-grammar transformation, applying the heuristic rules described in Section 2.4.5 to generate recovery rules. The tool is implemented in Aster (Kats et al., 2009c), a language for decorated attribute grammars that extends the Stratego transformation language.

We adapted the JSGLR parser implementation so that it can efficiently parse correct and incorrect syntax fragments using the productions defined by the permissive grammar. For this reason, we implemented a selective form of backtracking specificly for recover productions. In addition, we implemented region selection as an additional recovery technique. All mentioned techniques are implemented in Java and integrated in the JSGLR implementation. To summarize, we made the following adaptations to the Java based JSGLR parser:

- ignore grammar productions labeled with the recover annotation, unless the parser is in recovery mode (*permissive grammars*).
- the backtracking algorithm of Section 2.5.4, to efficiently apply recover productions when the parser is in recovery mode (*permissive grammars*).

- a runtime disambiguation filter that selects the branch with the lowest number of recover productions, preferring insertion productions over deletion productions (*permissive grammars*).

- the region selection algorithm of Section 3.4.4, to select erroneous, discardable regions (*region selection*).

- some code to integrate the different recovery techniques, as described in Section 5.2.

5.4 INTEGRATING ERROR RECOVERY IN AN IDE

A key goal of parse error recovery is its application in IDEs. Modern IDEs heavily rely on parsers to produce abstract syntax trees that form the basis for editor services such as the outline view, content completion, and refactoring. Users expect these services even when the program has syntactic errors, which is very common when source code is edited interactively. Using error recovery, the parser can construct an AST for a syntactically incorrect input program.

Ultimately, a recovered AST represents a speculative interpretation of the intended program, which may not always be the desired interpretation. Experience with modern IDEs (e.g. Eclipse, Visual Studio) shows that for most services it is not a problem to operate on inaccurate or incomplete information; in the worst case, the provided feedback is not optimal. For services that involve complex program modifications such as refactorings, errors and warnings can be presented to the user who can then decide whether or not to continue. In this section, we describe the role of error recovery in different editor services and show general techniques for using error recovery with these services.

5.4.1 *Guarantees on Recovery Correctness*

While a recovery technique ultimately provides a speculative interpretation of a syntactically incorrect input program, our approach does guarantee well-formedness of ASTs. That is, it will only produce ASTs that conform to the abstract structure imposed by the production rules of the original (non-permissive) grammar. This property is maintained for all our recovery techniques. With respect to permissive grammars (Section 2.4), the automatically derived recovery rules are chosen so that they do not violate the well-formedness property. That is, deletion and literal insertion recovery rules do not contribute AST terms since they produce respectively layout and literal symbols which are not part of the AST; insertion recovery rules for lexical productions only contribute lexical terms that correspond to the recovered lexicals. Region recovery (Section 3.4) does not compromise the well-formedness property of the parse result since it only modifies the input string by skipping over a text fragment.

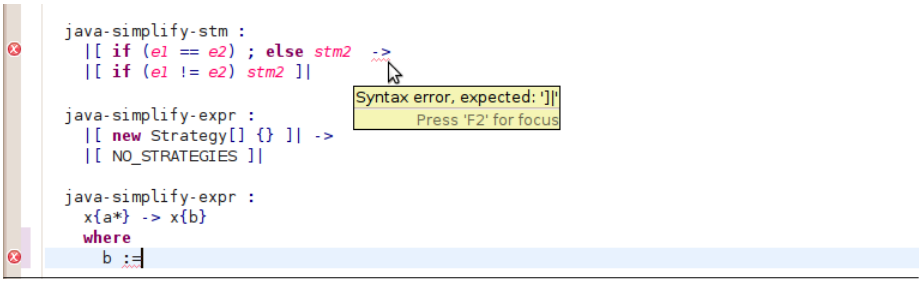


Figure 5.3 An editor for Stratego with embedded quotations of Java code.

The property of well-formedness of trees significantly simplifies the implementation and specification of editor services, since they do not require any special logic to handle artificial recovery constructs with missing terms or special constructors introduced by the recovery technique. This approach also ensures separation of concerns: error recovery is purely performed by the parser, while editor services take as input a well-formed AST that represents a complete and correct program. Still, there are a number of editor services that inherently require some interaction with the recovery strategy, which we discuss next.

5.4.2 *Syntactic Error Reporting*

The traditional use case of error recovery has been to report multiple errors in a file. Syntax errors are reported to users by means of an error location and an error message. In traditional compilers, the error location was reported as a line/column offset, while modern IDEs use the location for the placement of error markers in the editor. Figure 5.3 shows a screenshot of an editor for Stratego with embedded Java. The shown code fragment contains two syntax errors which are marked inline with a red squiggle. The error message pops up when the cursor hovers over the error marker.

We reconstruct the information required for error reporting from the constructed parse tree, e.g., before it is imploded into an AST. Application of a recovery rule (permissive grammars) is recorded in the parse tree in the form of a deletion/insertion term that stores information about the deleted/inserted substring. To detect discarded regions (region recovery), we compare the sequence of parsed characters extracted from the parse tree, with the sequence of characters that forms the original input.

We report syntax errors using generic error messages that depend on the type of recovery. For deletion rules and for discarded regions, we use "[string] not expected", for insertion rules and inserted scope closings we use "expected: [string]", and for insertion rules that terminate a construct we use "construct not terminated". The location at which the errors are reported is determined by the location at which a recovery rule is applied, rather than by the location of the parse failure. For region recoveries, where no recovery rule

is applied, the start and end location of the region, plus the original failure location is reported instead.

Compared to hand-written parsers, which are commonly used in interactive editing, our error reports tend to be of a more generic nature. For example, for an unterminated string, our editor gives a generic message "construct not terminated" instead of a language-specific message "string literal not terminated". For custom recovery rules it makes sense to allow language developers to define a custom error message which can be specified as an additional grammar annotation. After parsing, the annotation can be read from the parsetree and reported to the user. Custom error messages are not yet supported in Spoofax.

5.4.3 *Syntax Highlighting*

Syntax highlighting has traditionally been based on a purely lexical analysis of programs. The most basic approach is to use regular expressions to recognize reserved words and other constructs and assign them a particular color. Unfortunately, for language engineers the maintenance of regular expressions for highlighting can be tedious and error prone. A more flexible approach is to use the grammar of a language; a scanner recognizes tokens in a stream, which are used to assign colors.

More recent implementations of syntax highlighting do a full context-free syntax analysis. By inspecting the parse tree constructed by a (scannerless) parser, these implementations can distinguish between keywords in different sublanguages. Some editors also use the semantics of a language for syntax highlighting. For example, they may assign Java field accesses a different color than local variable accesses. In both cases the syntax highligthing mechanism requires a full syntax analysis provided by the parser.

Scannerless syntax highlighting When using a scannerless parser such as SGLR, a scanner-based approach to syntax highlighting is not an option; files must be fully parsed instead. This makes it important that a proper parse tree is available at all times, even in case of syntactic errors. To illustrate this, consider the following incomplete Java statement:

```
Tree t = new
```

Using a scanner, the word `new` can be recognized as one of the reserved keywords and can be highlighted as such. In the context of scannerless parsing, a well-formed parse tree must be constructed for the keyword to be highlighted. For fragments with syntax errors such as this one, that may not be possible, resulting in no highlighting for the `new` keyword.

Fallback syntax highlighting Syntax highlighting is equally or even more important for syntactically incorrect fragments than for syntactically correct fragments, as it indicates how the editor interprets the fragment as a programmer is editing it. In the context of scannerless parsing, the syntax highlighting of a fragment depends on the interpretation of that fragment constructed by the

parser. As a consequence, syntax highligthing cannot be provided for erroneous fragments for which the parser failed to construct an interpretation.

To address this issue, we implemented a *fallback syntax highlighting* mechanism. The fallback syntax highlighting uses a lexical analysis for those fragments where the full context-free parser is unable to interpret the words to be highlighted. This analysis is performed by a rudimentary tokenizer that recognizes separate words to distinguish them for colorization. Simple coloring rules can then be applied to highlight, for example, all the reserved keywords, comments and string literals.

A limitation of the approach is that with a tokenizer it cannot distinguish between keywords in different sublanguages, making the approach only viable as a fall-back option. We use the fallback syntax highlighting for discarded regions and in case the combined recovery technique fails, e.g., no AST is constructed for the erroneous program.

5.4.4 *Content Completion*

Content completion, sometimes called content assist, is an editor service that provides completion proposals based on the syntactic and semantic context of the expression that is being edited. Where other editor services should behave robustly in case of incomplete or syntactically incorrect programs, the content completion service is almost exclusively targeted towards incomplete programs. Content completion suggestions must be provided regardless of the syntactic state of a program, e.g., an incomplete expression '`blog.`' does not conform to the syntax, but for content completion it must still have an abstract representation.

Completion recovery rules In case content completion is applied to an incomplete construct, the syntactic context of that construct must be recovered. This is especially challenging for language constructs with many elements, such as the "for each" statement in the Java language. Even if only part of such a statement is entered by a user, it is important for the content completion service that there is an abstract representation for it. Based on the recovery rules of Section 2.4 this is not always the case. Deletion recovery rules interpret the incomplete expression as layout. As a consequence, the syntactic context is lost. Insertion recovery rules can recover some incomplete expressions, but they only insert missing terminal symbols.

To address this limitation, we introduce specific recovery rules for content completion that specify what abstract representation to use for incomplete syntactic constructs. Figure 5.4 shows examples of these rules. The first rule is a normal production rule for the Java "for each" construct. The second rule indicates how to recover this construct if the `Stm` non-terminal is omitted, while the third rule handles the case where both non-terminals are omitted. The completion recovery rules use the `{ast(`*p*`)}` annotation of SDF to specify a pattern *p* as the abstract syntax to construct. Furthermore, a placeholder pattern `NULL()` is used in place of the abstract representation of an omitted non-terminal.

```
context-free syntax

  "for" "(" FormalParam ":" Expr ")" Stm  →
     Stm {cons("ForEach")}

  "for" "(" FormalParam ":" Expr ")"?  →
     Stm {ast("ForEach(<1>, <2>, NULL())"), completion}

  "for" "(" FormalParam ":"? ")"?  →
     Stm {ast("ForEach(<1>, NULL(), NULL())"), completion}
```

Figure 5.4 Java `ForEach` production and its derived completion rules.

```
context-free syntax

  "for" "(" FormalParam ":" Expr ")"?  →
     Stm {ast("ForEach(<1>, <2>, Block([]))"), completion}
```

Figure 5.5 Java `ForEach` completion rule with placeholder pattern that matches the signature of the original production.

The completion recovery rules are automatically derived by analyzing the original productions in the grammar, creating variations of existing rules with omitted non-terminals and terminals marked as optional patterns. For best results, we generate rules that use placeholder patterns that reflect the signature of the original production and therefore preserve the well-formedness property. For example, in the second rule of Figure 5.4, the pattern `Block([])` can be used instead of the `NULL()` placeholder, which results in the recovery rule shown in Figure 5.5. Sensible placeholder patterns are constructed by recursively analyzing the production rules for the omitted non-terminals. In the given example, the production rule `"{" Stm* "}" -> Stm {cons("Block")}` provides the pattern `Block([])` as a placeholder for the `Stm` non-terminal, using the empty list `[]` as the basic default for list productions.

Runtime support Completion recovery rules are only applied in case content completion suggestions are requested by the user. We implemented a slightly different runtime support for this scenario. In the case of a content completion request, an incomplete construct is likely to occur at the cursor location. We use this information to control the application of completion rules. That is, instead of backtracking after a parse failure, we apply completion recovery rules if they interpret a character sequence that overlaps with the cursor location. This approach adequately completes constructs at the cursor location and minimizes the overhead of completion rules in normal parsing and other recovery scenarios. It also ensures that the completion recovery rules have precedence over the normal deletion and insertion recovery rules for the content completion scenario.

A second difference with the normal recovery scenario is the handling of ambiguous recover interpretations. As an example, we consider the incomplete Java assignment `i = a` whereby the completion is requested at the "a" character. The assignment can be completed as `i = a;` or, alternatively, as `i = a();`. By applying automatically derived completion rules, the parser

```
lexical syntax

  → "@#$" {completion_start}

context-free syntax

  "@#$" "for" "(" FormalParam ":" Expr ")"?  →
     Stm {ast("ForEach(<1>, <2>, Block([]))"), completion}
```

Figure 5.6 A completion rule optimized for performance. The `completion_start` production is only applied in the nearby left context of the cursor location and thus prevents the construction of extra branches at other locations.

constructs two alternative interpretations: `Assign(Var("i"),Var("a"))` and `Assign(Var("i"),FunCall("a",[]))`. Since the intention of the programmer is unclear at this point, both interpretations must be preserved in the parser output and provided to the completion service. Based on these interpretations, the completion service constructs a list with names of visible declarations that start with an "a" and represent either variabele names (for `Var("a")`) or function names (for `FunCall("a",[])`).

For normal editing scenarios, the completion rules can also be applied as an additional recovery mechanism that is effective at the cursor location. In that case, completion rules that compromise the well-formedness of the AST must be excluded. In addition, alternative completion interpretations must be disambiguated, either at runtime, or by applying a post-parse filter.

Performance optimization The additional completion productions potentially increase parse times for the normal parsing scenario; first by increasing the size of the grammar, secondly by creating extra stack branches that are discarded only after the completion annotation is considered. Practical experimentation showed that in particular the extra stack branches cause inacceptable performance overhead for the normal parsing scenario.

To prevent the construction of extra branches that will eventually be discarded, we introduced a slightly different formulation of completion rules. The adapted formulation introduces a `completion_start` rule that produces an artificial literal which is required as the start literal for all completion rules. The application of the `completion_start` rule is restricted to the nearby left context of the cursor location, which prevents the construction of completion branches at other locations. Good practical results were obtained with an offset of 50 characters to the left of the cursor location. As an example, Figure 5.6 shows the `completion_start` rule and the adapted formulation of the `ForEach` rule of Figure 5.5.

5.5 EVALUATION

Parse error recovery is essential to provide editor services on erroneous input. Especially challenging is the content completion service, which almost exclusively targets incomplete programs. In Section 5.4.4 we discussed the strengths and limitations of our current approach with respect to content

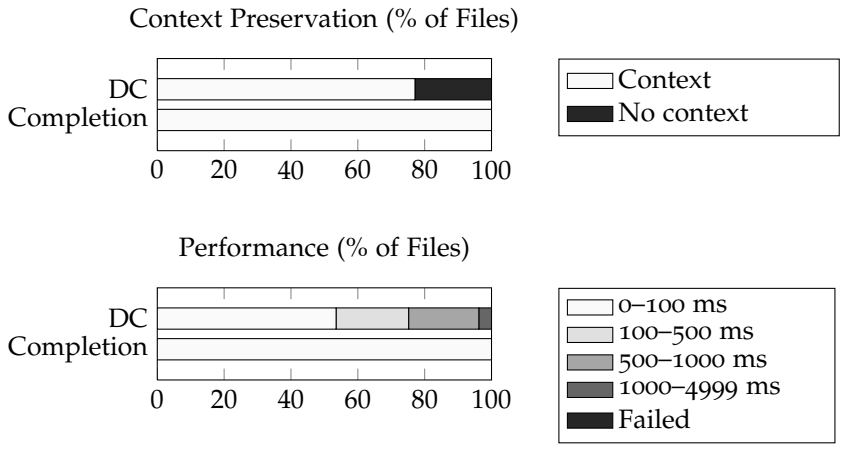


Figure 5.7 Context preservation and performance (recovery times) of the Stratego-Java grammar extended with insertion and deletion recovery rules (**DC**), and extended with completion rules (**Completion**).

completion. To overcome the limitations, we introduced a technique to automatically derive special completion rules that are applied near the cursor location. In this section we evaluate how well the current approach (deletion and insertion rules) serve the purpose of content completion, and how the completion rules improve on this.

We evaluated completion recovery on a set of 314 generated test programs that simulate the scenario of a programmer triggering the content completion service. Accurate completion suggestions require that the syntactic context, the term where completion is requested, is available in the recovered tree. To evaluate the applicability with respect to content completion, we distinguish between recoveries that preserve the syntactic context required for content completion and those that do not.

Figure 5.7 shows the results for our recovery technique with and without the use of completion recovery. Using the original approach (with the DC rule set, i.e., insertion and deletion rules), the syntactic context was preserved in 77 percent of the cases, which shows that the recovery approach is useful for content completion, but is prone to unsatisfactory recoveries in certain cases. Furthermore, as shown by the performance diagram, recovering large incomplete constructs can be inefficient since it requires many deletion and insertion rule applications.

Both problems are addressed by the completion recovery technique, which is specifically designed to recover syntax errors that occur at the cursor location and involve incomplete language constructs. Figure 5.7 shows the results for the completion recovery strategy of Section 5.4.4, using a permissive grammar with the DC rule set plus completion rules. Using this strategy, the syntactic context is preserved in all cases, without noticeable time overhead. The low recovery times are a consequence of the (adapted) runtime support

that exploits the fact that the incomplete construct is located at the cursor location.

A disadvantage of the completion rules is that they significantly increase the size of the grammar, which can negatively affect the parsing performance for syntactically correct inputs. We compared parse times of syntactically correct inputs for the DC/Completion grammar with parse times for the DC grammar, and measured an overhead factor of 1.2. Given that completion rules are highly effective and essential for the content completion functionality, this overhead seems acceptable.

5.6 CONCLUSION

The SGLR parsing algorithm implements scannerless, generalized parsing which is essential for parsing composite languages. In Chapter 2 and 3 we presented three different language independent techniques for syntax error recovery, that we combined in a multi-staged recovery approach for SGLR. The present chapter described the integration of this approach in Spoofax (Kats and Visser, 2010), an environment for developing languages and accompanying IDE support.

Error recovery is crucial for interactive editing, since it allows editor services to provide feedback on syntactically incorrect programs. The editor services operate on the recovered AST which provides a speculative interpretation of the intended program. By ensuring that the recovered AST is well-formed, separation of concerns can be achieved. Error recovery is purely performed by the parser, while the editor services take as input a well-formed AST that represents a syntactically correct program.

Editor services should behave robustly in case the recovery technique fails to construct an interpretation for (a part of) the input program. We developed techniques to provide the programmer with as much feedback as possible. Instead of scannerless highlighting, which requires a parse tree, fall-back syntax highlighting based on a lexical analysis is used to highlight the keywords in the source text. Furthermore, the user is presented with feedback of errors up to the point of where the parser fails, in addition, the failure location is reported to the user.

While other editor services should behave robustly in the presence of syntax errors, the content completion service almost exclusively targets towards incomplete programs. Essential for this service is the interpretation of the syntactic structure near the completion request location. In this chapter we developed an additional technique to recover the local context of the completion request location. Evaluation of this technique showed that it indeed improves the recovery in the content completion scenario.

Part II

Language-Parametric Refactoring Techniques

Source Code Reconstruction

6

ABSTRACT

Transformations and semantic analysis for source-to-source transformations such as refactorings are most effectively implemented using an abstract representation of the source code. An intrinsic limitation of transformation techniques based on abstract syntax trees is the loss of layout, i.e., whitespace and comments. This is especially relevant in the context of refactorings, which produce source code for human consumption. In this chapter, we present an algorithm for fully automatic source code reconstruction for source-to-source transformations. The algorithm preserves the layout of the unaffected parts and reconstructs the layout of the affected parts, using a set of clearly defined heuristic rules to handle comments.

Syntactic sugar enriches a programming language with syntactic constructs that express functionality that can already be expressed in the core syntax. The new syntactic constructs make the language "sweeter" for programmers to use. However, the additional constructs complicate the implementation of transformations and analyses. Therefore, transformations and analyses are sometimes preceded by a desugaring transformation that maps sugared constructs into equivalent constructs in the core syntax. Desugaring considerably simplifies the implementation of refactorings, however, the sugared constructs must be restored in the source code that results after the refactoring. At the end of this chapter, we present an extended version of the text reconstruction algorithm that preserves the original syntactic constructs.

6.1 INTRODUCTION

The successful development of new languages is currently hindered by the high cost of tool building. Developers are accustomed to the integrated development environments (IDEs) that exist for general purpose languages, and expect the same services for new languages. For the development of domain-specific languages (DSLs) this requirement is a particular problem, since these languages are often developed with fewer resources than general purpose languages.

Language workbenches aim at reducing that effort by facilitating efficient development of IDE support for software languages (Fowler, 2005). Examples of language workbenches include EMFText (Heidenreich et al., 2009), MontiCore (Krahn et al., 2008; Grönniger et al., 2008), TCS (Jouault et al., 2006), Xtext (Efftinge and Voelter, 2006), ASF+SDF Meta-Environment (Klint, 1993; van den Brand et al., 2001), IMP (Charles et al., 2007, 2009), Rascal (van der Storm, 2011; Klint et al., 2009), and Spoofax (Kats and Visser, 2010). The

Spoofax language workbench generates a complete implementation of an editor plugin with common syntactic services based on the syntax definition of a language in SDF (Visser, 1997c). Services that require semantic analysis and/or transformation are implemented in the Stratego transformation language (Bravenboer et al., 2008). We are extending Spoofax with a framework for the implementation of refactorings.

Refactorings are transformations applied to the source code of a program. Source code has a formal *linguistic structure* (de Vanter, 2001) defined by the programming language in which it is written, which includes identifiers, keywords, and lexical tokens. Whitespace and comments form the *documentary structure* (de Vanter, 2001) of the program that is not formally part of the linguistic structure, but determines the visual appearance of the code, which is essential for readability. A fundamental problem for refactoring tools is the informal connection between linguistic and documentary structure.

Refactorings transform the formal structure of a program and are specified on the abstract syntax tree (AST) representation of the source code, also used in the compiler for the language. Compilers translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code), which is intended for consumption by machines. In the context of compilation, the layout of the output is irrelevant. Thus, compiler architectures typically abstract over layout. Comments and whitespace are discarded during parsing and are not stored in the AST.

In the case of refactoring, the result of the transformation is intended for human consumption. Contrary to computers, humans consider comments and layout important for readability. Comments explain the purpose of code fragments in natural language, while indentation visualizes the hierarchical structure of the program and new lines helps to clarify the connections between code blocks. A refactoring tool that loses all comments and changes the original appearance of the source code completely is not useful in practice.

The loss of comments and layout is an intrinsic problem of transformation techniques when they are applied to refactorings. To address the concern of layout preservation, these techniques use layout-sensitive pretty-printing to construct the textual representation (Kitlei et al., 2009; Kort and Lämmel, 2003; Li et al., 2006; Lohmann and Riedewald, 2003; van den Brand and Vinju, 2000; de Jonge, 2002). Layout is stored in the syntax tree, either in the form of special layout terms or in the form of tree annotations. After the transformation, the new source code is reconstructed entirely by pretty-printing (unparsing) of the transformed tree. This approach is promising because it uses language-independent techniques. However, preservation of layout is still problematic. Known limitations are imperfections in the whitespace surrounding the affected parts (indentation and inter-token layout), and the handling of comments, which may end up at the wrong locations. The cause of these limitations lies in the orthogonality of the linguistic and documentary structure; projecting documentary structure onto linguistic structure loses crucial information (de Vanter, 2001).

A related problem with transformation techniques is the preservation of syntactic sugar. Syntactic sugar enriches a language with constructs expressing functionality that can already be expressed in the base language. These new syntactic constructs make the language "sweeter" for programmers to use. Desugaring is a step in the transformation process that transforms a syntax tree into an equivalent tree in the core syntax. The specification of refactorings is considerably simplified by desugaring, since the transformation and the semantic analysis only need to be implemented on the core syntax. However, the syntactic sugar must be restored in the result of the refactoring; the language constructs used in the refactored code must be the same as in the original code.

In this chapter, we address the limitations of AST-based approaches to source code reconstruction with an approach based on automated text patching. A text patch is an incremental modification of the original text, which can consist of a deletion, insertion or replacement of a text fragment at a given location. The patches are computed automatically by comparing the terms in the transformed tree, with their originating term in the original or desugared tree. The changes in the abstract terms are translated to text patches. The text patching algorithm uses origin tracking as a technique to relate transformed terms to original terms, and original terms to text positions (van Deursen et al., 1993).

Automated text patching offers more flexibility regarding layout handling compared to the pretty-print approach. We specify a layout adjustment strategy that corrects the whitespace at the beginning and end of the changed parts, and migrates comments so that they remain associated with the linguistic structures to which they refer. The layout adjustment strategy uses explicit, separately specified layout handling rules that are language independent and based on a local style analysis.

The chapter provides the following contributions:

- A text reconstruction algorithm that preserves the layout and syntactic sugar of the original source text.
- A formal analysis of the layout preservation problem, including correctness and preservation proofs for the reconstruction algorithm.
- A set of clearly defined heuristic rules to determine the connection of layout with the linguistic structure.

We start with an example and a formalization of the problem of layout preservation in Section 6.2. Section 6.3 outlines our approach. Origin tracking is introduced in Section 6.4. Section 6.5 explains the basic reconstruction algorithm, for which we prove correctness and preservation properties in Section 6.6. The algorithm is refined with layout adjustment and comment heuristics in Section 6.7. Preservation of syntactic sugar is the topic of Section 6.8. Finally, in Section 6.9 we evaluate our approach on a set of examples, while in Section 6.10 we discuss limitations with the current implementation of our approach.

```
entity User {
  name : String
  //account info
  pwd :  String //6ch
  user :  String
  expire : Date
}

/*Blog info*/
entity Blog { ... }
```

Figure 6.1 Concrete syntax before the refactoring.

```
entity User {
  name : String
  account : Account
  expire : Date
}

entity Account {
  //account info
  pwd : String //6ch
  user : String
}

/*Blog info*/
entity Blog { ... }
```

Figure 6.2 Concrete syntax after the refactoring.

```
[EntityNoSuper(
  "User"
, [PropNoAnno("name", "String")
  ,PropNoAnno("pwd", "String")
  ,PropNoAnno("user", "String")
  ,PropNoAnno("expire","Date")])
,EntityNoSuper("Blog", [...])]
```

Figure 6.3 Abstract syntax tree before the refactoring.

```
[EntityNoSuper(
  "User"
, [PropNoAnno("name", "String")
  ,PropNoAnno("account","Account")
  ,PropNoAnno("expire", "Date")])
,EntityNoSuper(
  "Account"
, [PropNoAnno("pwd", "String")
  ,PropNoAnno("user", "String")])
,EntityNoSuper("Blog", [...])]
```

Figure 6.4 Abstract syntax tree after the refactoring.

6.2 PROBLEM ANALYSIS

In this section we discuss the layout preservation problem for refactoring transformations. First, we illustrate the layout preservation problem with an example refactoring for WebDSL, a domain-specific language for web applications (Visser, 2007). Next, we provide a more formal analysis of the problem.

6.2.1 *Motivating Example*

Program fragments have a concrete representation (Figure 6.1) and an abstract representation (Figure 6.3) constructed by the parser in the form of an abstract syntax tree. The concrete syntax representation is used by humans to edit the program; comments and whitespace are essential for readability. The abstract syntax representation is used by language tools. Abstract syntax trees represent the formal structure of the program, abstracting from comments and whitespace.

Automatic refactorings are typically defined as transformations on abstract syntax trees. The structural representation of the program is necessary to reliably perform the analyses and transformations needed for correct application. Moreover, abstracting from the arbitrary layout of the source code simplifies the specification of the refactoring. As an example, Figure 6.4 shows the result of applying the Extract entity refactoring transformation on the abstract

```
entity User : {
  name : String ()
  pwd : String () //6ch
  user : String ()
  expire : Date ()
}

/*Blog info*/
entity Blog : { ... }
```

Figure 6.5 Concrete syntax fragment after desugaring.

```
[Entity(
  "User"
, None()
, [ Property("name", "String", [])
  , Property("pwd", "String", [])
  , Property("user", "String", [])
  , Property("expire", "Date", [])
  ])
,Entity("Blog", None(), [...])]
```

Figure 6.6 Abstract syntax tree after desugaring.

syntax tree of Figure 6.3.

After the abstract transformation, the consistency between concrete and abstract syntax must be restored while preserving the original layout. The required source code modifications are non trivial. First, the result must be a correct textual representation of the transformed program. Second, the layout of the unaffected program fragments must be preserved, while the layout of the affected fragments must adopt the formatting style used in the rest of the program. Finally, comments must be migrated so that they keep attached to their related program fragments.

As an illustration, Figure 6.2 shows the textual result after applying the Extract entity refactoring on the program fragment shown in Figure 6.1. A new entity (`Account`) is created from the selected properties, and inserted after the `User` entity. The selected properties are replaced by a new property that refers to the extracted entity. The layout of the affected constructs is adjusted to conform to the style used locally, in the adjacent constructs. In particular, indentation and a separating empty line are added to the inserted entity fragment. Comments remain attached to the code constructs to which they refer. Thus, the comments in the selected region are moved together with the selected properties. Furthermore, the comment `/*Blog Info*/` still precedes the `Blog` entity.

Sugar Preservation

Syntactic sugar provides new language constructs that express functionality that can already be expressed in the core language. The `EntityNoSuper` and `PropNoAnno` constructs of Figure 6.1 and 6.3 are in fact syntactic sugar for other, more fundamental, constructs that are part of the WebDSL core language. Figure 6.5 and 6.6 show the "sugar-free" versions of the example fragment in concrete, respectively abstract syntax. The source code fragments in Figure 6.1 and Figure 6.5 express the same functionality, however, the desugared version (Figure 6.5) is more verbose and therefore less attractive for humans to read and write.

At the other hand, sugared constructs complicate the specification of transformations and analysis, since all syntactic synonym constructs must be taken into account. For this reason, transformations and analysis are sometimes preceded by a desugaring step. Desugaring is an AST transformation that transforms "sugared" constructs in the enriched syntax into equivalent con-

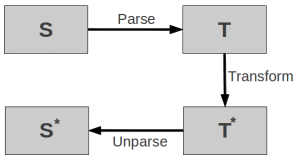


Figure 6.7 Unparsing.

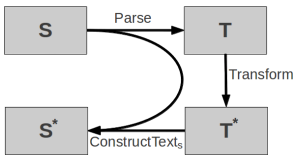


Figure 6.8 Text reconstruction.

structs in the core syntax. The implementation of refactorings is considerably simplified by desugaring, since the transformation and the semantic analysis only need to be implemented on the core syntax. However, this introduces an additional problem to layout preservation; all sugared constructs must be restored in the source code that results after the refactoring.

6.2.2 *Correctness and Preservation Criteria*

The refactoring transformation applied to the AST results in a modified abstract syntax tree. The AST modifications must be propagated to the concrete source text in order to restore the consistency between the concrete and abstract representation. Figure 6.7 illustrates the idea. S and T denote the concrete and the abstract representation of the program, the Parse function maps the concrete representation into the abstract representation, while Transform applies the transformation to the abstract syntax tree. To construct the textual representation of the transformed AST, an Unparse function must be implemented that maps abstract terms to strings.

The Parse function is surjective, so for each well-formed abstract syntax term t, there exists at least one string that forms a textual representation of t. An Unparse function can be defined that constructs such a string (van den Brand and Visser, 1996). The Parse function in general is not injective; strings with the same linguistic structure but different layout are mapped to the same abstract structure, that is $\exists s : \textsc{Unparse}(\textsc{Parse}(s)) \neq s$. It follows that layout preservation cannot be achieved by a function that only takes the abstract syntax as input, without having access to the original source text.

In the context of refactoring, it is required that the layout of the original text is preserved. A text reconstruction function that maps the abstract syntax tree to a concrete representation must take the original text into account to preserve the layout (Figure 6.8). The text reconstruction function ConstructText$_s$ in Figure 6.8 has information about the mapping from terms in T^* to text fragments in S. The access to this information is implemented in the form of a function OriginTerm and a function OriginText which maps respectively transformed terms to original terms, and original terms to concrete text fragments. These functions are in fact functional parameters of the text reconstruction function. For brevity we use the subscript $_s$ to express the dependency on origin information. The mapping from terms to origin fragments is explained in more detail in Section 6.4.

We define the following two criteria for the text reconstruction function:

Correctness. $\textsc{Parse}(\textsc{ConstructText}_s(\textsc{Transform}(\textsc{Parse}(s)))) = \textsc{Transform}(\textsc{Parse}(s))$

Preservation. $\textsc{ConstructText}_s(\textsc{Parse}(s)) = s$

The correctness criterion states that text reconstruction followed by parsing is the identity function on the AST after transformation. The preservation criterion states that parsing followed by text reconstruction returns the original source text for unaffected constructs. Preservation as defined above only covers the identity transformation on terms. Section 6.6.2 gives a more precise criterion that defines preservation in the context of (non-trivial) transformations. Section 6.7 discusses layout adjustment and comment migration as additional requirements for text reconstruction.

Refactoring transformations are preferable implemented on desugared abstract syntax trees, since in that case the transformation and the semantic analysis only needs to be implemented on the core syntax. However, the syntactic sugar must be restored in the source code that results after the refactoring. Preservation of syntactic sugar is problematic since essential information about the original syntactic variant is lost after desugaring. We define the following refined versions of the correctness and preservation criteria that apply to desugared trees:

Correctness. $\textsc{Parse}(\textsc{ConstructText}_s(\textsc{Transform}(\textsc{Desugar}(\textsc{Parse}(s))))) = \textsc{Transform}(\textsc{Desugar}(\textsc{Parse}(s)))$

Preservation. $\textsc{ConstructText}_s(\textsc{Desugar}(\textsc{Parse}(s))) = s$

The layout preservation problem falls in the wider category of view update problems. Foster et al. (2007) define a semantic framework for the view update problem in the context of tree structured data. They introduce lenses, which are bi-directional tree transformations. In one direction (Get), lenses map a concrete tree onto an abstract tree, in the other direction (Putback), they map a modified abstract tree, together with the original concrete tree onto a correspondingly modified concrete tree. A lens is well-behaved if and only if the Get and Putback functions obey the following laws: $\textsc{Get}(\textsc{Putback}(t, s)) = t$ and $\textsc{Putback}(\textsc{Get}(s), s) = s$. These laws resemble our correctness and preservation criterion. That is, the bi-directional transformation Parse and $\textsc{ConstructText}_s$ behaves like a lens. Instead of taking the original input as a parameter, the $\textsc{ConstructText}_s$ function has access to refined information about how abstract terms map to concrete terms; this allows it to fullfill preservation conditions that are a bit stronger than what is typically required by a lens.

6.2.3 *Summary*

We identified the following aspects that are important for the problem of layout preservation in refactoring transformations:

- Correctness: the reconstructed source code must correctly represent the result of the refactoring transformation.

- Layout preservation: the layout of the unaffected code fragments must be preserved in the reconstructed source code.
- Formatting of new elements: new elements must be formatted in accordance with the formatting style used in the rest of the program.
- Whitespace adjustment: the spacing (indentation and separating newlines) of the affected fragments must be adjusted so that it conforms to the spacing conventions used in the original program.
- Comment migration: comments must be migrated so that they keep attached to their related code constructs.
- Sugar preservation: the refactored program must preserve the syntactic variations used in the original program.

In the coming section we discuss how we address these aspects in our approach.

6.3 APPROACH

Given a refactoring transformation applied to the abstract syntax tree of a program, in this section we outline our approach to reconstruct the source text of the resulting program while preserving the layout of the original program.

To access information about the layout used in the original program, we use *origin tracking* as a technique to access original terms and their corresponding text fragments from the resulting terms in the AST after transformation. The technique of tracking origin information is described in Section 6.4.

Given a transformed abstract syntax tree with origin information attached to its terms, we reconstruct the source text by a recursive algorithm. The algorithm reconstructs the text of unaffected terms from their original text fragments, applying changes in the subterms as text patches. Newly inserted terms that lack a suitable origin term are reconstructed by pretty-printing, whereby text fragments of sub terms are reconstructed from their origin fragments (if any exists). The algorithm is described in Section 6.5, while Section 6.6 proves the correctness of the algorithm and its capability to preserve the layout of unaffected code constructs.

An additional requirement for layout preservation is the correct treatment of spacing and comments at the edges of affected code constructs. We extend the text reconstruction algorithm with a method to adjust the spacing of affected code constructs based on information retrieved from a local analysis of the layout style in the original source text. In addition, we formulate a set of heuristic rules that guide the treatment of comments so that they keep attached to the code fragments they refer to. Whitespace adjustment is discussed in Section 6.7, while Section 6.7.1 presents our set of heuristic rules.

In case the refactoring transformation is preceded by a desugaring transformation, layout preservation techniques face the additional challenge to restore the original syntactic variation of desugared constructs. We extend the text

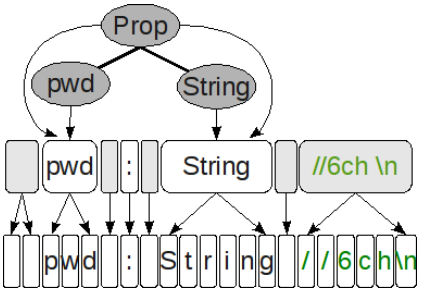


Figure 6.9 Internal representation.

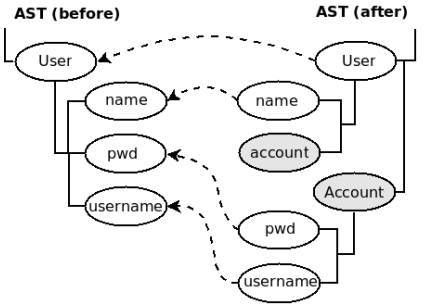


Figure 6.10 Origin tracking.

reconstruction algorithm so that it is also applicable to transformations on desugared ASTs. The main idea is to compare terms in the resulting AST with their origin terms in the desugared AST instead of their origin terms in the AST that results after parsing. Section 6.8 discusses the adaptions we made to meet the requirement of sugar preservation.

6.4 ORIGIN TRACKING

The text reconstruction algorithm proposed in this chapter requires an infrastructure for preserving origin information. This section describes origin tracking as a technique to relate terms in the transformed tree to terms in the original tree, and terms in the original tree to fragments in the source code.

Figure 6.9 illustrates the internal representation of the source code. The program structure is represented by an abstract syntax tree (AST). Each term in the AST keeps a reference to its leftmost and rightmost token in the token stream, which in turn keep a reference to their start and end offset in the character stream. Epsilon productions are represented by a token for which the start- and end- offset are equal. This architecture makes it possible to locate abstract terms in the source text and retrieve the corresponding text fragment. The layout structure surrounding the text fragment is accessible via the token stream, which contains layout and comment tokens.

When the AST is transformed during refactoring, location information is automatically propagated through *origin tracking* (Figure 6.10, dashed line arrows). Origin tracking is a general technique which relates subterms in the resulting tree back to their originating term in the original tree. The rewrite engine takes care of propagating origin information, such that terms in the new tree point to the term from which they originate. Origin tracking is introduced by Van Deursen et al. in (van Deursen et al., 1993), and implemented in Spoofax (Kats et al., 2010a).

For the purpose of syntactic sugar preservation, we extended the origin tracking technique to also track desugared origin terms. After the desugaring stage, all terms in the desugared tree are marked as desugared origins and tracked during the subsequent transformation steps. As a result, terms in the

transformed tree can access their originating term in the original tree, as well as their originating term in the desugared tree.

We implemented a library for retrieving origin information. The library exposes the original term constructed by the parser, the desugared version of the original term, the associated source code fragment, and details about surrounding layout such as indentation, separating whitespace and surrounding comments.

6.5 TEXT RECONSTRUCTION ALGORITHM

In this section we describe our basic reconstruction algorithm which reconstructs the text of unmodified code structures from the original source text, falling back on pretty-printing for newly inserted code structures.

6.5.1 *Notation*

Terms We introduce a formal notation for terms in concrete and abstract syntax, which stresses the correspondence between both representations. Given a grammar G which we assume to be non-ambiguous. Let S_G be the set of strings that represent concrete syntax (sub)terms, and let T_G be the set of well-formed abstract syntax (sub)terms. We use the following notation for term structures: $(t,\ [t_0\ ...\ t_k]) \in T_G$ denotes a term $t \in T_G$ with direct subterms $[t_0\ ...\ t_k] \in T_G$. Equivalently, $(s,\ [s_0\ ...\ s_k]) \in S_G$ means a string $s \in S_G$ with substrings $[s_0\ ...\ s_k] \in S_G$, so that each s_i parses to an abstract term $t_i \in T_G$, and s parses to a term $(t,\ [t_0\ ...\ t_k]) \in T_G$. Terms are characterized by their signature, consisting of the constructor name and the number of direct subterms. When the signature of a term is important, the relevant information is added in superscript, e.g., $x^{(N,\ k)}$ or $(x^N,\ [x_0\ ...\ x_k])$. Finally, for list terms[1] the notation $[x_0\ ...\ x_k]$ is used as a short notation for $(x^{[]},\ [x_0\ ...\ x_k])$, leaving out the list constructor term.

Operations We define the following operations on S_G and T_G, using the subscripts $_S$ and $_T$ to specify on which term representation the operation applies. Given a term $(x,\ [x_0\ ...\ x_k])$ with subterm x_i, then $\mathrm{R}(x_i,\ x_{new})(x)$ replaces the subterm x_i with a new term x_{new} in the term x. In case x is a list, additional operations are defined for deletion and insertion. $\mathrm{D}(x_i)(x)$ defines the deletion of the subterm x_i, while $\mathrm{IB}(x_i,\ x_{new})(x)$ and $\mathrm{IA}(x_i,\ x_{new})(x)$ define the insertion of x_{new} before (IB) or after (IA) the element x_i. Remark: we assume here that we have a notion of term identity. In our case, the term identity in the tree that results after parsing is given by the origin relation. That is, two terms are equal if and only if they are structurally equivalent and they are associated to the same source code fragment defined by its start and end offset.

[1] In SDF2 list terms are defined in the grammar as TRM*, respectively TRM+ for non-empty lists. {TRM ","}* denotes a list of TRMs separated by a comma.

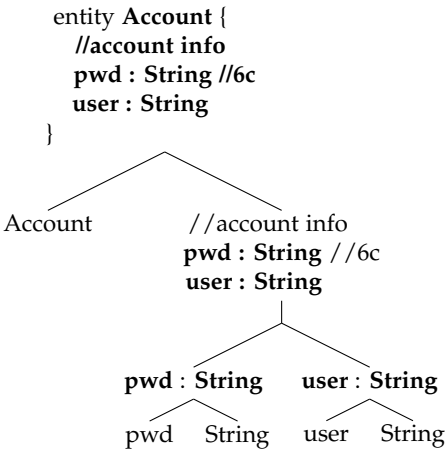


Figure 6.11 Reconstruction example.

Functions The following functions are defined for elements in S_G or T_G. PRS : $S_G \rightarrow T_G$ the parse function that maps concrete terms onto their corresponding abstract terms, ORTXT : $T_G \rightarrow S_G$ the function that returns the origin source fragment of a parsed term, PP : $T_G \rightarrow S_G$ a pretty-print function that constructs a string representation for elements in T_G, and ORTRM : $T_G \rightarrow T_G$ the function that returns the origin term of a term in the transformed tree.

6.5.2 *Algorithm*

We define an algorithm that reconstructs the source code after the refactoring transformation. The reconstruction algorithm implements a postorder traversal of the transformed abstract syntax tree, constructing the text fragment of the visited term from the text fragments of its subterms that were already constructed in the traversal. Figure 6.11 illustrates the reconstruction of the account entity. The substrings printed in bold are constructed by traversing the subterms, while the surrounding characters are either retrieved from the origin fragment, or constructed by pretty printing.

Figure 6.12 shows the algorithm in pseudo code. CONSTRUCTTEXT$_S$(*term*) takes an abstract syntax term as input and constructs a string representation for this term. Three cases are distinguished; reconstruction for constructor terms (l. 1-5), reconstruction for lists (l. 6-11), and pretty printing in case the origin term is missing, i.e., when a term is newly created in the transformation (l. 12-14). We discuss those cases.

If an origin term with the same signature exists (l. 2-3), the text fragment is reconstructed from the original text fragment, corrected for possible changes in the subterms. The function $R_S(s'_i, s_i)$: *String* $\rightarrow$ *String* subsequently replaces the substrings that represent original subterms with substrings for the new subterms constructed by a recursive call to CONSTRUCTTEXT$_S$ (l. 5). The (relative) offset is used to locate the text fragment associated to the original

ConstructText$_s$($term$) ▷ Abbreviated as CT

```
1  if
2      (t^N, [t_0 ... t_k]) ← term
3      (t'^N, [t'_0 ... t'_k]) ← OrTrm(term)
4  then  ▷ Term with origin info
5      return R_S(OrTxt(t'_0), CT(t_0)) ∘ ··· ∘ R_S(OrTxt(t'_k), CT(t_k))
                ∘ OrTxt(OrTrm(term))
6  elseif
7      [t_0 ... t_k] ← term
8      [t'_0 ... t'_j] ← OrTrm(term)
9  then  ▷ List term with origin info
10     [Mod_0 ... Mod_z] ← ListChanges(OrTrm(term), term)
11     return Mod_0 ∘ ··· ∘ Mod_z(OrTxt(OrTrm(term)))
12 else  ▷ New constructed term
13     (t^N, [t_0,...t_k]) ← term
14     return PP ∘ R_T(t_0, CT(t_0)) ∘ ··· ∘ R_T(t_k, CT(t_k))(term)
```

ListChanges($oldLst, newLst$)

```
15 lcs ← LCS(oldLst, newLst)
16 diffs ← []
17 for each (oldEl, newEl) ∈ lcs do
18     diffs ← R_S(OrTxt(oldEl), CT(newEl)) :: diffs
19 for each oldEl ∈ oldLst :  (oldEl, _) ∉ lcs do
20     diffs ← D_S(OrTxt(oldEl)) :: diffs
21 unmatched ← []
22 for each newEl ∈ newLst do
23     if (_, newEl) ∉ lcs then
24         unmatched ← unmatched ::: [newEl]
25     if (oldEl, newEl) ∈ lcs and unmatched ≠ [] then
26         diffs ← IB_S(OrTxt(oldEl), CT(unmatched)) :: diffs
27         unmatched ← []
28 if unmatched ≠ [] then
29     diffs ← IA_S(OrTxt(oldLst), CT(unmatched)) :: diffs
30 return diffs
```

LCS($oldLst, newLst$)

```
31 ▷ Constructs a longest common subsequence based on origin matching.
     The result is a list of tuples (oldEl, newEl) with oldEl = OrTrm(newEl).
```

Figure 6.12 Pseudo code reconstruction algorithm.

subterm ($\textsc{OrTxt}(t'_i)$), this detail is left out of the pseudo code.

Text reconstruction for list terms (l. 6-11) implements the same idea, except that the changes in the subterms may include insertions and deletions. The textual modifications are calculated by a differencing function (ListChanges) and subsequently applied to the original list fragment (l. 11). The function ListChanges matches elements of the new list with their origin term in the original list (LCS, l. 15); the matched elements are returned as replacements (l. 18), the unmatched elements of the old list form the deletions (l. 20), while the insertions consist of the unmatched elements in the new list (l. 26, l. 29). It is crucial that the elements of the new list are correctly matched with related elements from the old list, since they automatically adopt the surrounding layout at the position of the old term, which may contain explanatory comments.

New terms are reconstructed by pretty-printing (l. 12, l. 14). To preserve the layout of subterms associated with an origin fragment, the pretty print function is applied after replacing the subterms with their textual representation, constructed recursively by a call to $\textsc{ConstructText}_s$ (l. 14).

6.5.3 *Optimizations*

The algorithm shown in Figure 6.12 represents a simplified version of the actual algorithm that we implemented for text reconstruction. The algorithm discussed so far illustrates the reconstruction idea and allows reasoning about correctness and preservation properties. However, it also puts some limitations with respect to robustness and applicability. First, the algorithm is not robust in case origin information is accidentally lost during the transformation. That is, terms that are not associated to an origin term are pretty-printed, even if the structure has not changed at that point in the program. Secondly, the algorithm requires that all origin terms are associated to a source fragment. This is the case for origin terms that result after parsing, but not necessarily for desugared origin terms which result after a desugaring transformation. Desugared origin terms are used for sugar preservation as discussed in Section 6.8.

To overcome these limitations, we implemented a slightly different version of the algorithm that only replaces fragments associated to terms that have actually been changed during the transformation. That is, given a new term with an associated origin term plus origin fragment, text reconstruction proceeds as follows. First, the new term is structurally compared to the original term to detect all (top) changes in the descendant terms. Next, all changes in the descendant terms are translated to textual changes and bubbled up as changes on the origin fragment of the new term. Finally, these changes are applied to the fragment under construction. Figure 6.13 shows the function that collects the textual changes by traversing the subterms. Nested changes are handled via a recursive call to $\textsc{ConstructText}_s$ (l. 12). The collected textual changes are applied to the original fragment in Figure 6.12, line 5 and line 18.

```
SubtermChanges(oldTerm, newTerm)
 1  if
 2      (t^N, [t_0 ... t_k]) ← newTerm
 3      (t'^N, [t'_0 ... t'_k]) ← oldTerm
 4  then  ▷ Terms with same signature
 5      return ⋃_{0≤i≤k} SubtermChanges(t'_i, t_i)
 6  elseif
 7      [t_0 ... t_k] ← newTerm
 8      [t'_0 ... t'_j] ← oldTerm
 9  then  ▷ List terms
10      return ListChanges(oldTerm, newTerm)
11  else  ▷ Term Change
12      return [R_S(OrTxt(oldTerm), CT(newTerm))]
```

Figure 6.13 Function that collects textual changes along a frontier of the new term.

This implementation is more robust since lost origin relations only have a negative affect on the result if they occur at terms that are moved during the transformation. Furthermore, changes on (desugared) origin terms that are not associated to a source fragment can be propagated as changes on the parent terms. This makes the algorithm applicable for sugar preservation (Section 6.8).

6.6 CORRECTNESS AND PRESERVATION PROOFS

In this section we prove correctness and preservation properties for the text reconstruction algorithm shown in Figure 6.12. We start by formulating the assumptions, definitions and lemmas that are used in the correctness proof.

The text reconstruction algorithm translates the transformation in the abstract representation to the corresponding transformation in the concrete representation. This translation essentially exploits an assumed homomorphic relation between abstract and concrete terms. We state this homomorphic relation as an assumption. Exceptions to this assumption can occur in practice, for example because of priority and associativity filters, and because of follow restrictions defined in the grammar. The applicability of the homomorphism assumption and techniques to overcome exceptional cases are discussed later in this section.

Assumption 1. *Let* Prs : $S_G \to T_G$ *the parse function as defined in Section 6.5.1.* Prs *is a homomorphism on tree structures.*

Definition. *Given the functions* Prs : $S_G \to T_G$, OrTxt : $T_G \to S_G$, PP : $T_G \to S_G$, OrTrm : $T_G \to T_G$ *as defined in Section 6.5.1. The following properties hold:*

D 1. $\text{OrTrm}(\text{Prs}(s)) = \text{Prs}(s)$

D 2. $\textsc{OrTxt}(\textsc{Prs}(s)) = s$

D 3. $\textsc{Prs}(\textsc{OrTxt}(\textsc{OrTrm}(t))) = \textsc{OrTrm}(t)$

D 4. $\textsc{Prs}(\text{PP}(t)) = t$

D 5. $\text{PP}(s) = s$ *for all string constants* s

Proof. This follows directly from the definition of the functions. □

Lemma. *Let* $\textsc{Prs} : S_G \to T_G$ *the parse function, and assume* $\textsc{Prs} : S_G \to T_G$ *is a homomorphism on tree structures. Then the following equations hold:*

L 1. $\textsc{Prs} \circ \text{R}_S(s'_i, s_i)(s) = \text{R}_T(\textsc{Prs}(s'_i), \textsc{Prs}(s_i)) \circ \textsc{Prs}(s)$

L 2. $\textsc{Prs} \circ \text{D}_S(s'_i)(s) = \text{D}_T(\textsc{Prs}(s'_i)) \circ \textsc{Prs}(s)$

L 3. $\textsc{Prs} \circ \text{IB}_S(s'_i, s_i)(s) = \text{IB}_T(\textsc{Prs}(s'_i), \textsc{Prs}(s_i)) \circ \textsc{Prs}(s)$

L 4. $\textsc{Prs} \circ \text{IA}_S(s'_i, s_i)(s) = \text{IA}_T(\textsc{Prs}(s'_i), \textsc{Prs}(s_i)) \circ \textsc{Prs}(s)$

Proof. This follows from the assumption that Prs is a homomorphism on tree structures. □

6.6.1 *Correctness*

We prove correctness of $\textsc{ConstructText}_s : T_G \to S_G$ (Figure 6.12, abbreviated as CT), assuming that $\textsc{Prs} : S_G \to T_G$ is a homomorphism on tree structures.

Theorem (Correctness). $\forall t \in T_G\ \textsc{Prs}(\text{CT}(t)) = t$

The proof is by induction on tree structures. We distinguish two cases for the leaf terms, dependent on whether an origin term exists with the same signature.

Base case (a). Let $t = (t^N, [])$ a leaf term with origin term $(t^N, [])$.
$\textsc{Prs}(\text{CT}(t)) =^{line\ 5} \textsc{Prs}(\textsc{OrTxt}(\textsc{OrTrm}(t))) =^{D\ 3} \textsc{OrTrm}(t) = (t^N, [])$ □

Base case (b). Let $(t, [])$ a leaf term for which no origin term exists.
$\textsc{Prs}(\text{CT}(t)) =^{line\ 14} \textsc{Prs}(\text{PP}(t)) =^{D\ 4} t$ □

IH. $\textsc{Prs}(\text{CT}(t_i)) = t_i$ *holds for all subterms* t_0 *to* t_k *of a term* $(t, [t_0 \ldots t_k])$.

We now proof the induction step $\textsc{Prs}(\text{CT}(t)) = t$.

Induction step (a). Assuming the induction hypothesis, we first prove a property of text modification operations as applied in the lines 5 and 11.

p 1. *Given a concrete syntax term* s *with a direct subterm* $\textsc{OrTxt}(t'_i)$. *The following holds for modification operations* $\textsc{Mod} \in R, D, IB, IA$.
$\textsc{Prs} \circ \textsc{Mod}_S(\textsc{OrTxt}(t'_i), \text{CT}(t_i))(s) =^{L\ 1,L\ 2,L\ 3,L\ 4}$
$\textsc{Mod}_T(\textsc{Prs} \circ \textsc{OrTxt}(t'_i), \textsc{Prs} \circ \text{CT}(t_i)) \circ \textsc{Prs}(s) =^{D\ 3,IH}$
$\textsc{Mod}_T(t'_i, t_i) \circ \textsc{Prs}(s)$

We prove the induction step for constructor terms (t^N) with origin information below, the proof for list terms follows the same logic. Let $t = (t^N,\ [t_0\ ...\ t_k])$ a term with origin term $t' = (t'^N,\ [t'_0\ ...\ t'_k])$.
$\textsc{Prs} \circ \mathrm{CT}(t) =^{line\ 5}$
$\textsc{Prs} \circ \mathrm{R}_S(\textsc{OrTxt}(t'_0),\ \mathrm{CT}(t_0)) ... \circ \mathrm{R}_S(\textsc{OrTxt}(t'_k),\ \mathrm{CT}(t_k)) \circ \textsc{OrTxt}(t') =^{p\ 1}$
$\mathrm{R}_T(t'_0,\ t_0) \circ ... \circ \mathrm{R}_T(t'_k,\ t_k) \circ \textsc{Prs} \circ \textsc{OrTxt}(t') = D\ 3$
$\mathrm{R}_T(t'_0,\ t_0) \circ ... \circ \mathrm{R}_T(t'_k,\ t_k)(t'^N,\ [t'_0\ ...\ t'_k]) = (t^N,\ [t_0\ ...\ t_k]) = t$ □

Induction step (b). First, we prove a property for the pretty-print function (PP).

p 2. $\textsc{Prs} \circ \mathrm{PP} \circ \mathrm{R}_T(t'_i,\ t_i)(t) =^{D\ 4}$
$\mathrm{R}_T(t'_i,\ t_i)(t) =^{D\ 4}$
$\mathrm{R}_T(\textsc{Prs} \circ \mathrm{PP}(t'_i),\ \textsc{Prs} \circ \mathrm{PP}(t_i)) \circ \textsc{Prs} \circ \mathrm{PP}(t) =^{L\ 1}$
$\textsc{Prs} \circ \mathrm{R}_S(\mathrm{PP}(t'_i),\ \mathrm{PP}(t_i)) \circ \mathrm{PP}(t)$

We now prove the induction step for constructor terms (t^N) that lack origin information. Let $(t,\ [t_0\ ...\ t_k])$ a term for which no origin term exists.
$\textsc{Prs} \circ \mathrm{CT}(t) =^{line\ 14}$
$\textsc{Prs} \circ \mathrm{PP} \circ \mathrm{R}_T(t_0,\ \mathrm{CT}(t_0)) \circ ... \circ \mathrm{R}_T(t_k,\ \mathrm{CT}(t_k))(t) =^{p\ 2}$
$\textsc{Prs} \circ \mathrm{R}_S(\mathrm{PP}(t_0),\ \mathrm{PP} \circ \mathrm{CT}(t_0)) \circ ... \circ \mathrm{R}_S(\mathrm{PP}(t_k),\ \mathrm{PP} \circ \mathrm{CT}(t_k)) \circ \mathrm{PP}(t) =^{L\ 1, D\ 5}$
$\mathrm{R}_T(\textsc{Prs} \circ \mathrm{PP}(t_0),\ \textsc{Prs} \circ \mathrm{CT}(t_0)) \circ ... \circ \mathrm{R}_T(\textsc{Prs} \circ \mathrm{PP}(t_k),\ \textsc{Prs} \circ \mathrm{CT}(t_k)) \circ \textsc{Prs} \circ \mathrm{PP}(t) =^{D\ 4, IH}$ $\mathrm{R}_T(t_0,\ t_0) \circ ... \circ \mathrm{R}_T(t_k,\ t_k)(t) =^{D\ 4} t$ □

6.6.2 *Layout Preservation*

Abstract syntax terms in general have multiple textual representations. These representations differ in the use of layout between the linguistic elements. In addition, small differences may occur in the linguistic elements; typically the use of braces is optional in some cases. We introduce the notion of formatting that covers these differences. Then we prove that the text reconstruction algorithm preserves formatting for terms that are not changed in the transformation, although they may have changes in their subterms or they may have been moved to another position in the abstract syntax tree.

Definition. *Given $(s,\ [s_0\ ...\ s_k]) \in S_G$. The formatting of s is defined as the list consisting of the substring contained in s and preceding s_0, the substrings that appear between the subterms $s_0\ ...\ s_k$, plus the substring contained in s and succeeding s_k.*

Theorem (Layout Preservation). *Let $t \in T_G$ with origin term $\textsc{OrTrm}(t) \in T_G$. If t and $\textsc{OrTrm}(t)$ are constructor terms with the same signature, then $\mathrm{CT}(t)$ and $\textsc{OrTxt}(\textsc{OrTrm}(t))$ have the same formatting. If t and $\textsc{OrTrm}(t)$ are list terms, then the formatting is preserved for sublists for which no child terms are deleted or inserted during the transformation.*

Proof. Let $(t^N,\ [t_0\ ...\ t_k])$ a term with origin term $(t'^N,\ [t'_0\ ...\ t'_k])$, then $\mathrm{CT}(t) = \mathrm{R}_S(\textsc{OrTxt}(t'_0),\ \mathrm{CT}(t_0)) \circ ... \circ \mathrm{R}_S(\textsc{OrTxt}(t'_k),\ \mathrm{CT}(t_k)) \circ \textsc{OrTxt}(\textsc{OrTrm}(t))$. Since R_S only affects the substrings that represent the child terms, the formatting of the parent string is left intact. For list terms: Let $t = [t_0\ ...\ t_k]$ a

list with origin term $\text{ORTRM}(t) = [t_0 \ldots t_l]$, then $\text{CT}(t) = \text{MOD}_{t'_0} \circ \ldots \circ \text{MOD}_{t'_l} \circ \text{ORTXT}(\text{ORTRM}(t))$ $\text{MOD}_{t'_i} \in \{\text{R}_S, \text{D}_S, \text{IB}_S, \text{IA}_S\}$. By definition, the modification functions replace, delete or insert substrings representing child terms. The formatting is preserved for sublists that are not affected by deletions or insertions. □

6.6.3 *Irregularities*

The correctness proof depends on the assumption that parsing is a homomorphism on tree structures. Inspection of grammars for different languages (Mobl (Hemel and Visser, 2011), Stratego (Visser, 2004), SDF (Heering et al., 1989b; Visser, 1997c) and Java) showed that this requirement is met in general, except for some common exceptions. Tree structures in the concrete syntax representation can be ambiguous, in which case the parse result is determined by disambiguation rules. Disambiguation rules can invalidate the homomorphic nature of the parse function. Below we discuss priority and associativity rules, follow restrictions, and preference rules. In addition, we discuss the practical problem of inserting and removing list elements together with a separator or whitespace token. We propose language generic solutions for most of the identified problems. Furthermore, we implemented a hook that allows developers to provide custom text reconstruction strategies for other irregular cases that may occur in a particular grammar.

Priority and associativity Priority and associativity rules are used to disambiguate expression syntax. For instance, the expression $2 * 3 + 4$ is parsed as $(t^{Plus}, [(t^{Mult}, [2, 3]), 4])$, while the alternate parse, $(t^{Mult}, [2, (t^{Plus}, [3, 4])])$ is rejected because of a priority rule that gives higher priority to the t^{Mult} production. In case the latter interpretation is in fact the intended interpretation, the programmer can use brackets, i.e., $2 * (3 + 4)$.

Priority and associativity rules invalidate the homomorphic nature of the parse function. That is, bottom up text reconstruction fails to produce the correct code fragment for $(t^{Mult}, [2, (t^{Plus}, [3, 4])])$ in case $(t^{Plus}, [3, 4])$ is reconstructed as "$3 + 4$" instead of "$(3 + 4)$". This can happen for example in case $(t^{Plus}, [3, 4])$ is the result of an Inline refactoring.

We implemented a language generic solution to handle this situation. The implementation is based on the optimized version of the text reconstruction algorithm as described in Section 6.5.3. First, the text reconstruction algorithm takes as an additional parameter a strategy for adding parentheses terms, $(t^{Brackets}, [expr])$, at the necessary places in the tree. We then apply this strategy to terms of the transformed tree, as well as to their origin terms. We distinguish two cases for expression terms that are "parenthesized" in the transformed tree. In case a bracket term in the transformed tree is compared to a bracket term in the original tree, then these $t^{Brackets}$ terms cancel each other out. This situation only happens in case the origin text already contains brackets. In other cases, the bracket term is pretty printed as "(" + $\text{CONSTRUCTTEXT}_s(expr)$ + ")". The strategy for parentheses insertion is auto-

matically derived from the syntax definition (van den Brand and Visser, 1996; de Jonge, 2000).

Follow restrictions Follow restrictions are used to implement longest match, and to enforce some layout separation between keywords and identifiers. As an example, we look at a rule condition in the Stratego language. The string `where foo` is parsed as the rule condition $(t^{Where}, [(t^{SCall}, [\text{"}foo\text{"}])])$. The leaf nodes of the parse tree are the keyword literal `"where"`, the layout string `" "`, and the identifier `"foo"`. Without any lexical disambiguation, the scannerless parser also parses the string `wherefoo` as the given rule condition. This time the optional layout between `where` and `foo` is left out. Typically, languages forbid the occurrence of an identifier immediately after a keyword. For this reason a follow restriction is added to the grammar, e.g., `"where" -/- [a-zA-Z0-9\-\_]`. The follow restriction indicates that the `"where"` literal may not directly be followed by a character in the range `[a-zA-Z0-9\-\_]`.

Follow restrictions may invalidate the homomorphic nature of the parse function. As a consequence, ConstructText$_s$ may generate a parse error or a construct that was not intended. We illustrate the problem with a transformation on Stratego rule conditions. The rule condition string `where<foo> m` is parsed as the abstract term $(t^{Where}, [(t^{BA}, [(t^{SCall}, [\text{"}foo\text{"}]), t^{Var}, [\text{"}m\text{"}])]])$. Now suppose that this term is transformed into the term $(t^{Where}, [(t^{SCall}, [\text{"}foo\text{"}])])$. The text reconstruction algorithm applies the abstract term change as a textual change that replaces the sub string `<foo> m` with the string `foo` on the parent fragment `where<foo> m`. The result is the string `wherefoo`. However, this is not a valid rule condition because of the follow restriction on the `"where"` keyword. A pragmatic solution that will work in most cases is to insert some additional layout, e.g., a single whitespace character, in case a replacement on the concrete syntax places two keyword or identifier tokens immediately after each other.

Preference rules A third categorie of syntactic disambiguation methods is given by preference attributes, i.e., `avoid` and `prefer`. Preference attributes are used to select a default interpretation among several alternative interpretations. As an example, we look at the notorious "dangling else" construction. The input sentence `if 0 then if 1 then foo else bar` can be parsed as the abstract term $(t^{If}, [0, (t^{IfElse}, [1, \text{"}foo\text{"}, \text{"}bar\text{"}])])$, or alternatively, as the term $(t^{IfElse}, [0, (t^{If}, [1, \text{"}foo\text{"}]), \text{"}bar\text{"}])$. We can select the first interpretation by adding a `prefer` attribute on the production rule for the t^{If} term, or vice-versa, we can select the second interpretation by adding a `prefer` attribute on the production rule for the t^{IfElse} term.

The ConstructText$_s$ function may reconstruct a text fragment with a syntactic ambiguity that is solved by a preference rule. The correctness criterion is violated in case the preference rule selects the interpretation that does not correspond with the reconstructed abstract term. We illustrate this problem for the "dangling else" example discussed above. Suppose that the term $(t^{If}, [0, (t^{IfElse}, [1, \text{"}foo\text{"}, \text{"}bar\text{"}])])$ is obtained after an Inline refactoring that

puts the (t^{IfElse}, [1, "*foo*", "*bar*"]) statement into the body of the (t^{If}, [0, ...]) statement. Text reconstruction for this term results in the text fragment (we ignore the arbitrary layout) `if 0 then if 1 then foo else bar`. It depends on the specified preference rule whether or not the resulting fragment correctly represents the reconstructed term.

A generic solution can not be implemented for correctness issues due to ambiguities that are resolved by preference rules. For some languages, the transformation should simply be rejected because the transformed abstract syntax tree does not have a textual representation, i.e., the abstract tree is not syntactically well-formed. For other languages, the correct interpretation may be enforced by adding a pair of brackets around the inner construct. We leave it to the refactoring implementor to implement a custom solution for this situation. Correctness issues due to preference rules do not seem to occur frequently in real world grammars. For example, the discussed "dangling else" issue only occurs in case the t^{IfElse} production is preferred over the t^{If} production; while in the inspected Java grammar, the `prefer` attribute is placed on the t^{If} production.

Separators Another exception with respect to the homomorphism property concerns separation between list elements. When a list element is inserted (or deleted), it must be inserted (deleted) including some separating whitespace plus a possible separator. Whether a separator is required or not is determined by the list sort. We implemented the following solution. List elements are inserted (and deleted) including their separation with the next (or preceding) element. When a list element is inserted, the separation is retrieved from the original source text in case the origin list has two or more elements, otherwise it is looked up in the pretty-print table.

6.7 LAYOUT ADJUSTMENT

The algorithm shown in Figure 6.12 preserves the layout of the unaffected regions, but fails to manage spacing and comments at the frontier between the changed parts and the unchanged parts. Figure 6.14 shows the result of applying the algorithm to the refactoring described in Section 6.2.1 (Figure 6.1 and 6.2). Comments end up at the wrong location (`//account info`, `/*Blog info*/`), the whitespace separation around the `account` property and the `Account` entity is not in accordance with the separation in the original text, and the indentation of the `Account` entity is disorderly.

The algorithm in Figure 6.12 translates AST-changes to modifications on code structures, but ignores the layout that surrounds these structures. To overcome this shortcoming, we refine the implementation of the algorithm so that whitespace and comments are migrated together with their associated code structures. Layout migration is implemented by using a layout-sensitive version of the OriginText and OriginOffset functions used to access the original text.

To correct the whitespace of reconstructed fragments, language generic layout adjustment functions are implemented that adopt the spacing of the sur-

```
entity User {
  name : String
  //account info

  account : Account expire : Date
}

/*Blog info*/
entity Account {
password : String //6 chars
  username : String
}entity Blog { ... }
```

Figure 6.14 Layout deviation without adjustment.

IBADJUSTED(t_{old}, t_{new})

1 $text \leftarrow$ CONSTRUCTTEXT$_S(t_{new})$
2 $text \leftarrow$ REMOVEINDENT($text$)
3 $text \leftarrow$ ADDINDENT(
  $text$,
  ORIGININDENT(t_{old}))
4 $text \leftarrow$ CONCATSTRINGS([
  $text$,
  SEPARATION(t_{old})])
5 $offset \leftarrow$ OFFSETWITHLO(t_{old})
6 **return** IB$_S$($offset, text$)

Figure 6.15 Layout adjustment function.

rounding code for the inserted fragments. In particular, an inserted construct is indented and separated according to the layout of the adjacent constructs. Figure 6.15 shows the layout adjustment steps for IB$_S$. First, the text is reconstructed with its associated comments (l. 1). Then, the existing indentation is removed, leaving the nesting indentation intact (l. 2). Subsequently, the start indentation at the insert location is retrieved from the adjacent term (t_{old}) and appended to all lines (l. 3). Finally, separation is added to separate the construct from its successor (l. 4). The separation is retrieved by inspecting the layout surrounding t_{old} or by consulting the pretty-print table in case the original list has less than two elements.

6.7.1 *Comment Heuristics*

Comment migration requires a proper interpretation of how comments attach to the linguistic structure, which is problematic because of the informal nature of comments. The use of comments differs, depending on style conventions for a particular language and the personal preference of the programmer. de Vanter (2001) gives a detailed analysis.

Figure 6.16 illustrates the use of comments with different style conventions used in combination. Fragment #1 is a block comment that explains the purpose of the accompanying method. The comment resides in front of its structural referent. This is also the case for the comments in #2a,b,c. However, these comments do not attach to a single structure element, but instead relate to a group of statements. The blank lines that surround these grouped statements are essential in understanding the scope of the comments. Contrary to the previous examples, the line comment in #3 points backwards to the preceding statement. #6 provides an example of a comment in the context of list elements separated by a comma. In this case, the location of the comma determines whether the comment points forward or backward. The

```
/**
 * Processes income data and displays statistics #1
 */
public static void displayStatistics(Scanner input) {
   //Initialize variables #2a
   int    count = 0;   // Number of values #3a
   double total = 0;   // Sum of all incomes #3b

   //Process input values until EOF #2b
   System.out.println("Enter income values");
   while (input.hasNextDouble()) {
      double income = input.nextDouble();
      //System.out.println("processing: " + income); #4
      if(income>=0){
         count++;            // Keep track of count
         total += income;    // and total income #5
      }
   }

   //Display statistics #2c
   double average = calcAverage(count, /*sum*/ total); #6
   System.out.println("Number of values = " + count);
   System.out.println("Average = "  + average);
}
```

Figure 6.16 Different comment styles in a Java source code fragment.

commented-out `println` statement in #4 does not have a structural referent. It can best be seen as lying between the surrounding code elements. Finally, #5 illustrates a single comment that is spread over two lines. A human reader will recognize it as a single comment, although it is structurally split in two separate parts. In this case, the vertical alignment hints at the fact that both parts belong together.

Figure 6.16 makes clear why attaching comments to AST terms is problematic. The connection of comments with abstract terms only becomes clear when taking into account the full documentary structure, including newlines, indentation and separator tokens. Comments can point forward, as well as backward and, purely based on analysis of the tree structure, it is impossible to decide which one is the case. Even more problematic are #2 and #4; both comment lines lack an explicit referent in terms of a single AST term. The former refers to a sublist, while the latter falls between the surrounding terms.

Text reconstruction allows for a more flexible approach towards the interpretation of comments. Instead of a fixed mapping between comments and abstract terms, heuristic rules are defined that interpret the documentary structure around the moved AST terms. Comment heuristics are defined as layout patterns using newlines, indentation, and separators as building blocks (Figure 6.17). If a pattern applies to a given term (or group of terms), the term (group) is considered as the structural referent of the comment(s) that take part in the pattern. The binding heuristics have the following effect on the textual transformation; if a term (group) is (re)moved, all adjacent comments that bind to the term(s) are (re)moved as well. Adjacent comments that do not bind, stay at their original position in the source code. Comments that lie inside the region of the migrated term(s) automatically migrate jointly.

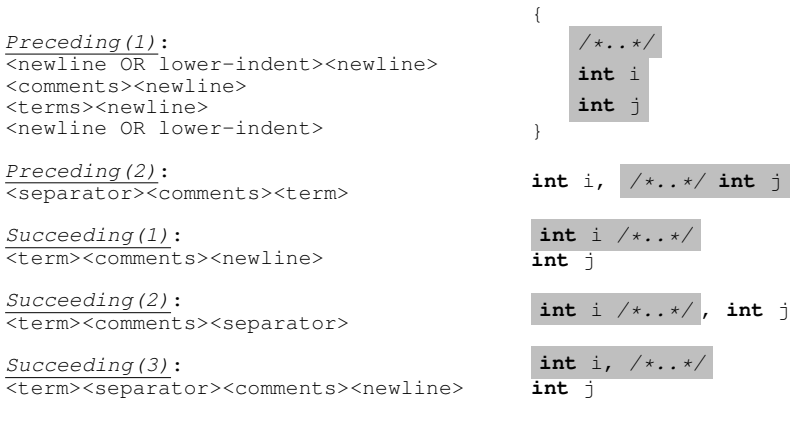


Figure 6.17 Comment patterns.

The patterns in Figure 6.17 handle the majority of comment styles correctly. The comment styles in Figure 6.16 are recognized by the patterns, with the exception of vertical alignment (#5), which is not detected. Preceding(1) binds #1 to the `displayStatistics` method, and #2a,b,c to the statement groups they refer to. #3 is interpreted by Succeeding(1). None of the patterns applies to #4, which indeed neither binds to the preceding nor to the succeeding construct. The comment in #6 is associated with the succeeding construct by application of Preceding(2). Finally, #5 is interpreted as two separate comments associated to their preceding statement, but not recognized as a single comment spread over two lines.

Heuristic rules will never handle all cases correctly; ultimately, it requires understanding of the natural language to decide the meaning of a comment and how it relates to the program structure. While our experience so far suggests that the comment heuristics are adequate, further experience with other languages, other refactorings, and other code bases is needed to determine whether these rules are sufficient.

6.8 SYNTACTIC SUGAR PRESERVATION

Syntactic sugar provides new language constructs that support expression of functionality that can already be expressed in the base language. These new syntactic constructs make the language "sweeter" for programmers to use: things can be expressed more clearly, more concisely, or in an alternative style that someone may prefer. For instance, `if ... then` can be used in stead of `if ... then ... else` in case the `else` branch is empty. Another example is given by `for` loops, which can be systematically replaced with `while` loops.

Desugaring is a step in the transformation process that transforms a syntax tree into an equivalent tree in the core syntax. Desugaring is typically im-

plemented by a top-down traversal which repeatedly applies normalization rules to terms in the tree. These rules can be simple local rules, but sometimes more complex rules are applied that require context information. The desugaring ensures that later stages of the transformation only need to deal with one syntactic variant.

The specification of refactorings is considerably simplified by desugaring, since the transformation and the semantic analyses only need to be implemented on the core syntax. However, the syntactic sugar must be restored in the result of the refactoring. The language constructs used in the refactored code should be the same as in the original code, otherwise, programmers will argue that the sources were alienated which violates the requirement of preservation. Preservation of syntactic sugar is problematic in pure AST based approaches, since it requires access to the original syntax.

In our approach, access to the original syntax is provided by origin tracking. The remaining challenge is to distinct between term modifications made in the desugaring transformation and modifications that form the actual refactoring transformation. The former must be ignored, while the latter must be applied as textual modifications on the source text. We adjust the layout preservation algorithm so that it preserves syntactic sugar. We discuss the adaptations we made, starting from the algorithm of Figure 6.12 with the improvements discussed in Section 6.5.3 and shown in Figure 6.13.

6.8.1 *Adaptations for Sugar Preservation*

To identify the term changes caused by the refactoring transformation, we compare the reconstructed terms to their *desugared* origin term instead of their base origin term. The found term changes are translated into textual changes by accessing the original text fragments via the base origin terms. This ensures that *1)* the same syntactic variation is used in the comparison, and *2)* the original text fragment with the original syntactic variation is used as a template for the newly constructed fragment.

The proposed solution for sugar preservation discussed so far depends on two assumptions: *1)* (Desugared) origin terms are uniquely associated to a corresponding textfragment *2)* The nesting and ordering relation of (desugared) origin terms reflects the nesting and ordering relation of their associated text fragments. These assumptions are met for base origin terms but not necessarily for desugared origin terms. Below we discuss the adaptations we made to implement support for desugarings that violate one of the assumptions mentioned above.

Missing origin terms A common category of desugarings is given by local-to-local transformations that map a specialized construct into a more generic core construct. These type of desugarings introduce new terms, typically empty lists or placeholder constructors, that miss a corresponding origin representation. A problem occurs when a newly introduced term is changed during the refactoring transformation; because of the missing origin information the term change cannot be translated into a textual change. To address

this problem, we propagate the change as a change of the parent term, which is then reconstructed by pretty-printing. This solution fulfills the correctness and layout preservation conditions; the parent term cannot be reconstructed from the original text since it cannot be "resugared" into the original syntactic variation.

We illustrate the situation by an example based on the desugaring discussed in Section 6.2.1. Suppose that we have a `PropertyNoAnno(name, type)` term that is desugared into a `Property(name, type, [])` term, and then transformed into the term `Property(name, type, [NotNullAnno()])`. Text patching of the term change `[]` to `[NotNullAnno()]` fails because the `[]` term cannot be located in the source text. Therefore, we propagate this change as a change of the parent terms. Then, the old, desugared parent term `Property(name, type, [])` is located in the source text via its origin term `PropertyNoAnno(name, type)`; and the corresponding text fragment is replaced by the pretty-print result of the new parent term `Property(name, type, [NotNullAnno()])`.

Duplicated origin terms A second category of local-to-local desugarings that require special attention are desugarings that produce multiple terms from the same origin term. A problem occurs in case the subsequent transformation affects the duplicated terms in different ways, leading to conflicting term changes. The problem is solved by propagating the conflicting changes as a change on the common ancestor term. The ancestor term is then reconstructed by pretty-printing. Again, pretty-printing of the ancestor term is required because it cannot be resugared into its original syntactic variation.

We found an example of such a desugaring in the Stratego compiler; the desugaring transforms a lambda rule (`\w -> ...\`) into a scoped rule (`{w:(w -> ...)}`), thereby duplicating the variables at the left hand side. Suppose that the subsequent transformation transforms the desugared origin term into the new term `{v:(w -> ...)}`. Comparison of the new term with its desugared origin term results in the conflicting term changes: `w => v` and `w => w`, the latter representing the identity term change. We solve the conflict by propagating a textual change for the common ancestor term. That is, the pretty-printed construct `{v:(w -> ...)}` replaces the original text fragment `\w -> ...\` which is accessed via its desugared and base origin term.

Changed ordering and nesting relations Local-to-global and global-to-local desugarings affect the ordering and nesting relations between terms in the abstract syntax tree. The text reconstruction algorithm offers only limited support for these kind of desugaring transformations. Problems may occur when dislocated terms take part in a term change caused by the refactoring transformation. For example, the replacement of a term `c1` with a term `c2` in an ancestor term `p` can only be translated into a textual replacement in case the text fragment associated to `p` is an ancestor of the text fragment associated to `c1`. Furthermore, the intended location of a newly inserted term becomes unclear in case the term is inserted between two list elements that originate from two different locations in the original program. Because of these prob-

lems, complex non-local desugarings should be avoided, or, when possible, undone in a resugaring step applied just before text reconstruction. To diagnose and report ordering and nesting problems to the language developer, an ordering and nesting check algorithm can be implemented.

6.9 EVALUATION

We implemented the text reconstruction algorithm in Spoofax (Kats and Visser, 2010), the sources of the library are available on-line (Spoofax, 2011). We successfully applied the algorithm to Renaming, Extract and Inline refactorings defined in the Mobl (Hemel and Visser, 2011) and Stratego (Bravenboer et al., 2008) editor. In both cases, the refactorings were specified on the abstract syntax tree that results after desugaring. In addition, we applied the algorithm to the Java restructurings mentioned in this section. For future work we will implement more refactorings and we will experiment with different languages and layout conventions.

It is impossible for automatic tools to handle all layout correctly. After all, textual comments are written for human beings. Ultimately, comments can only be related to the code by understanding natural language. Therefore, instead of trying to prove that our tool handles layout correctly, we show that our approach meets practical standards for refactoring tools. We compare the layout handling of our technique with the refactoring support in Eclipse Java Development Tools (JDT), which is widely used in practice.

Van De Vanter (de Vanter, 2001) points out the importance of the documentary structure for the comprehensibility and maintainability of source code. The paper gives a detailed analysis of the documentary structure consisting of indentation, line breaks, extra spaces and comments. The paper sketches the prerequisites for a better layout handling by transformation tools. We use the examples and requirements pointed out by Van De Vanter to provide a qualitative evaluation of our approach. We constructed a test set consisting of Java fragments with different layout styles. This set includes test cases for indentation and separating whitespace, as well as test cases for different comment styles, covering all comment styles discussed by Van De Vanter (de Vanter, 2001) and illustrated in Figure 6.17.

The results are summarized in Table 6.1; + means that the layout is accurately handled, −/+ indicates some minor issues, while − is used in case more serious defects were found. A minor issue is reported when the layout is acceptable but does not precisely follows the style used in the rest of the code, a serious defect is reported in case the layout is untidy or when comments are lost.

The results show that our approach handles layout adequately in most cases. Different comment styles are supported (1-15), and the adjustment of whitespace gives acceptable results (16-19). 17 and 19 show that variations in code style only led to some minor issues. For example in 17, the indent of the new inserted method correctly follows the indentation of the adjacent methods, but the indentation in the body follows the style defined in the

	Cat.	Description	E	CT
1	P1	Inline on method preceded by block comment	+	+
2		Inline on method preceded by a commented-out method	-	+
3		Move method preceded by multiple comments	+	+
4		Convert-to-field on the first statement of a group preceded by a comment	-	+
5		Convert-to-field on statement below commented-out line	-	+
6	P2	Change method signature	+	+
7	S1	Extract method, last stm ends with line comments	+	+
8		Extract method, preceding stm ends with line comments	+	+
9		Convert-to-field, decl with succeeding line comments	-	+
10	S2	Change method signature	+	+
11	S3	Change method signature	-/+	-/+
12	Inside	Extract method with comments in body	+	+
13		Inline method with comments in body	+	+
14	Selection	Extract method, preceding comments in selection	+	+
15		Extract method, preceding comments outside selection	+	+
16	Indent	Extract method, code style follows standards	+	+
17		Extract method, code style deviates from standards	-	-/+
18	Sep. ws	Extract method, code style follows standards	+	+
19		Extract method, code style deviates from standards	-/+	+
20		Extract method, mixed code styles	+	-/+
21	Format	Extract method, standard code style	+	+
22		Extract method, code style deviates from standard	-/+	-/+
23	V. align	Renaming so that v. alignment of "=" is spoiled	-	-
24		Renaming so that v. alignment of comments is spoiled	-	-

E : Eclipse Helios (3.6.2)

CT: Text Construction

Table 6.1 Layout preservation results for the Eclipse based Java IDE and the text reconstruction algorithm.

pretty-print definition. Mixed code styles (20) can lead to inferior results, the reason is that the style of a surrounding construct with a deviating code style may be adopted. Vertical alignment (23, 24) is not restored. A possible improvement is to restore vertical alignment in a separate phase, using a post processor.

Eclipse does not implement the same refined heuristic patterns as our technique, which explains the deviating results in 2, 4, and 5. In those three

cases, the comments were incorrectly associated with the moved code structures and, consequently, did not remain at their original location. In all three cases the comment did not show up in the modified source code. In 9, the comment was not migrated to the new inserted field, although it was (correctly) associated to the selected variable declaration. The reason is that the relation between the inserted field and the deleted local variable is not set. In our implementation, the origin tracking mechanism keeps track of this relation. Eclipse uses editor settings to adjust the whitespace surrounding new inserted fragments, which works well under the condition that the file being edited adopts these settings.

We implemented a general solution for layout preservation with the objective to support the implementation of refactorings for new (domain-specific) languages. Using our approach, the layout preservation is not a concern for the refactoring programmer but it is automatically provided by the reconstruction algorithm. The evaluation indicates that our generic approach is on par with practical standards.

6.10 DISCUSSION

In this section we will discuss some limitations that we found while experimenting with the implementation of our approach in the Spoofax language workbench. Some of these limitations are implementation issues, while other limitations are more intrinsic to the approach. We will also sketch possible solutions to the identified issues.

A problem seen in practice is that origin information is not always properly propagated during rewriting. This is for most part a limitation of the current implementation, however, in some cases origins of terms can only be determined heuristically. As a consequence, fragments may be reconstructed by pretty printing instead of from their origin fragment which leads to inferior results. To handle this problem, we improved the robustness of the reconstruction algorithm. First, as discussed in Section 6.5.3, we only apply text patching for subterms that are actually changed, which limits the number of fragments that need to be reconstructed. Secondly, we implemented a heuristic to detect the origin of a term based on the origin relations of its sub terms. Finally, we allow the refactoring implementor to set (or remove) the origin of a term programmatically. We leave it as future implementation work to improve the origin tracking technique itself. A possible solution to the problem of loosing origin information can be found in Kort and Lämmel (2003), which suggests a lightweight notation to make origin relations explicit in the rewrite rules.

A second issue related to origin tracking is the selection of the origin term in case of multiple, equivalent alternatives. It is not always possible to attribute a unique origin to a term. For example, consider the following rewrite rule `Plus(x, x) -> Mult(2, x)`. The origin of `x` in the resulting term `Mult(2, x)` could be either one of the `x` terms of the original term `Plus(x, x)`. The actually selected origin term in this case depends on the

compiler of the rule. For example, in the Stratego compiler the left `x` would win since the compiler first binds `x` and then matches against `x`. Another more sophisticated example is seen with an Extract method refactoring that also replaces code clones of the selected statements with a method call. In this case, the user selection in the editor determines the origin of the statements in the extracted method body, ignoring the alternatives offered by the code clones. In both examples, the formatting of the actually selected origin term is preserved in the resulting program fragment, while the formatting of the equivalent alternatives is ignored. In practice, the effect of arbitrary origin selection seems marginal, since typically all possible alternatives give acceptable results.

Programmers are not always consequent in the formatting of code constructs, which may depend on their specific context. For example, line wrapping could be applied to a particular code construct to prevent long lines of code. A construct with a slightly deviating style might be refactored in such a way that the reason for applying this style no longer holds after the refactoring. Since code fragments are always constructed from their original fragment (if any exists), this situation leads to possible inferior results. We consider this a trade-off of our approach: we favor preservation of the existing formatting above applying a default formatting convention.

Our current approach applies a predefined pretty-print function to format newly inserted code constructs. Since programmers may prefer different formatting conventions, the formatting defined by the pretty-print function may differ from the formatting style preferred by the programmer, and it may be inconsistent with the conventions used in the rest of the program. To overcome this limitation, users should be able to configure the formatting that is applied by the pretty-print function. This functionality is not yet supported in our current implementation.

The correctness of the text reconstruction algorithm depends on the homomorphism assumption discussed in Section 6.6.3. This assumption must be met by the grammar defined for the language. When the algorithm is applied to desugared abstract syntax trees, additional assumptions must be met by the specified desugaring transformation; that is, only local-to-local desugarings are fully supported (Section 6.8). When these assumptions are violated, text reconstruction may fail or may lead to syntactically invalid code. Refactoring implementors that are aware of these limitations can customize the reconstruction algorithm by overriding the reconstruction of particular terms with their own reconstruction strategy.

6.11 RELATED WORK

We implemented an algorithm for layout and syntactic sugar preservation in refactoring transformations. Instead of trying to construct the entire source code from the transformed AST, the algorithm applies text patches to the original source text. The text patches are calculated from changes in the abstract structure, using origin tracking to relate terms in the transformed AST with

their originating term and code fragment. We extended the origin tracking mechanism to also track desugared origin terms, which made it possible to preserve syntactic sugar.

A main challenge is the treatment of spacing and comments at the edges between the changed and the unchanged code constructs. We specified layout adjustment functions that correct the whitespace of reconstructed fragments, so that the spacing of the surrounding code is adopted. Comments are migrated according to their intent. We define heuristic patterns for comment binding that interpret the documentary structure near the constructs. The comment patterns are flexible in the sense that they do not assume a one-to-one relation between comments and abstract terms. The heuristic rules are language generic and cover layout styles commonly seen in practice.

AST based approaches Various attempts have been made to address the concern of layout preservation by adding layout information to the AST. The layout information is used after a refactoring transformation to reconstruct the modified source code from the transformed AST. In (Li et al., 2006) comments are stored as AST annotations, while the RefactorErl tool (Kitlei et al., 2009) stores layout information in an external data structure. (Lohmann and Riedewald, 2003) describes an approach that introduces additional layout branches to the abstract tree; the authors propose an automated migration of the transformation rules to take care of these layout branches.

All approaches based on extended ASTs succeed, to a certain extent, in preserving the original layout. In most approaches, layout is preserved for the unaffected parts, but the reconstruction of the affected parts has limitations. The implicit assumption is that the documentary structure can be mapped satisfactorily onto abstract syntax trees. However, the mapping of layout elements to abstract terms has intrinsic limitations. Attaching comments to preceding (or succeeding) terms is a simplification that fails in cases when a comment is not associated with a single AST term, as is shown in examples provided by de Vanter (de Vanter, 2001). Another shortcoming is related to indentation and whitespace separation at the beginning and end of changed parts. Migrating whitespace is not sufficient since the indentation at the new position may differ from the indentation at the old position, due to a different nesting level. Furthermore, newly constructed structures should be inserted with indentation and separating whitespace. A final limitation is the lack of support for the preservation of syntactic sugar, which gets lost in case a refactoring transformation is preceded by a desugaring transformation.

ASF + SDF As an alternative to using some abstract representation, rewriting techniques can also be implemented on concrete syntax trees. The rewriting technique that is used in the original ASF+SDF Meta-Environment (Klint, 1993) uses rewrite rules in concrete syntax that operate on structured terms in concrete syntax. Layout is efficiently ignored by discarding all layout nodes from the syntax trees of both the rewrite rules and the term.

To address the concern of layout preservation, van den Brand and Vinju (2000) propose to use full parse trees instead of parse trees that are stripped

from layout. The rewrite engine is adapted to deal with the extra layout branches. That is, term equality and matching are implemented modulo layout nodes (i.e. any two layout nodes always match independent of their content), while newly constructed terms use the layout of the right hand side of the rewrite rules. This method succeeds in preserving the layout of sub terms that are not rewritten, while for sub terms that are rewritten the original layout is lost permanently. The method does not provide the ability to utilize layout nodes explicitly.

To address these limitations, Vinju (2005, Chapter 8) presents a method to propagate layout selectively with help of layout variables. The layout variables allow the programmer to consume and analyze the original layout, and to propagate it or produce new layout in the constructed terms. The static semantic of ASF+SDF guarantees that the transformations are syntactically save, i.e., they can not introduce syntax errors. In principle, layout preserving transformations can be implemented using this technique to rewrite the layout of terms. However, a strong limitation for our use case is that the refactoring implementor must explicitly implement the layout handling in the rewrite rules, which involves complicated strategies for layout adjustment and comment migration.

An interesting direction could be to automate implicit propagation of layout nodes from the left-hand side to the right-hand side of rewrite rules. This requires a technique to determine the origins of constructed sub terms at the right hand side of the rule. If the constructed sub term has the same signature as its origin term, then the original formatting can be applied. Furthermore, indentation correction, comment migration and bracket insertion must be implemented to ensure maximal layout preservation and correctness. Preservation of syntactic sugar can not be achieved using this technique, since all information about the original constructs is permanently lost in the desugaring transformations.

Parse tree annotations Kort and Lämmel (2003) present a general technique to handle crosscutting concerns in rewriting. This technique relies on tree annotations to store relevant information. Separation of concerns is achieved by implementing *progression methods* that define generically how annotations are propagated from the input to the output of a rewrite step based on sharing relations between input and output sub terms. The paper gives several examples of concerns that could be implemented using this technique; among these examples are layout preservation and syntactic sugar preservation. The paper does not go into details about how these preservation concerns are implemented best.

As for the concern of layout preservation, there are different ways to group layout in term annotations, but, as explained in this chapter, projecting layout onto terms involves the risk of loosing crucial information. A possible approach is to work with concrete syntax trees and only annotate the leaf nodes, i.e., tokens, capturing the layout preceding a token as its annotation. This approach at least ensures that no layout information is lost for terms that are not (yet) rewritten. The remaining problem is to implement a progression method

that fulfills the requirements identified in Section 6.2.3. Although this seems possible in principle, a condition is that the implementation of progression methods allows access to the layout structure of surrounding terms, which is required for indentation correction and comment migration.

HaRe HaRe (Li and Thompson, 2006; Li et al., 2005) is a refactoring tool for Haskell that preserves layout. The program is internally represented by the abstract syntax tree and the token stream, which are linked by source location information. Layout preservation is performed explicitly in the transformation steps, which process the token stream and the AST in parallel. After the transformation, the source code is extracted from the modified token stream.

Haskell programs can be written in layout-sensitive style, which means that the interpretation of a syntax phrase may depend on its layout. For this reason, it is essential for the refactoring tool not to violate the layout rules when transforming the program. HaRe implements a layout adjustment algorithm to keep the layout correct. The algorithm ensures that the meaning of the code fragments is not changed, which does not necessarily mean that the code is as much as possible like the original one in appearance. HaRe uses heuristic rules to move/remove comments together with the associated program structures. These heuristics include rules for comments that precede a program structure and end-of-line comments that follow after a structure.

Similar to our approach, HaRe uses the token stream to apply layout analysis and to extract source code fragments. The main difference is that HaRe modifies the token stream during the transformation, while we reconstruct the source code afterwards, using origin-tracking to access the original source code. By using an extended version of the origin tracking technique, we also preserve syntactic sugar for refactoring transformations that are preceded by a desugaring transformation. The requirement to change the AST and token stream in parallel makes it harder to implement new transformations and requires an extension of the rewrite machinery specific for source-to-source transformations. We clearly separate layout handling from rewriting, which enables us to use the existing compiler infrastructure for refactoring transformations.

Eclipse The Java Developer Toolkit (JDT) used in Eclipse offers an infrastructure for implementing refactorings (Eclipse documentation, 2010). Refactoring transformations are specified with replace, insert and remove operations on AST terms, which are used afterwards to calculate the corresponding textual changes. Common to our approach, the replace, insert and remove operations on terms are translated to textual modifications of the source code. However, instead of being restricted to the replace, delete and insert operations on terms, we compute the primitive AST modifications by applying a tree differencing algorithm to the transformed abstract syntax tree. As a result, the transformation and text reconstruction are clearly separated. Thanks to this separation of concerns, we can specify refactorings in a specialized transformation language (Stratego).

The master thesis (Stocker, 2010) describes the refactoring framework implemented for the Scala IDE, also paying attention to the problem of text reconstruction. As with our approach, the textual changes that follow from the abstract transformation are reconstructed after the transformation. The work does not mention how they trace back terms in the transformed AST to terms in the original or desugared AST as we do with origin tracking.

Text patching The LS/2000 system (Dean et al., 2001; Malton et al., 2001) is a design-recovery and transformation system, implemented in TXL. LS/2000 is successfully applied for "year 2000" remediation of legacy COBOL, PL/I, and RPG applications. The system implements an approach based on automated text patching. The differences between the original code and the transformed code are calculated with a standard differencing algorithm, operating on the token stream. The deviating text regions are merged back into the original text.

The token based differencing successfully captured changes that were relatively small. For millennium bug renovations, typical changes were the local insertion of a few lines of code. When the changes are large, or involve code movement, standard differencing algorithms do not work well (Malton et al., 2001). We implemented a tree differencing algorithm that reconstructs moved code fragments by using origin tracking, furthermore, fragments with nested changes are reconstructed by recursion on subtrees.

Lenses Foster et al. (2007) implement a generic framework for synchronizing tree-structured data. Their approach to the view update problem is based on compoundable bi-directional transformations, called lenses. In the GET direction, the abstract view is created from the concrete view, projecting away some information; in the PUTBACK direction, the modified abstract view is mapped to a concrete representation, restoring the projected elements from the original concrete representation. The lens laws, which resemble our preservation and correctness criteria, impose some constraints on the behavior of the lens. Given a certain GET function, in general, many different PUTBACK functions can be defined. The real problem is to define a PUTBACK function that does what is required for a given situation. We define CONSTRUCTTEXT$_S$ as a PUTBACK function for parsing, and prove that it fulfills the correctness and (maximal) layout preservation criteria.

Our approach is based on origin tracking as a mechanism to relate abstract terms with their corresponding concrete representation. Origin tracking makes it possible to locate moved subtrees in the original text. Furthermore, lists are compared using the origin relation to match corresponding elements. In contrast, lenses use the concrete representation as an input parameter to the PUTBACK function. As a consequence, details are lost about how subterms relate to text fragments. This seems especially problematic in case terms have nested changes, or when they are moved to another location in the tree. We defined heuristic rules for comment binding and layout adjustment functions to correct the spacing surrounding the changed parts. Layout adjustment and comment migration might be hard to express in the lenses framework. Foster

et al. (2007) mention the expressiveness of their approach as an open question. Layout preservation seems a challenging problem in this respect.

6.12 CONCLUSION

Refactorings are source-to-source transformations that help programmers to improve the structure of their code. With the popularity and ubiquity of IDEs for mainstream general purpose languages, software developers come to expect rich editor support including refactorings also for domain-specific software languages. Since the effort that can be spent on IDEs for DSLs is often significantly smaller than the effort that is spent on IDEs for languages such as Java, this requires tool support for the high-level definition of refactorings for new (domain-specific) software languages.

An important requirement for the acceptability of refactorings for daily use is their faithful preservation of the layout of programs. Precisely this aspect, as trivial as it often seems compared to the actual refactoring transformation, has confounded meta-tool developers. The result is typically that the definitions of refactorings are contaminated with code for layout preservation. The lack of a generic solution for layout preservation has held back widespread development of refactoring tools for domain-specific languages.

In this chapter, we have presented an approach to layout preservation that separates the layout preservation concern from the structural definition of refactorings. The approach allows the refactoring developer to concentrate on the structural transformation, leaving source code reconstruction to a generic algorithm. The algorithm computes text patches based on the differences between the old and the new abstract syntax tree, relying on origin tracking to identify the origins of subtrees. The algorithm applies layout conventions for indentation and blank lines from the old code to newly created pieces of code; heuristic rules are defined for comment migration. The algorithm also supports preservation of syntactic sugar, in case the refactoring transformation is preceded by a desugaring stage that maps sugared constructs in an enriched syntax into equivalent constructs in the more restricted core syntax.

Name Binding Preservation

7

ABSTRACT

The implementation of refactorings for new languages requires considerable effort for language developers. Part of this effort is in name binding preservation, i.e., making sure that variable names bind to the same declaration before and after the transformation. We show how this effort can be reduced by using language parametric techniques. We present a language-parametric technique to detect name binding violations, and, as a refinement, a technique to restore name bindings by introducing qualified names. Both techniques are implemented within the paradigm of strategic term rewriting. The techniques offer an efficient and reliable solution by reusing the name analysis implemented in the compiler front end; the semantics of the language is implemented only once, with the compiler being the single source of truth. We evaluate our approach by applying the preservation techniques in Rename refactorings implemented for two domain-specific languages and a subset of the Java language.

7.1 INTRODUCTION

Full-featured integrated development environments (IDEs) are essential for developers to be productive in a language. A key factor in the success of these IDEs is the provision of services specifically tailored to the language The implementation of language-specific editor services requires considerable effort from the language engineer, involving syntactic and semantic analyses similar to those found in the compiler front end. For many current programming languages, IDEs have been developed separately from a compiler or interpreter of the language. A problem with separate development of IDEs and language compilers/interpreters is that there tends to be significant overlap between them. For a complex IDE that supports deeply integrated semantic editor services, either an existing compiler should be integrated into the IDE, or the language syntax and semantics should be re-implemented leading to undesirable redundancy.

Language workbenches (Fowler, 2005, 2011) address this problem by enabling developers to create new languages together with high-quality IDE support. Language workbenches make language engineering more efficient by *a)* providing an integrated environment for the development of both languages and their IDEs and *b)* providing meta-languages and frameworks for performing language engineering tasks, and providing IDE support for these meta-tools. Some prominent examples of language workbenches for textual editors include: EMFText (Heidenreich et al., 2009), TCS (Jouault et al., 2006),

Xtext (Efftinge and Voelter, 2006), MontiCore (Krahn et al., 2008) and Spoofax (Kats et al., 2010a). We are extending Spoofax with a framework for the implementation of refactorings.

Refactorings are transformations that improve the internal structure of a program while preserving its behavior. Users rely on refactoring tools to warn them against possible changes in the behavior of the program. The implementation of behavior preservation conditions is challenging, requiring deep analysis of the semantic structure of the program. Even with specialized compiler techniques such as rewriting languages and attribute grammars, implementing conditions for behavior preservation is a tough task that requires global understanding of the semantics of the language for which the refactoring is developed.

Traditionally, conditions for behavior preservation are implemented as preconditions that are checked before the transformation (Opdyke, 1992; Roberts, 1999). This approach has some clear weaknesses. It is extremely difficult to derive a correct set of preconditions that guarantees behavior preservation without excluding refactorings that in fact could be carried out. Moreover, additional preconditions have to be implemented in case the language evolves. Finally, preconditions are not easily shared among different refactorings, nor do they transfer to different languages. Current refactoring implementations rarely guarantee behavior preservation. Rather, refactorings are tried on examples and validated 'in the field', which may result in subtle errors triggered by corner cases not foreseen by the developer. Even mature refactoring frameworks used in current IDEs contain bugs as a result of insufficient preconditions (Schäfer et al., 2008; Soares, 2010).

The limitations of preconditions are addressed in (Ekman et al., 2008). The authors propose an invariant-based approach which is implemented in JastAdd (Ekman and Hedin, 2007), an attribute grammar system that extends Java with support for circular reference attribute grammars (RAGs) (Hedin, 2000). Invariants for name binding, control-flow and data-flow are implemented as complementary analysis functions that are checked after the transformation. Compared to refactoring frameworks used in existing IDEs, Java refactorings implemented in JastAdd proved to be more reliable and required less effort in terms of lines of code. The success of this approach can be explained by the fact that, compared to preconditions, the specification of the invariants more closely follows compiler analysis that define the static semantics of a language.

Attribute grammars allow for a high-level declarative specification of semantic analysis, however, they offer no specific language features to declaratively express AST transformations. An alternative approach is to use term rewriting for implementing refactorings. Term rewriting is used in systems such as Maude (Garrido and Meseguer, 2006), Tom (Balland et al., 2007), Strafunski (Lämmel, 2002) and Stratego (Bravenboer et al., 2008). Term rewriting makes it easy to describe syntax tree transformations, but is less declarative with respect to semantic analysis. Rewriting systems typically view syntax trees as terms without a concept of term identity; AST terms are character-

ized only by their subtrees, and not by their position within the whole tree. Reference attributes crucially rely on term identity, and hence have no direct equivalent in a term-based representation of syntax trees.

In this chapter we implement an invariant-based approach to behavior preservation within the paradigm of term rewriting. The chapter focuses on preservation of name bindings. All refactorings that introduce new names into a scope have to guard against accidental changes of existing name bindings, which change the semantic behavior of the program. We implement a preservation criterion for statically known name bindings that is generic applicable to different languages and different refactorings. The preservation criterion takes as arguments a language-specific name analysis and a refactoring specific transformation.

A limitation of the preservation criterion is that it rejects refactorings that could in fact be carried out after applying some small modifications. Many languages offer the possibility to access variables defined in a namespace via qualified names. A notable example is Java; a field that is shadowed by a local variable can still be accessed using a qualifier such as `this` or `super`. As a refinement to the preservation criterion, we show that name analysis can be implemented as a reusable traversal strategy that can be applied to restore name bindings by creating qualified names, thereby enabling refactorings that would otherwise be rejected.

The described techniques for name binding preservation and name binding restoring form the main contribution of the chapter. Both techniques use strategic term rewriting and are implemented in the Stratego rewriting language (Bravenboer et al., 2008). By reusing the name analysis implemented in the compiler front end, the techniques adhere to the "Single Source of Truth" also called "Don't Repeat Yourself" principle (Hunt and Thomas, 2000), with the compiler being the single authoritative representation of semantic behavior.

To see how our approach works in practice, we used the techniques to implement rename refactorings for Stratego (Bravenboer et al., 2008), Mobl (Hemel and Visser, 2011), and a subset of the Java language. Our experience shows that the effort it takes to implement name binding preservation is significantly reduced by using language-parametric techniques. Furthermore, manual and automated testing demonstrated that the implemented rename refactorings indeed preserve the original binding structure.

7.2 THE STRATEGO TRANSFORMATION LANGUAGE

This section gives a short introduction into the Stratego transformation language, which is used to implement the techniques described in this chapter. A more complete overview of the Stratego language is given in (Visser, 2004).

Stratego (Visser et al., 1998; Visser, 2004) is a language for the specification of program transformations and analyses, based on the paradigm of term rewriting with programmable traversal strategies. Stratego uses conditional

rewrite rules to define basic transformations on terms. These rules adhere to the following schema:

```
r : p1 → p2 where c
```

The rule `r` applies to a term when its left-hand side `p1` matches the term, and the (optional) condition `c` succeeds. The result is the instantiation of `p2` with the variable bindings found during pattern matching in `p1` and `c`. The rule is said to *fail* when either the subject term does not match the left-hand side or when the condition fails.

Rules are basic strategies that perform the transformation specified by the rule or fail. Strategies can be parameterized with strategy and term arguments, e.g., `r(s1 ... sm|t1 ... tn)`. Furthermore, strategies can be overloaded. That is, when invoking a rule with a given signature, all rules with that signature are tried in some unspecified order until one succeeds.

Strategies can be combined into more complex strategies by means of strategy operators. Sequential operators combine strategies that apply to the root of a term, examples are: identity (`id`), failure (`fail`), sequential composition (`s1 ; s2`), choice (`s1 + s2`), guarded choice (`s1 < s2 + s3`), negation (`not(s)`), and recursive closure (`rec x(s)`). Term traversal operators, e.g., `all(s)`, `one(s)`, and `some(s)`, express strategy application to the direct subterms of a term.

Combining these operators allows the generic definition of a wide range of term traversals. For example, `bottomup(s) = all(bottomup(s)); s` generically defines a post-order traversal. The Stratego standard library provides a collection of such strategies for general use.

7.3 MOTIVATION

Refactorings are behavior preserving source-to-source transformations with the objective of improving the design of existing code (Fowler, 2002). Although it is possible to refactor manually, tool support reduces evolution costs by automating error-prone and tedious tasks. In particular, behavior preservation conditions are automatically checked; in case the transformation changes the semantic behavior of the program, the refactoring is rejected and the problem is reported to the user.

Name bindings associate identifiers with program entities such as variables, methods and types. Name bindings form a semantic concern that should be preserved by refactorings. Intuitively, all name accesses in a program should bind to the same declarations before and after the transformation. That is, declarations should not accidentally become shadowed by declarations introduced by the refactoring transformation. The problem of maintaining existing name bindings occurs in many refactorings, with renaming being the obvious example. Automatically renaming an identifier requires binding information to determine which names must be renamed. Furthermore, behavior preservation requires that conflicting and accidentally shadowed declarations are detected.

```
class Person {
  int age;
  void setAge(int s)
  {
    age = s;
  }
}
```

```
class Person {
  int age;
  void setAge(int age)
  {
    age = age;
  }
}
```

```
class Person {
  int age;
  void setAge(int age)
  {
    this.age = age;
  }
}
```

Figure 7.1 Renaming `s` to `age`, incorrectly and correctly applied to code fragment.

Shadowing occurs when the same identifier is used for different entities in nested lexical scopes. Shadowing poses a challenge for binding preservation in refactorings, Figure 7.1 illustrates variable shadowing. The `age` variable in the body of the `setAge` method refers to the `age` field of the surrounding class. After renaming `s` to `age` (Figure 7.1, mid), the age field is shadowed by the method parameter `age`, declared in an inner scope. As a consequence, the binding of the `age` variable has changed. As a minimal requirement, refactoring tools are expected to detect name binding violations to guarantee the save application of the refactoring.

A namespace is a scope that groups related identifiers, and allows the disambiguation of homonym identifiers residing in different namespaces. In many programming languages, identifiers that appear in namespaces have a short local name and a long qualified name. By using the qualified name, an identifier can still be accessed in an inner scope, even though a program entity with the same name is declared within that inner scope. Figure 7.1 (right) provides an example. By adding a name qualifier (`this`) to the `age` variable, the original name bindings are restored so that the refactoring can still be carried out. Thus, the policy to reject all refactorings with name binding violations is in fact too restrictive, preventing refactorings that only require a small modification to be applied correctly. Therefore, a wished feature for refactoring tools is that they are able to restore violated name bindings by creating qualified names.

Name binding analysis is implemented by the compiler of a language, involving complex semantic rules for namespaces, scoping and visibility. Name binding preservation conditions in refactoring frameworks must implement the same semantic rules. This can hardly be guaranteed with an ad hoc, precondition based approach. Indeed, existing refactoring implementations based on preconditions contain bugs because of (new) language features that are not (yet) supported (Schäfer et al., 2008). A more reliable solution can be realized when the criterion for name binding preservation is directly based on the name analysis implemented in the compiler. This way, the semantics assumed by the refactoring tool is guaranteed to be consistent with the semantics implemented in the compiler, even when the language evolves.

We implement a name binding preservation criterion that reuses the existing name analysis defined in the compiler front end. As a refinement, we present an approach for restoring name bindings by introducing qualified names. The name binding preservation criterion is discussed in Section 7.4,

```
ClassDec(
  "Person"{"n0"}
, [ FieldDec("int", "age"{"n1"})
  , MethodDec(
      Void(), "setAge"{"n2"}, [Param("int", "age"{"n3"})]
    , [Assign(QA(This(), "age"{"n1"}), Var("age"{"n3"}))]
)])
```

Figure 7.2 Name annotations make bindings explicit. The reference names 'n1' and 'n3' distinguish the parameter 'age' from the field 'age'.

```
int age{n1};
void setAge(
    int s{n3}){
  age{n1} = s{n3};
}
```

```
int age{o1};
void setAge(
    int age{o3}){
  age{o3} = age{o3};
}
```

```
int age{p1};
void setAge(
    int age{p3}){
  this.age{p1} = age{p3};
}
```

Figure 7.3 Name binding pattern `[n1, n3, n1, n3]` in the left fragment, has binding violations in the mid fragment `[o1, o3, o3, o3]`, and is preserved in the right fragment `[p1, p3, p1, p3]`.

while Section 7.5 discusses restoration of name bindings. Section 7.6 reports on our experience with applying these techniques to different languages.

7.4 PRESERVING NAME BINDINGS

Refactorings must preserve the name binding structure of a program, implicitly defined by the name analysis implemented in the compiler. To make the binding structure explicit, we use the technique of explicit renaming implemented in Stratego by means of term annotations. The result of the name analysis is an abstract syntax tree in which all identifiers are annotated with a globally unique reference name. That is, two identifiers are annotated with the same reference name if and only if they bind to the same declaration. Figure 7.2 shows the analyzed abstract syntax tree of the Java fragment of Figure 7.1 (right) in the ATerm format (van den Brand et al., 2000). The names in the AST are annotated with unique reference names. The reference names 'n1' and 'n3' distinguish the parameter 'age' from the field 'age'. This information can be used for example in a rename refactoring to determine which identifiers must be renamed.

We use name annotations to implement name binding preservation as a post condition on the transformed tree. The preservation condition compares the original annotations, which are preserved during the transformation and represent the original bindings, with new annotations, obtained by reanalyzing the transformed tree. Preservation is realized if and only if the original name annotations are equal to the new name annotations, modulo renaming. The criterion does not make any assumptions about the reference names that are generated for the declarations, except that they are unique. In particular, existing reference names may change after reanalysis.

Figure 7.3 illustrates the preservation condition for our running example.

```
class A { void foo{n1}(){ } }
class B extends A { }
class C extends B { void foo{n2}(){ }}
class D extends B { void bar(){ foo{n1}(); }}
```

Figure 7.4 Pull-Up: before transformation.

```
class A { void foo{n1}(){ } }
class B extends A { void foo{n2}(){ } }
class C extends B { }
class D extends B { void bar(){ foo{n1}(); }}
```

Figure 7.5 Pull-Up: after transformation.

```
class A { void foo{n8}(){ } }
class B extends A { void foo{n9}(){ } }
class C extends B { }
class D extends B { void bar(){ foo{n9}(); }}
```

Figure 7.6 Pull-Up: after reanalysis.

For convenience, the reference annotations are placed on the concrete syntax instead of the abstract syntax, furthermore, only the relevant name annotations are shown. The left and right code fragments are equivalent modulo renaming, that is, the name annotations follow the same pattern (`[a,b,a,b]`). In contrast, the name annotations in the mid fragment follow a different pattern (`[a,b,b,b]`), which means that the fragment has a different binding structure.

The preservation criterion also applies to refactorings that change the AST structure and/or introduce new elements, as long as the name annotations in the transformed AST represent the intended binding structure. This condition can easily be fulfilled. The name annotations of elements in the original AST are preserved during the transformation and express the intended binding of these elements in the transformed AST; new elements can be inserted together with a name annotation that denotes their intended binding. For example, to express the intended binding structure for the Extract method refactoring, a new name annotation must be placed on the extracted method definition as well as on its inserted call site.

Figure 7.4 illustrates the preservation criterion applied to a transformation that changes the structure of the AST. The figure shows an incorrect Pull-Up method refactoring which moves a method defined in a subclass to its superclass. In the given example, the Pull-Up method transformation violates existing name bindings; the `foo` method inserted in the `B` class shadows the `foo` method defined in its superclass `A`, which affects the method binding in

```
apply-refactoring(analyze, transform):
  ast → (ast-reanalyzed, violation-errors)
  where
    ast-transformed  := <analyze; transform> ast;
    ast-reanalyzed   := <analyze> ast-transformed;
    violation-errors := <namebinding-violation-errors>
                          (ast-transformed, ast-reanalyzed)

namebinding-violation-errors:
  (ast-transformed, ast-reanalyzed) → violation-errors
  where
    old-nb     := <collect-all(is-name)> ast-transformed;
    new-nb     := <collect-all(is-name)> ast-reanalyzed;
    violations := <zip; (binding-violations <+ ?[])> (old-nb, new-nb)
    violation-errors := <map(to-namebinding-error)> violations

binding-violations:
  [(x,y)|tl] → <conc> (hd-violations, <binding-violations <+ ?[]> tl')
  where
    hd-violations := <filter(is-binding-violation(|x,y))> tl;
    tl'           := <filter(not(?(_,y) <+ ?(x,_)))> tl

is-binding-violation(|x,y) =
  (?(_,y) <+ ?(x,_)); not(?(x,y))

is-name = is-string; has-anno

to-namebinding-error:
  (t, _) → (t, $[Name collision with name '[t]'])
```

Figure 7.7 Collecting binding violation errors by comparing name annotations[a].

[a] In the Stratego language, the syntax *<r>* is used for rule application; *s*1;*s*2 represents the sequential operator, first apply *s*1, then apply *s*2 to the resulting term; *s*1 <+ *s*2 is Stratego syntax for guarded left choice, first apply *s*1 and, only if it fails, apply *s*2 to the original term.

`D`, a subclass of `B`. The binding violation is detected by comparing the annotations that represent the original bindings collected from the transformed tree (Figure 7.5), with the annotations that result after reanalysis of this tree (Figure 7.6). A binding violation is detected because the original names cannot be mapped uniquely to the new names, that is: `[n1, n2, n1]` and `[p1, p2, p2]` give a conflict for `n1` that maps to `p1` at the first list position and to `p2` at the third position in the list.

Figure 7.7 shows the Stratego implementation of the name binding preservation technique. The rewrite rule `apply-refactoring` implements the name binding violation detection, taking a language-specific name analysis strategy and a refactoring specific transformation rule as arguments. The refactoring application returns the tree that results after applying the transformation, plus a (possible empty) list of name binding violation errors. The name binding violations are collected by comparing the name annotations that express the original bindings (`old-nb`) with the name annotations that express the actual, reanalyzed bindings (`new-nb`), in the order in which they occur in the transformed tree. All name tuples that break the implicit mapping between old and new names are returned as binding violations by the `binding-violations` rule.

Names are collected by applying the `is-name` rule, which succeeds only

```
class A { int a; }
class B extends A { int b; }
class C extends B {
  int c;
  class D extends E {
    int d; int x;
    void foo() {
      int i;
      i = <name>; }}}
class E { int e; int x;}
```

Figure 7.8 Name lookup in Java proceeds in an outward-upward motion.

for terms that are annotated strings. To improve performance, the `is-name` pattern can be implemented refactoring specific to enforce that only the endangered names are checked. In that case, the `is-name` rule must be given as an additional parameter to the `apply-refactoring` rule and passed to the `namebinding-violation-errors` rule. For convenience, we did not include this optimization possibility in Figure 7.7.

7.5 RESTORING NAME BINDINGS

In the previous section we implemented a generic name binding preservation criterion that is parameterized with an existing name analysis strategy. In this section we propose an alternate approach. Instead of passing the name analysis as a parameter to the preservation criterion, we pass the preservation criterion as a parameter to the name analysis. The advantage of this approach is that we can extend the preservation criterion with a repair rule that creates qualified names for accesses to variable declarations that have become shadowed.

The remainder of this section is organized as follows. Section 7.5.1 illustrates name lookup for a subset of the Java language. Section 7.5.2 discusses how name analysis can be implemented as a reusable traversal strategy; which can be applied to *a)* resolve name bindings for variable accesses (Section 7.5.3), *b)* detect name binding violations (Section 7.5.4), and *c)* restore name bindings by creating qualified names (Section 7.5.5).

7.5.1 *Name Lookup in Java*

As a leading example, we implement our approach for a subset of Java that models some essential features of the Java language with respect to resolving name bindings. Supported features are: class inheritance and class nesting, local variable, field and method declarations, and variable access by simple or qualified names. Most other Java features are excluded.

Figure 7.8 shows a small Java fragment which features class inheritance, class nesting and variable declarations. To illustrate name lookup in Java, we show how name bindings are resolved for variable accesses at the `<name>` location. The binding is looked up in the set of visible declarations, starting at the most local scope and proceeding in an "outwards-upwards" motion. In

```
//Class body declarations
Reference: (A{c0}, Vars(), a) → a{v0}
Reference: (B{c1}, Vars(), b) → b{v1}
Reference: (C{c2}, Vars(), c) → c{v2}
Reference: (D{c3}, Vars(), d) → d{v3}
Reference: (D{c3}, Vars(), x) → x{v4}
Reference: (D{c3}, Methods(), foo) → foo{m0}
Reference: (E{c4}, Vars(), e) → e{v5}
Reference: (E{c4}, Vars(), x) → x{v6}
Reference: (F{c5}, Vars(), f) → f{v7}

//Local scope declarations
Reference: (Local(),Vars(),i) → i{v8}

//Class inheritance relations
Super: B{c1} → A{c0}
Super: C{c2} → B{c1}
Super: D{c3} → E{c4}

//Class nesting relations
Outer: D{c3} → C{c2}

//Enclosing Class
CurrentClass: _ → D{c3}
```

Figure 7.9 Dynamic rules that store context sensitive information for the `<name>` location of the fragment in Figure 7.8.

the given example, the declared variables are looked up in the following order: local declarations (`i`), fields declared in the enclosing class (`d` and `x`), fields declared in the super class (`e` and `x`), the outer class (`c`), and the super classes of the outer class (`b` and `a`). The field `x` in the super class `E` is not accessible by its simple name `x`, since it is shadowed by the field `x` in the enclosing class `D`. However, the shadowed field can still be accessed by its qualified name, `super.x`. Qualifiers force the lookup to start at a specific namespace, thereby skipping over the names declared in nested or inner scopes.

7.5.2 *Name Analysis*

Name lookup requires context sensitive information such as the set of visible declarations, nesting and inheritance relations between namespaces, and type information. Based on the paradigm of strategic rewriting, name analysis is implemented in Stratego as a custom traversal strategy that takes care of propagating the required contextual information to the access terms. That is, the name analysis creates unique reference names at declaration sites which are stored as contextual information. This information is later used to look up the reference name at access sites. Since we treat name analysis as a black box, we do not provide implementation details. Example implementations of name analysis in Stratego are given in (Hemel et al., 2008) and (Kats and Visser, 2010).

The propagated contextual information is stored as (scoped) dynamic rewrite rules (Bravenboer et al., 2006b), the Stratego equivalent of symbol tables. Unlike standard rewrite rules, dynamic rules are created at run-time, and propagate information available at their creation contexts. As an example,

```
annotate-names:
  ast → <analyze-names(annotate-name)> ast

annotate-name:
  Var(name) → Var(name{ref-name})
  where
    ref-name := <lookup> (Local(), name{})

annotate-name:
  QA(qualifier, name) → QA(qualifier, name{ref-name})
  where
    ref-name := <lookup> (<get-type> qualifier, name{})

get-type : This() → <CurrentClass>          // this
get-type : Super() → <CurrentClass; Super> // super
get-type : QThis(ctype) → ctype             // C.this
get-type : QSuper(ctype) → <Super> ctype    // C.super
get-type : CastRef(ctype, t) → ctype        // ((A)t)

lookup:
  (ns, name) → ref-name
  where
    <while-not(
      while-not(
        // looks up reference for name
        // in namespace (Local() or classname)
        ref-name := <Reference> (<id>, name),
        ns-up
      ),
      ns-out
    )> ns

// lookup rules for the name of the
// super class, current class and outer class
ns-up : cl → <Super> cl
ns-out: Local() → <CurrentClass>
ns-out: cl → <Outer> cl
```

Figure 7.10 `annotate-names` sets reference annotations at variable accesses. The reference is looked up in an outward-upward motion.

Figure 7.9 shows dynamic rules that contain the required contextual information for the `<name>` location in the fragment of Figure 7.8.

7.5.3 *Resolving Name References*

As a complement of the name analysis traversal, a name lookup rule is implemented in the compiler front end. The name lookup rule is applied at the access terms to calculate the reference name from the propagated contextual information. Adhering to the Stratego paradigm of reusable traversal strategies, we assume that the lookup rule is passed as a parameter to the name analysis traversal.

Figure 7.10 shows the Stratego code for name lookup in our Java sublanguage. The given `annotate-name` rule calls the `lookup` rule to set a looked up reference name as an annotation to the access term. Lookup starts respectively at the local namespace for simple names (`Var(name)`), or at the namespace associated to the qualifier for qualified names (`QA(qualifier, name)`). The lookup logic is implemented in the `lookup` rule; which progresses outwards

```
collect-binding-violations:
  ast → violations
  where
    <analyse-names(store-binding-violation)> ast;
    violations := <bagof-BindingViolation>

store-binding-violation:
  access → access
  where
    <is-binding-violation> access;
    rules(BindingViolation:+ access)

is-binding-violation:
  access → access
  where
    new-annotated-access := <annotate-name> access;
    <not(equal)> (access, new-annotated-access)
```

Figure 7.11 Binding violations are collected by comparing the old- and newly analyzed binding annotations during a name analysis traversal.

lexically (`ns-out`), taking a detour upwards the inheritance hierarchy (`ns-up`). When the processed namespace contains a declaration with the given name, the `Reference` rule applies successfully and the reference name is returned. Otherwise, name lookup proceeds with the next, more global namespace set by `ns-up` or `ns-out`.

7.5.4 *Checking Name Bindings*

Name analysis makes the binding structure of an abstract syntax tree explicit in the form of name annotations at declaration and access sites. The annotations are preserved during transformation and represent the original binding structure in the transformed tree. Name binding preservation requires that the original binding structure corresponds to the actual binding structure of this tree. In Section 7.4 we showed how name binding violations can be detected *after* reanalysis of the transformed tree by constructing a mapping between original and actual binding annotations. As an alternative, this section shows how name binding violations can be detected *during* reanalysis of the transformed tree.

Figure 7.11, `collect-binding-violations`, implements a strategy that collects binding violations by comparing the name bindings that hold before and after the transformation. The bindings that held before the transformation are stored in the tree as annotations, the bindings that hold after the transformation are looked up during the re-applied name analysis. To allow the comparison of old and new bindings, we assume that declarations keep their original reference name when revisited. The comparison of old and new bindings is implemented by the `is-binding-violation` rule (Figure 7.11), which succeeds if original and newly looked up reference annotations are different. In case a binding violation is detected, a binding violation error is stored (`BindingViolation`) which will later be reported to the user.

```
restore-name-bindings:
  ast → <analyze-names(restore-binding)> ast

restore-binding:
  access → qualified-access
  where
    qualified-access := <
      create-qualified-access <+
      rules(BindingViolation:+ access)
    > access

create-qualified-access:
  access → qualified-access
  where
    qualified-access :=
      <while-not(
        while-not(
          where(not(is-binding-violation)),
          cast-up
        ),
        cast-out
      )> access

//cast-up and cast-out qualified names
cast-up:  QA(qualifier, name) →
          QA(<cast-up-qualifier> qualifier, name)
cast-out: QA(qualifier, name) →
          QA(<cast-out-qualifier> qualifier, name)
cast-out: Var(name) → QA(This(), name)

//cast-up and cast-out qualifiers
cast-up-qualifier:  This() → Super()
cast-up-qualifier:
  Super() → CastRef(<get-type; ns-up> Super(), This())
cast-up-qualifier:
  CastRef(c, v) → CastRef(<ns-up> c, v)
cast-up-qualifier:  QThis(c) → QSuper(c)
cast-up-qualifier:
  qs@QSuper(t) → CastRef(<get-type; ns-up>qs, QThis(t))
cast-out-qualifier:
  This() → QThis(<get-type; ns-out> This())
cast-out-qualifier:
  QThis(t) → QThis(<get-type; ns-out> t)
```

Figure 7.12 Restoring violated name bindings by creating qualified names.

7.5.5 *Restoring Name Bindings by Creating Qualified Names*

A name binding violation occurs when the original reference of an access term is shadowed by a new name declared in an inner scope. The name binding can be restored by introducing a qualified name that forces the lookup in a more general namespace. The construction of an appropriate qualified name requires contextual information which is propagated during the name analysis traversal. As an example, we discuss name binding repair for the Java sublanguage.

The code fragment in Figure 7.12 implements name binding repair for the Java sublanguage. The `create-qualified-access` rule restores the name bindings of violated access terms. The rule repeatedly creates qualified names that enforce lookup in a more general namespace, until a qualified name is constructed that preserves the original binding. The preservation is checked

by applying the `is-binding-violation` rule discussed before. As with the name lookup strategy (Figure 7.10), the "outwards upwards" motion is followed. The `cast-up` and `cast-out` rules implement the construction of a new qualified name that targets the next, more general namespace.

7.6 EVALUATION

We evaluated the name binding preservation techniques by implementing rename refactorings for different languages, namely Mobl (Hemel and Visser, 2011), Stratego (Visser, 2004) and the subset of Java discussed in this chapter. In the case of Mobl and Stratego, we applied the preservation criterion of Section 7.4, reusing the name analysis defined in the existing compilers, (Hemel, 2010) and (Kats, 2008) respectively. For the Java subset we implemented the compiler from scratch. We applied the preservation technique discussed in Section 7.5.5 which offers support for restoring name bindings by creating qualified names. In this section we report on our experience, focussing on coverage (Section 7.6.1), correctness (Section 7.6.2) and performance (Section 7.6.3).

7.6.1 *Coverage*

To see how well our techniques cover different language features, we implemented rename refactorings for the Java, Mobl and Stratego compilers. The implementations show that the preservation condition is applicable to languages with statically known name bindings. The technique is not applicable to dynamic languages for which the name bindings are not statically known. To realize name binding preservation for Java we had to cope with method overriding and hence dynamic dispatch. Changes introduced in the method names of sub types may affect dynamic name bindings without changing static name bindings. We implemented an additional semantic condition that checks if a refactoring changes the overriding structure of the program in which case a warning is reported.

7.6.2 *Correctness*

To evaluate the correctness of our approach, we manually tested rename refactorings on existing projects in Mobl and Stratego. In addition, we implemented test suites that cover critical cases for Mobl, Stratego and the Java sublanguage. All test inputs were statically correct programs. The test results confirm the correctness of the preservation criterion. Finally, we implemented an automated test strategy that uses an inverse oracle (Daniel et al., 2007) to test whether the name bindings are preserved. First, a list of potential harmful names is created by collecting all names that appear in the program. Then, rename refactorings are applied to all names in the program with the new name randomly chosen from the set of potential harmful names. The renaming is only applied in case no binding violations are detected. As a last step, re-

name refactorings are applied that revert the names of all declarations to their original name. The inverse oracle states that the resulting tree is equal to the original tree modulo annotations. We successfully applied this automated test strategy to evaluate the correctness of the preservation criterion of Section 7.4 on the Stratego compiler front end written in Stratego. During the application of the test strategy, 484 renamings were applied, while 196 renamings were rejected because of binding violations.

7.6.3 *Performance*

The final important question to evaluate is if the preservation technique is practical with respect to performance. The overall performance of the preservation criterion depends on the performance of the following steps; first the possible affected ASTs before the refactoring are analyzed, then the same ASTs after the refactoring are analyzed, next the binding annotations of the endangered names are collected, and finally the binding patterns of the collected annotations are compared. Name analysis is the most expensive step since it requires a multi-stage traversal over the relevant ASTs. The performance of name analysis is discussed in (Konat, 2012).

To experiment with the performance, we applied a number of renamings on the source code of the Mobl compiler, which consists of about 8000 lines of Stratego code. The response times for individual refactorings largely depend on the number and size of the ASTs that are possibly affected, i.e. ASTs that contain terms equal to either the selected reference name or to the new name provided by the user. We selected the relevant ASTs by inspecting cached ASTs. Unfortunately, the cached ASTs could not be used for the refactoring since they missed the origin information required for the source code reconstruction algorithm discussed in Chapter 6. We therefore had to reparse and reanalyze the ASTs before applying the refactoring transformation. Still, our experience showed that the approach scales to this size of projects. We did not yet do performance tests on larger programs.

7.7 RELATED WORK

This chapter presents two language-parametric techniques to name binding preservation. The techniques are defined within the paradigm of strategic term rewriting and are implemented in Stratego. First, we describe a preservation criterion which reuses an existing name analysis defined in the compiler of the language to check whether the original binding structure still holds for the program after transformation. Secondly, we show that name analysis can be defined as a reusable traversal strategy that is applied to set binding annotations, to check bindings and to restore bindings by creating qualified names. These techniques offer an efficient and reliable solution; efficient because the refactoring developer does not have to implement complex conditions for name binding preservation, reliable because the semantics of

the language is implemented only once, with the compiler being the single source of truth.

Precondition approaches Behavior preservation of refactorings has been a primary concern in refactoring research. Opdyke (Opdyke, 1992) and Roberts (Roberts, 1999) propose a precondition based approach. Preconditions specify which conditions a program has to meet for the refactoring to be correct. Limitations of a precondition based approach are pointed out in (Schäfer et al., 2008). Complex scope nesting rules common in current languages make it hard to define sufficient preconditions, furthermore, additional preconditions have to be implemented in case the language evolves. A second limitation is that preconditions are often too strong, preventing refactorings that could in fact be carried out with only some small modifications. The mentioned paper gives examples where widely used refactoring tools (Eclipse[1], NetBeans[2], and IntelliJ[3]) admitted unsound rename refactorings where names did not bind to the correct declarations after the renaming.

JastAdd JastAdd (Ekman and Hedin, 2007) implements an approach based on reference attribute grammars (Hedin, 2000) that allow to express name analysis in a concise and modular manner. The JastAdd approach to name binding preservation (Schäfer et al., 2008) is based on the idea of inverted lookup functions. The inverted lookup functions compute names for variable accesses that bind to a given declaration. The access computation can be tailored to create qualified names, allowing the refactoring to proceed where otherwise a conflict would occur. After a refactoring transformation is performed, all endangered accesses are updated so that they resolve to the same declaration as before. If the updating fails, the refactoring is rejected and all changes are undone. The inverted lookup functions closely follow the lookup functions that specify the name analysis, as noticed in the paper, the size of the access computation code compares to the size of the lookup code. The correspondence between lookup and access computation helps to avoid many pitfalls overseen in precondition based approaches and to adjust the access computation when new language features are introduced.

JunGL JunGL (Verbaere et al., 2006) is a domain-specific language for implementing refactorings. The refactorings manipulate a graph representation of the program with user-defined edges for static semantic properties such as name binding and control-flow. The semantic analyses required for refactorings are expressed by path queries, which are also applied to create qualified access names for field declarations that are shadowed by the renamed variable.

EMFText EMFText (Heidenreich et al., 2009) is an Eclipse plugin that supports the definition of a syntax for textual languages described by an Ecore metamodel. EMFText offers a generic refactoring framework based on role models (Reimann et al., 2010). Transformation steps and generic precondi-

[1] `http://www.http://eclipse.org/`.
[2] `http://www.http://netbeans.org/`.
[3] `http://www.jetbrains.com/idea/`.

tions are defined generically in a meta-metamodel that contains structural commonalities of object-oriented models (e.g., classes, methods, attributes and parameters). Generic refactorings are adapted to a target meta model, by defining a role mapping between the elements in the meta model and the elements of the meta-meta model. The generic refactoring framework provides extension points to add additional constraints that enforce behavior preservation of the refactoring.

Formal specification Garrido and Meseguer (2006) present a formal approach to the specification and mechanised verification of refactorings. Refactorings are specified formally, with conditional rewrite rules in the form of executable Maude equations. The refactoring specifications extend the equational semantics of the language at hand. Starting from a formal specification of the Java semantics, the paper provides detailed correctness proofs for behavior preservation of two Java refactorings. According to the authors, the approach can be used in conjunction with any language for which an equational semantics has been provided. (Sultana and Thompson, 2008) describes the formal verification of refactorings for untyped and typed lambda-calculi. The focus of this paper is more on studying the method rather than aiming for a more complex object language. The proposed method may be applied to study more realistic languages.

Refactoring tools for functional languages Li and Thompson (2008) present HaRe, a refactoring framework for Haskell. The HaRe framework makes use of the static analysis provided by the Haskell compiler front end Programatica (Hallgren et al., 2004). Transformations and analysis are implemented using Strafunski (Lämmel and Visser, 2003), a library for functional strategic programming in Haskell. Instead of an invariant-based approach, behavior preservation is implemented with help of side conditions and possible compensation strategies in case the conditions are violated.

Lammel (Lämmel, 2002) sketches the idea of a generic refactoring framework that could be instantiated for a variety of languages. The implementation is based on functional strategic programming in Haskell (Lämmel and Visser, 2003), and includes generic transformations and analysis functions as building blocks. The generic functions are parameterized with the language-specific ingredients. The intention of the paper is to investigate the idea of a generic refactoring framework, but its applicability does not seem to go beyond a proof of concept.

7.8 CONCLUSION

The implementation of refactorings for new languages requires considerable effort from language developers. We aim at reducing that effort by using language-parametric techniques. An important requirement for the correct application of refactorings is the preservation of static name bindings. In this chapter we presented a technique to detect violated name bindings, and a technique to restore violated name bindings by introducing qualified names.

Both techniques reuse the name analysis defined in the compiler front end. The techniques are implemented within the paradigm of strategic term rewriting. We successfully applied the techniques to implement Rename refactorings for different languages, which resulted in reliable implementations that required only a small effort in terms of lines of code.

Conclusion

8

In this dissertation we studied research problems in the area of language and IDE engineering. In the first part of this dissertation we investigated generic techniques to recover from syntax errors that occur during interactive editing. In the second part we looked into language-parametric techniques for the implementation of refactoring tools.

We proposed a novel error recovery approach for scannerless generalized parsing. The approach combines a correcting technique based on automatically derived recovery productions with a non-correcting technique that uses indentation to select erroneous regions. To evaluate our recovery approach, we implemented an automated evaluation technique that combines automated generation of erroneous test inputs with automated assessment of the recovered outputs. An extensive evaluation showed that our automated recovery approach performs on par with the hand-crafted parser of the Eclipse JDT in terms of recovery quality. Scannerless generalized parsing is essential for parsing composite languages. Error recovery makes it possible to apply this algorithm in an interactive environment. We implemented our recovery approach in the JSGLR parser generator that is used in the Spoofax language workbench.

We investigated language-parametric techniques for the implementation of refactoring support. First, we proposed a generic text reconstruction algorithm that preserves the layout of the original source text. The algorithm also preserves syntactic sugar in case the refactoring is applied to a desugared abstract syntax tree. Secondly, we proposed a technique that reuses the name analysis implemented in the compiler to detect name binding violations caused by the refactoring transformation. We integrated these techniques in a refactoring framework for the Spoofax language workbench.

In the remainder of this chapter we give a summary of the core contributions of this dissertation, revisit the research questions posed in the introductory chapter, and provide recommendations for further research in the domain of error recovery and refactoring techniques.

8.1 CONTRIBUTIONS

Each chapter in this thesis lists distinct contributions. We summarize the core contributions below:

Parse error recovery

- A novel approach to error recovery for SGLR based on automated grammar relaxation (Chapter 2).

- A secondary recovery technique that uses layout to select erroneous regions. The selected regions can be repaired by a correcting technique or discarded as a fall back recovery (Chapter 3).

- An automated technique to evaluate the quality of parse error recovery techniques (Chapter 4).

- General techniques for the implementation of editor services that interact with the parse error recovery technique (Chapter 5).

Refactoring techniques

- A text reconstruction algorithm that preserves the layout and syntactic variation used in the original source text (Chapter 6).

- A language-parametric technique to guarantee name binding preservation for refactoring transformations. (Chapter 7).

8.2 RESEARCH QUESTIONS REVISITED

Research Question 1

> *What techniques are needed to efficiently recover from syntax errors with scannerless, generalized parsers?*

A grammar is a finite set of production rules that describe how symbols can be combined to form sentences in a language. A parser for a grammar applies the production rules to construct a derivation for syntactically valid strings. Strings that are syntactically invalid cannot be derived by applying production rules.

In Chapter 2 we showed how grammars can be extended with additional *recover productions* that are automatically derived from the grammar. These productions take the form of *insertion recovery rules* which derive a missing input symbol from the empty string, and *deletion recovery rules*, which discard an erroneous substring by interpreting it as layout. The recover productions make the original grammar more permissive of its inputs, allowing the parser to construct a derivation for strings that are syntactically invalid with respect to the original grammar.

The extended grammars are not only permissive, but also highly ambiguous. Generalized parsers support ambiguous grammars by parsing multiple interpretations in parallel. To cope with the added complexity of grammars with recovery rules, we adapted the parser to apply the recovery rules in an on-demand fashion, using a backtracking algorithm. Evaluation of the technique (Section 2.6) showed that permissive grammars can be efficiently parsed and in most cases provide good or excellent recovery results according to the quality criteria of Pennello and DeRemer (1978).

Research Question 2

What language generic techniques can be used to detect erroneous regions?

A parser that supports error recovery typically operates by consuming tokens (or characters) until an unexpected token is found. At the point of detection of an error, the recovery mechanism is activated. A major problem for error recovery techniques is the difference between the point of detection and the actual location of an error in the source program.

In Chapter 3 we introduced a technique that uses indentation to partition files and identify erroneous regions. The region selection technique improves the quality and performance of the permissive grammar technique. First, by constraining the application of recovery rules to an erroneous region, secondly, by offering region discarding as a fall back recovery solution in case the permissive grammar technique fails. The benefits of using region detection in combination with the permissive grammar technique were demonstrated by practical experiments (Section 3.5).

Research Question 3a

What kinds of syntax errors occur during interactive editing? How are syntax errors typically distributed over a file?

To gain insight into the type and distribution of syntax errors that occur during interactive editing, we performed a statistical analysis on collected edit data for different languages. From this analysis we conclude that syntax errors are typically clustered in one or two lines of code that cover the construct being edited. We also provided a classification of syntax errors that are common for interactive editing. The results are discussed in Section 4.2.

Research Question 3b

How to obtain test inputs for error recovery techniques that are representative for practical editing? How to automate quality assessment of the recovered test outputs?

To obtain representative test inputs for error recovery evaluation, we implemented a mutation based fuzzing technique that seeds syntax errors at random locations in an input file. The syntax errors are implemented by error generation rules that specify how to construct an erroneous fragment from a syntactically correct fragment. We predefined a set of error generation rules that cover common syntax errors that are generic for different languages. The set can be extended by compiler testers to test the recovery of syntax errors that are specific for a given language. The error generation technique is discussed in Section 4.3.

To allow automated quality assessment, the error generation rules are complemented with an oracle generation rule that specifies how to construct the intended recovery for the erroneous fragment. The remaining problem is to define a suitable metric between recovered programs and their oracle programs. We compared four differential oracle metrics based on their accuracy

and on qualitative aspects such as applicability and comprehensibility. We concluded that all evaluated metrics accurately reflect recovery quality. The token based diff metric was considered the preferred metric, since it reflects human intuition and is independent from a particular abstract interpretation defined for a language. For practical reasons we chose tree-edit distance as the metric that we used in our evaluations. Automated quality assessment of recovery outputs is discussed in Section 4.4.

Research Question 4

> *What general techniques can be used to improve the feedback provided by editor services that interact with the parse error recovery technique?*

Error recovery is crucial for interactive editing, since it allows editor services to provide feedback on syntactically incorrect programs. The editor services operate on the recovered AST which provides a speculative interpretation of the intended program. By ensuring that the recovered AST is well-formed, separation of concerns can be achieved. Error recovery is purely performed by the parser, while the editor services take as input a well-formed AST that represents a syntactically correct program.

Editor services should behave robustly in case the recovery technique fails to construct an interpretation for (a part of) the input program. In Section 5.4 we developed techniques to provide the programmer with as much feedback as possible. Instead of scannerless highlighting, which requires a parse tree, fall-back syntax highlighting based on a lexical analysis is used to highlight the keywords in source fragments that miss a recovery interpretation. Furthermore, the user is presented with feedback of errors up to the point of where the parser fails, in addition, the failure location is reported to the user.

While other editor services should behave robustly in the presence of syntax errors, the content completion service almost exclusively targets towards incomplete programs. Essential for this service is the interpretation of the syntactic structure near the completion request location. In Section 5.4.4 we propose special *completion recovery rules* that compensate for missing characters at the suffix of an incomplete construct. The completion recovery rules are only applied near the cursor location. Furthermore, ambiguous completion interpretations are deliberately not resolved since all possible completion interpretations must be taken into account when providing completion suggestions. Evaluation of the completion recovery technique (Section 5.5) showed that it indeed improves the recovery in the content completion scenario.

Research Question 5a

> *What language-parametric techniques can be used to derive the textual transformation from the transformation applied to the abstract syntax tree? How to migrate comments and adjust the whitespace at the edges of the changed fragments?*

In Chapter 6 we described a text reconstruction algorithm that calculates textual changes from changes in the abstract structure. The algorithm uses an

origin tracking technique to relate terms in the transformed AST to terms in the original AST, and terms in the original AST to text fragments. To correct the whitespace around the changed fragments, layout conventions about indentation and separating new lines are copied from the original source text. Comments are migrated together with their associated code constructs, which are identified by applying a set of clearly defined heuristic rules.

Research Question 5b

> *How to extend the text reconstruction algorithm so that it preserves syntactic sugar for refactorings that take as input a desugared AST?*

Refactoring transformations may be preceded by a desugaring transformation that maps constructs in an enriched syntax onto equivalent constructs in the core syntax. To support this scenario, we extended the text reconstruction algorithm so that it also preserves the original syntactic variation of desugared constructs. Section 6.8 describes the adaptations we made. First, we compare the transformed terms to their *desugared* origin term instead of their base origin term; this ensures that the same syntactic variation is used in the comparison, while the detected changes are applied to the original text fragment with the original syntactic variation. Secondly, we made the algorithm robust against desugared terms that miss a one-to-one correspondence with original terms. That is, term changes with missing or conflicting origin relations are applied as a term change on an ancestor term that can be translated consistently into a textual change.

Research Question 6

> *Is it possible to guarantee the preservation of static semantic invariants in term rewriting systems?*

In Section 7.4 we proposed a preservation criterion for statically known name bindings. The criterion assumes an existing name analysis that annotates identifier terms with a unique reference name. Name binding violations are detected by comparing the *intended* binding structure with the *actual* binding structure, both set as annotations on the transformed term. The intended binding structure is constructed by applying the name analysis followed by the refactoring transformation; while the actual binding structure is constructed by first applying the transformation and then applying the name analysis.

Many languages offer the possibility to access variables defined in a namespace via qualified names. A limitation of the preservation criterion is that it rejects refactorings that could be carried out by introducing qualified names for violated name bindings. As a refinement to the preservation criterion, we showed in Section 7.5 that name analysis can be implemented as a reusable traversal strategy that can be applied to set name binding annotations, to detect name binding violations, and to restore name bindings by creating qualified names.

Error recovery We developed a language-independent error recovery technique for scannerless generalized parsing. Generalized parsers support the full set of context-free grammars. However, many languages in practice are not context-free. An important class of such languages is layout-sensitive languages, in which the interpretation of a code fragment depends on its layout. Examples of layout sensitive languages are Python, Haskell and F#.

Recent work by (Erdweg et al., 2012) introduces an extension of scannerless generalized parsing to parse layout sensitive languages. Layout constraints are declared as annotations on the productions of a context-free grammar. The extended SGLR parser enforces these constraints at parse time when possible, while the remaining constraints are validated by a post processor on the generated parse forest.

The combination of layout sensitive parsing and error recovery is yet to be explored. We foresee the following adaptations. First, the recovery framework must be extended to recover from layout errors which may be detected at parse time or at disambiguation time. Secondly, the runtime recovery disambiguation filter must be extended to keep ambiguities that will be disambiguated by the (runtime or post-parse) layout filter. Extending the error recovery mechanism to offer full support for layout sensitive languages is left for future work.

We argued that our correcting recovery technique for SGLR (Chapter 2) can best be applied in combination with a region selection technique. In Chapter 3 we proposed a technique that uses indentation to detect discardable erroneous regions. A promising alternative for parsing in an IDE is to use the parse history to detect regions of code with possible syntax errors (Wagner and Graham, 1997). The affected, possibly erroneous regions can be calculated by comparing the current input with the latest correct input. Beside the application for syntax error recovery, these regions could also be used for incremental parsing which has the potential to significantly reduce the response time during interactive editing. Region detection based on the parser history and incremental parsing in combination with syntax error recovery are interesting directions for future work.

Refactoring techniques As a result of our research on refactoring techniques, we implemented a refactoring framework for the Spoofax language workbench that incorporates the layout preservation and name binding preservation techniques. In addition, the framework introduces a small specification language to declaratively define a refactoring including its user interaction aspects. The framework is described in (de Jonge and Visser, 2013). The objective of the framework is to factor out all language generic aspects of the refactoring workflow into generic framework components, and to implement refactorings that are generically applicable to different languages. Practical experience with the framework is required to answer the question how generically applicable the framework components are, and how well they cover different types of programming languages.

The Spoofax refactoring framework focuses on the implementation of predefined refactorings for end users of a language. An interesting direction for future work is support for the implementation of refactorings for and by language developers. Given the fact that language development in Spoofax involves multiple DSLs, this requires an integrated approach to cross-language analysis and refactoring (Strein et al., 2006). Furthermore, the IDE support for refactorings must implement an open structure that allows the application of user-defined transformations that may target different languages (Li and Thompson, 2012).

Bibliography

Aho, A. and Peterson, T. G. (1972). A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1:305. (Cited on pages 7, 60, and 61.)

Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E., and Reilles, A. (2007). Tom: Piggybacking rewriting on java. In *RTA'07*, pages 36–47. (Cited on page 152.)

Barnard, D. T. and Holt, R. C. (1982). Hierarchic syntax error repair for LR grammars. *International Journal of Computer and Information Sciences*, 11(4):231–258. (Cited on pages 4, 52, 72, and 73.)

Basten, H., Klint, P., and Vinju, J. (2012). Ambiguity detection: Scaling to scannerless. In Sloane, A. and Aßmann, U., editors, *Software Language Engineering*, volume 6940 of *Lecture Notes in Computer Science*, pages 303–323. Springer Berlin Heidelberg. (Cited on page 14.)

Basten, H. and Vinju, J. (2012). Parse forest diagnostics with dr. ambiguity. In Sloane, A. and Aßmann, U., editors, *Software Language Engineering*, volume 6940 of *Lecture Notes in Computer Science*, pages 283–302. Springer Berlin Heidelberg. (Cited on page 14.)

Bischofberger, W. R. (1992). Sniff - a pragmatic approach to a c++ programming environment. In *In USENIX C++ Conference*, pages 67–82. (Cited on page 54.)

van den Brand, M., van Deursen, A., Heering, J., de Jong, H., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., and Visser, J. (2001). The Asf+Sdf Meta-environment: A component-based language development environment. In Wilhelm, R., editor, *Proceedings of the 10th International Conference on Compiler Construction*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer. (Cited on pages 3, 103, and 117.)

van den Brand, M. G. J., Bruntink, M., Economopoulos, G. R., de Jong, H. A., Klint, P., Kooiker, T., van der Storm, T., and Vinju, J. J. (2007). Using the Meta-Environment for maintenance and renovation. In *Proceedings of the 11th European Conference on The European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 331–332. IEEE. (Cited on page 53.)

van den Brand, M. G. J., de Jong, H. A., Klint, P., and Olivier, P. A. (2000). Efficient annotated terms. *Software – Practice & Experience*, 30(3):259–291. (Cited on pages 88 and 156.)

van den Brand, M. G. J., Heering, J., Klint, P., and Olivier, P. A. (2002). Compiling language definitions: the ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368. (Cited on pages 25 and 53.)

Bravenboer, M. (2008). Transforming java with stratego. `http://releases.strategoxt.org/strategoxt-manual/unstable/manual/chunk-part/java-in-stratego.html`. (Cited on page 89.)

Bravenboer, M., Dolstra, E., and Visser, E. (2007). Preventing injection attacks with syntax embeddings. In *GPCE*, pages 3–12. (Cited on page 27.)

Bravenboer, M., Dolstra, E., and Visser, E. (2010). Preventing injection attacks with syntax embeddings. *Science of Computer Programming*, 75(7):473–495. (Cited on pages 24, 27, and 94.)

Bravenboer, M., Kalleberg, K. T., Vermaas, R., and Visser, E. (2008). Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70. (Cited on pages 25, 78, 118, 141, 152, and 153.)

Bravenboer, M., Tanter, E., and Visser, E. (2006a). Declarative, formal, and extensible syntax definition for AspectJ. A case for scannerless generalized-LR parsing. In Cook, W. R., editor, *Proceedings of the 21th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*, volume 41 of *SIGPLAN Notices*, pages 209–228, Portland, Oregon, USA. ACM. (Cited on pages 4, 7, 13, 23, 25, 30, 31, and 40.)

Bravenboer, M., van Dam, A., Olmos, K., and Visser, E. (2006b). Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69(1–2):123–178. (Cited on page 160.)

Bravenboer, M. and Visser, E. (2004). Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In Vlissides, J. M. and Schmidt, D. C., editors, *Proceedings of the 19th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2004)*, volume 39 of *SIGPLAN Notices*, pages 365–383. ACM. (Cited on pages 22, 27, and 31.)

Burke, M. G. and Fisher, G. A. (1987). A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Transactions on Programming Languages and Systems*, 9(2):164–197. (Cited on pages 59 and 72.)

Cantor, D. G. (1962). On the ambiguity problem of backus systems. *JACM*, 9(4):477–479. (Cited on page 14.)

Cerecke, C. (2002). Repairing syntax errors in lr-based parsers. In Oudshoorn, M. J., editor, *ACSC*, volume 4 of *CRPIT*, pages 17–22. Australian Computer Society. (Cited on pages 52 and 59.)

Charles, P., Fuhrer, R. M., Sutton, Jr., S. M., Duesterwald, E., and Vinju, J. (2009). Accelerating the creation of customized, language-specific IDEs in Eclipse. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA 2009)*, volume 44 of *SIGPLAN Notices*, pages 191–206. ACM. (Cited on pages 53 and 117.)

Charles, P., Fuhrer, R. M., and Sutton, Jr., S. M. (2007). IMP: a meta-tooling platform for creating language-specific IDEs in Eclipse. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE 2007)*, pages 485–488. ACM. (Cited on pages 53 and 117.)

Chawathe, S. S., Rajaraman, A., Garcia-Molina, H., and Widom, J. (1996). Change detection in hierarchically structured information. *SIGMOD Rec.*, 25:493–504. (Cited on pages 49, 70, and 88.)

Corchuelo, R., Pérez, J. A., Cortés, A. R., and Toro, M. (2002). Repairing syntax errors in LR parsers. *ACM Trans. Program. Lang. Syst.*, 24(6):698–710. (Cited on pages 4, 52, 72, 77, and 88.)

Crnkovic, G. (2010). Constructive Research and Info-Computational Knowledge Generation. *Model-Based Reasoning in Science and Technology*, pages 359–380. (Cited on page 14.)

Daniel, B., Dig, D., Garcia, K., and Marinov, D. (2007). Automated testing of refactoring engines. In Crnkovic, I. and Bertolino, A., editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the Int. Symposium on Foundations of Software Engineering (ESEC/FSE 2007)*, pages 185–194. ACM. (Cited on pages 16 and 164.)

de Jonge, M. (2000). A pretty-printer for every occasion. In Ferguson, I., Gray, J., and Scott, L., editors, *The International Symposium on Constructing Software Engineering Tools (CoSET2000)*, pages 68–77. University of Wollongong, Australia. (Cited on page 134.)

de Jonge, M. (2002). Pretty-printing for software reengineering. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, page 550, Washington, DC, USA. IEEE Computer Society. (Cited on page 118.)

de Jonge, M., Kats, L. C. L., Visser, E., and Söderberg, E. (2012). Natural and flexible error recovery for generated modular language environments. *ACM Trans. Program. Lang. Syst.*, 34(4):15:1–15:50. (Cited on pages 16 and 17.)

de Jonge, M., Nilsson-Nyman, E., Kats, L. C. L., and Visser, E. (2009). Natural and flexible error recovery for generated parsers. In van den Brand, M., Gasevic, D., and Gray, J., editors, *Proceedings of the Second International Conference on Software Language Engineering (SLE 2009)*, volume 5969 of *Lecture Notes in Computer Science*, pages 204–223. Springer. (Cited on pages 16, 17, 77, and 99.)

de Jonge, M. and Visser, E. (2012a). An algorithm for layout preservation in refactoring transformations. In Aßmann, U. and Sloane, T., editors, *Software Language Engineering, Fourth International Conference, SLE 2011, Braga, Portugal, July, 2011, Revised Selected Papers*. Springer. (Cited on pages 17 and 82.)

de Jonge, M. and Visser, E. (2012b). Automated evaluation of syntax error recovery. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 322–325, New York, NY, USA. ACM. (Cited on page 17.)

de Jonge, M. and Visser, E. (2012c). A language generic solution for name binding preservation in refactorings. In Andova, S. and Sloane, T., editors, *Workshop on Language Descriptions, Tools, and Applications, Proceedings*. (Cited on page 17.)

de Jonge, M. and Visser, E. (2013). Implementing refactorings in the spoofax language workbench. Technical Report TUD-SERG-2013-008, Software Engineering Research Group, Delft University of Technology. (Cited on pages 6, 11, 16, and 174.)

de Vanter, M. L. V. (2001). Preserving the documentary structure of source code in language-based transformation tools. In *1st IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001), 10 November 2001, Florence, Italy*, pages 133–143. IEEE Computer Society. (Cited on pages 118, 136, 141, and 145.)

Dean, T. R., Cordy, J. R., Schneider, K. A., and Malton, A. J. (2001). Using design recovery techniques to transform legacy systems. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 622, Washington, DC, USA. IEEE Computer Society. (Cited on page 148.)

Degano, P. and Priami, C. (1995). Comparison of syntactic error handling in LR parsers. *Software – Practice & Experience*, 25(6):657–679. (Cited on pages 3, 4, 7, 23, 39, 41, 43, 48, 51, 52, 58, 59, 60, 61, 72, 75, 77, and 88.)

van Deursen, A., Klint, P., and Tip, F. (1993). Origin tracking. *Journal of Symbolic Computation*, 15(5/6):523–545. (Cited on pages 10, 119, and 125.)

van Deursen, A. and Kuipers, T. (1999). Building documentation generators. In *IEEE International Conference on Software Maintenance (ICSM 1999)*, page 40. IEEE. (Cited on pages 23, 27, 28, and 54.)

Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A. P. (2006). Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388. (Cited on page 24.)

Eclipse documentation (2010). Eclipse documentation: Astrewrite. `http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/rewrite/ASTRewrite.html`. Eclipse JDT 3.6. (Cited on page 147.)

Efftinge, S. and Voelter, M. (2006). oAW xText: a framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*. (Cited on pages 103, 117, and 152.)

Ekman, T. and Hedin, G. (2007). The jastadd extensible java compiler. In Gabriel, R. P., Bacon, D. F., Lopes, C. V., and Jr., G. L. S., editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 1–18. ACM. (Cited on pages 152 and 166.)

Ekman, T., Schäfer, M., and Verbaere, M. (2008). Refactoring is not (yet) about transformation. In *Proceedings of the 2nd Workshop on Refactoring Tools*, WRT '08, pages 5:1–5:4, New York. ACM. (Cited on pages 11 and 152.)

Erdweg, S., Kats, L. C. L., Rendel, T., Kästner, C., Ostermann, K., and Visser, E. (2011a). Growing a language environment with editor libraries. In Denney, E. and Schultz, U. P., editors, *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE 2011)*. ACM. (Cited on page 16.)

Erdweg, S., Rendel, T., Kästner, C., and Ostermann, K. (2011b). SugarJ: Library-based syntactic language extensibility. In Fisher, K. S., editor, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2011)*, SIGPLAN Notices, Portland, Oregon, USA. ACM. (Cited on page 16.)

Erdweg, S., Rendel, T., Kästner, C., and Ostermann, K. (2012). Layout-sensitive generalized parsing. In *SLE*, pages 244–263. (Cited on page 174.)

Fischer, C. N., Milton, D. R., and Quiring, S. B. (1980). Efficient LL(1) error correction and recovery using only insertions. *Acta Inf.*, 13:141–154. (Cited on pages 4, 52, and 72.)

Ford, B. (2002). Packrat parsing: simple, powerful, lazy, linear time. Functional pearl. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming (ICFP 2002)*, pages 36–47, New York, NY, USA. ACM. (Cited on pages 12, 14, 26, and 53.)

Ford, B. (2004). Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, pages 111–122, New York, NY, USA. ACM. (Cited on page 12.)

Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C., and Schmitt, A. (2007). Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3). (Cited on pages 123 and 148.)

Fowler, M. (2002). Refactoring: Improving the design of existing code. volume 2418 of *Lecture Notes in Computer Science*, page 256. Springer. (Cited on pages 4, 9, and 154.)

Fowler, M. (2005). Language workbenches: The killer-app for domain specific languages? `http://martinfowler.com/articles/languageWorkbench.html`. (Cited on pages 117 and 151.)

Fowler, M. (2011). *Domain-Specific Languages*. Addison Wesley. (Cited on page 151.)

Garrido, A. and Meseguer, J. (2006). Formal specification and verification of java refactorings. *Source Code Analysis and Manipulation, IEEE International Workshop*. (Cited on pages 152 and 167.)

Ginsburg, S. and Ullian, J. (1966). Ambiguity in context free languages. *J. ACM*, 13(1):62–89. (Cited on page 14.)

Graham, S. L., Haley, C. B., and Joy, W. N. (1979). Practical LR error recovery. In *SIGPLAN '79: Symposium on Compiler Construction*, pages 168–175. ACM. (Cited on page 61.)

Grimm, R. (2006). Better extensibility through modular syntax. In *PLDI*, pages 38–51. (Cited on pages 14 and 53.)

Groenewegen, D. M., Hemel, Z., Kats, L. C. L., and Visser, E. (2008). WebDSL: A domain-specific language for dynamic web applications. In Mielke, N. and Zimmermann, O., editors, *Companion to the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008)*, pages 779–780, New York, NY, USA. ACM. (poster). (Cited on pages 16, 49, 70, and 79.)

Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., and Völkel, S. (2008). Monticore: a framework for the development of textual domain specific languages. In Schäfer, W., Dwyer, M. B., and Gruhn, V., editors, *Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume*, pages 925–926. ACM. (Cited on pages 14, 103, and 117.)

Hallgren, T., Hook, J., Jones, M. P., and Kieburtz, R. B. (2004). An overview of the programatica toolset. In *High Confidence Software and Systems Conference, HCSS04, http://www.cse.ogi.edu/ hallgren/ Programatica/HCSS04*. (Cited on page 167.)

Harm, J. and Lämmel, R. (2000). Two-dimensional approximation coverage. *Informatica (Slovenia)*, 24(3). (Cited on page 100.)

Hedin, G. (2000). Reference attributed grammars. *Informatica (Slovenia)*, 24(3):301–317. (Cited on pages 11, 152, and 166.)

Heering, J., Hendriks, P. R. H., Klint, P., and Rekers, J. (1989a). The syntax definition formalism sdf - reference manual. *SIGPLAN*, 24(11):43–75. (Cited on page 3.)

Heering, J., Hendriks, P. R. H., Klint, P., and Rekers, J. (1989b). The syntax definition formalism SDF – reference manual. *SIGPLAN Notices*, 24(11):43–75. (Cited on pages 26, 105, and 133.)

Heidenreich, F., Johannes, J., Karol, S., Seifert, M., and Wende, C. (2009). Derivation and refinement of textual syntax for models. In Paige, R. F., Hartman, A., and Rensink, A., editors, *Proceedings of Model Driven Architecture - Foundations and Applications, 5th European Conference (ECMDA-FA 2009)*, volume 5562 of *Lecture Notes in Computer Science*, pages 114–129. Springer. (Cited on pages 103, 117, 151, and 166.)

Hemel, Z. (2010). Mobl compiler. `https://github.com/eelcovisser/mobl`. (Cited on page 164.)

Hemel, Z., Kats, L. C. L., and Visser, E. (2008). Code generation by model transformation. In *ICMT*, pages 183–198. (Cited on page 160.)

Hemel, Z. and Visser, E. (2011). Declaratively programming the mobile web with mobl. In Fisher, K. S., editor, *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2011)*, SIGPLAN Notices, Portland, Oregon, USA. ACM. (Cited on pages 16, 133, 141, 153, and 164.)

Hevner, A. R., March, S. T., Park, J., Ram, S., and Ram, S. (2004). Design science in information systems research. *MIS Quarterly*. (Cited on page 15.)

Hirzel, M. and Grimm, R. (2007). Jeannie: granting Java Native Interface developers their wishes. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA 2007)*, volume 42 of *SIGPLAN Notices*, pages 19–38, New York, NY, USA. ACM. (Cited on page 12.)

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages and Computation*. Addison Wesley. (Cited on pages 3, 22, and 25.)

Horning, J. J. (1976). Structuring compiler development. In Bauer, F. L. and Eickel, J., editors, *Compiler Construction, An Advanced Course, 2nd ed*, volume 21 of *Lecture Notes in Computer Science*, pages 498–513. Springer. (Cited on pages 8 and 76.)

Hunt, A. and Thomas, D. (2000). *The Pragmatic Programmer*. Addison Wesley. (Cited on page 153.)

Jiang, T., Wang, L., and Zhang, K. (1994). Alignment of trees - an alternative to tree edit. In *CPM '94: Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, volume 807 of *LNCS*, pages 75–86, London, UK. Springer-Verlag. (Cited on page 77.)

Johnson, S. C. (1975). YACC—yet another compiler-compiler. Technical Report CS-32, AT & T Bell Laboratories. (Cited on page 2.)

Johnstone, A., Scott, E., and Economopoulos, G. (2004). Generalised parsing: Some costs. *Lecture Notes in Computer Science*, 2985:89–103. (Cited on page 43.)

Jouault, F., Bézivin, J., and Kurtev, I. (2006). TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th international conference on Generative and Component Engineering (GPCE 2006)*, pages 249–254. ACM. (Cited on pages 103, 117, and 151.)

Kats, L. C. (2008). Stratego IDE. `https://svn.strategoxt.org/repos/StrategoXT/spoofax-imp/trunk/org.strategoxt.imp.editors.stratego`. (Cited on page 164.)

Kats, L. C. L., Bravenboer, M., and Visser, E. (2008). Mixing source and bytecode. A case for compilation by normalization. In Kiczales, G., editor, *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008)*, volume 43 of *SIGPLAN Notices*, pages 91–108, New York, NY, USA. ACM. (Cited on pages 27, 49, 70, and 94.)

Kats, L. C. L., de Jonge, M., Nilsson-Nyman, E., and Visser, E. (2009a). Providing rapid feedback in generated modular language environments. Adding error recovery to scannerless generalized-LR parsing. In Leavens, G. T., editor, *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2009)*, volume 44 of *SIGPLAN Notices*, pages 445–464, New York, NY, USA. ACM. (Cited on pages 8, 16, 17, and 76.)

Kats, L. C. L., Kalleberg, K. T., and Visser, E. (2009b). Domain-specific languages for composable editor plugins. In *Proceedings of the Ninth International Workshop on Language Descriptions, Tools, and Applications (LDTA 2009)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 149–163. Elsevier. (Cited on pages 23 and 53.)

Kats, L. C. L., Kalleberg, K. T., and Visser, E. (2010a). Domain-specific languages for composable editor plugins. *Electronic Notes in Theoretical Computer Science*, 253(7):149–163. (Cited on pages 16, 125, and 152.)

Kats, L. C. L., Sloane, A. M., and Visser, E. (2009c). Decorated attribute grammars: Attribute evaluation meets strategic programming. In de Moor, O. and Schwartzbach, M. I., editors, *Proceedings of the 18th International Conference on Compiler Construction (CC 2009)*, volume 5501 of *Lecture Notes in Computer Science*, pages 142–157. Springer. (Cited on page 106.)

Kats, L. C. L. and Visser, E. (2010). The Spoofax language workbench: rules for declarative specification of languages and IDEs. In Cook, W. R., Clarke, S., and Rinard, M. C., editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada. ACM. (Cited on pages 9, 103, 104, 105, 114, 117, 141, and 160.)

Kats, L. C. L., Visser, E., and Wachsmuth, G. (2010b). Pure and declarative syntax definition: paradise lost and regained. In *Proceedings of the ACM*

international conference on Object oriented programming systems languages and applications (OOPSLA 2010), volume 45 of *SIGPLAN Notices*, pages 918–932, New York, NY, USA. ACM. (Cited on page 13.)

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'07)*, volume 1241 of *LNCS*, pages 220–242. Springer. (Cited on page 24.)

Kitlei, R., Lövei, L., Nagy, T., Horváth, Z., and Kozsik, T. (2009). Layout preserving parser for refactoring in Erlang. *Acta Electrotechnica et Informatica*, 9(3):54–63. (Cited on pages 118 and 145.)

Klint, P. (1993). A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201. (Cited on pages 103, 117, and 145.)

Klint, P., van der Storm, T., and Vinju, J. (2009). Rascal: a domain specific language for source code analysis and manipulation. In *Proceedings of the Ninth International Working Conference on Source Code Analysis and Manipulation (SCAM 2009)*, pages 168–177. (Cited on pages 53, 103, and 117.)

Klusener, S. and Lämmel, R. (2003). Deriving tolerant grammars from a base-line grammar. In *International Conference on Software Maintenance (ICSM 2003)*, pages 179–189. IEEE. (Cited on pages 23, 27, 29, and 54.)

Knuth, D. E. (1965). On the translation of languages from left to right. *Information and control*, 8(6):607–639. (Cited on page 12.)

Konat, G. (2012). *Language-Parametric Incremental and Parallel Name Resolution*. PhD thesis, Technical University Delft. (Cited on page 165.)

Koppler, R. (1997). A systematic approach to fuzzy parsing. *Softw. Pract. Exper.*, 27(6):637–649. (Cited on pages 27, 29, and 54.)

Kort, J. and Lämmel, R. (2003). Parse-tree annotations meet re-engineering concerns. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*. (Cited on pages 118, 143, and 146.)

Kossatchev, A. and Posypkin, M. (2005). Survey of compiler testing methods. *Programming and Computer Software*, 31(1):10–19. (Cited on page 100.)

Krahn, H., Rumpe, B., and Völkel, S. (2007). Efficient editor generation for compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*, technical report TR-38, pages 218–228. University of Jyväskylä. (Cited on page 53.)

Krahn, H., Rumpe, B., and Völkel, S. (2008). Monticore: Modular development of textual domain specific languages. In Paige, R. F. and Meyer,

B., editors, *Objects, Components, Models and Patterns, TOOLS EUROPE 2008*, volume 11 of *Lecture Notes in Business Information Processing*, pages 297–315. Springer. (Cited on pages 26, 53, 103, 117, and 152.)

Kuhn, T. and Thomann, O. (2006). Eclipse corner: Abstract syntax tree. `http://eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html`. (Cited on page 95.)

Lämmel, R. (2002). Towards generic refactoring. *Workshop on Rule-Based Programming*, cs.PL/0203001. informal publication. (Cited on pages 152 and 167.)

Lämmel, R. and Visser, J. (2003). A strafunski application letter. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, PADL '03, pages 357–375, London, UK, UK. Springer-Verlag. (Cited on page 167.)

Lavie, A. and Tomita, M. (1993). GLR* – an efficient noise skipping parsing algorithm for context free grammars. In *Proceedings of the Third International Workshop on Parsing Technologies*, pages 123–134. (Cited on pages 27, 29, and 52.)

Lévy, J.-P. (1971). *Automatic Correction of Syntax Errors in Programming Languages*. PhD thesis, Cornell University, Ithaca, NY, USA. (Cited on pages 4, 7, 51, 60, 72, and 73.)

Lewis II, P. M. and Stearns, R. E. (1968). Syntax-directed transduction. *Journal of the ACM*, 15(3):465–488. (Cited on page 12.)

Li, H. and Thompson, S. (2006). A comparative study of refactoring Haskell and Erlang programs. In Penta, M. D. and Moonen, L., editors, *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, pages 197–206. IEEE. (Cited on page 147.)

Li, H. and Thompson, S. (2012). Let's make refactoring tools user-extensible! In *Proceedings of the Fifth Workshop on Refactoring Tools*, WRT '12, pages 32–39, New York, NY, USA. ACM. (Cited on page 175.)

Li, H., Thompson, S., L?vei, L., Horv?th, Z., Kozsik, T., V?g, A., and Nagy, T. (2006). Refactoring erlang programs. In *The Proceedings of 12th International Erlang/OTP User Conference*, Stockholm, Sweden. (Cited on pages 118 and 145.)

Li, H., Thompson, S., and Reinke, C. (2005). The Haskell Refactorer: HaRe, and its API. In Boyland, J. and Hedin, G., editors, *Proceedings of the 5th workshop on Language Descriptions, Tools and Applications (LDTA 2005)*. (Cited on page 147.)

Li, H. and Thompson, S. J. (2008). Tool support for refactoring functional programs. In Glück, R. and de Moor, O., editors, *Proceedings of the 2008*

ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008, pages 199–203. ACM. (Cited on page 167.)

Litecky, C. R. and Davis, G. B. (1976). A study of errors, error-proneness, and error diagnosis in cobol. *Commun. ACM*, 19:33–38. (Cited on page 100.)

Lohmann, W. and Riedewald, G. (2003). Towards automatical migration of transformation rules after grammar extension. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, page 30, Washington, DC, USA. IEEE Computer Society. (Cited on pages 118 and 145.)

Luttik, B. and Visser, E. (1997). Specification of rewriting strategies. In Sellink, M. P. A., editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF 1997)*, Electronic Workshops in Computing, Berlin. Springer-Verlag. (Cited on page 78.)

Lyon, G. (1974). Syntax-directed least-errors analysis for context-free languages: a practical approach. *Commun. ACM*, 17(1):3–14. (Cited on pages 7 and 60.)

Malgady, R. G. and Krebs, D. E. (1986). Understanding correlation coefficients and regression. *Physical Therapy*, 66:110–120. (Cited on page 89.)

Malton, A., Schneider, K. A., Cordy, J. R., Dean, T. R., Cousineau, D., and Reynolds, J. (2001). Processing software source text in automated design recovery and transformation. In *In Proc. International Workshop on Program Comprehension (IWPC'01*, pages 127–134. IEEE Press. (Cited on page 148.)

Mauney, J. and Fischer, C. N. (1988). Determining the extent of lookahead in syntactic error repair. *ACM Transactions on Programming Languages and Systems*, 10(3):456–469. (Cited on pages 4, 7, 51, 60, 72, and 73.)

McKenzie, B. J., Yeatman, C., and Vere, L. D. (1995). Error repair in shift-reduce parsers. *ACM Trans. Program. Lang. Syst.*, 17(4):672–689. (Cited on pages 4, 52, 59, and 72.)

McPeak, S. and Necula, G. C. (2004). Elkhound: A fast, practical GLR parser generator. In *CC*, pages 73–88. (Cited on page 13.)

Miller, C. and Peterson, Z. N. J. (2007). Analysis of mutation and generation-based fuzzing. Technical report, Independent Security Evaluators. (Cited on page 100.)

Moonen, L. (2001). Generating robust parsers using island grammars. In *Proceedings of the Working Conference on Reverse Engineering (WCRE 2001)*, pages 13–22. IEEE. (Cited on pages 23, 27, 28, and 54.)

Moonen, L. (2002). Lightweight impact analysis using island grammars. In *Proceedings of the 10th IEEE International Workshop of Program Comprehension*, pages 219–228. IEEE. (Cited on pages 23 and 27.)

Nilsson-Nyman, E., Ekman, T., and Hedin, G. (2009). Practical scope recovery using bridge parsing. In Gasevic, D., Lämmel, R., and Wyk, E. V., editors, *Proceedings of the First International Conference on Software Language Engineering (SLE 2008)*, volume 5452 of *Lecture Notes in Computer Science*, pages 95–113. (Cited on pages 8, 73, 76, 77, and 99.)

Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois. (Cited on pages 5, 11, 152, and 166.)

Pai, A. B. and Kieburtz, R. B. (1980). Global context recovery: A new strategy for syntactic error recovery by table-drive parsers. *ACM Transactions on Programming Languages and Systems*, 2(1):18–41. (Cited on pages 4, 7, 52, 60, 72, 73, 77, and 88.)

Parr, T. and Fisher, K. (2011). LL(*): the foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 425–436, New York, NY, USA. ACM. (Cited on pages 13 and 27.)

Parr, T. and Quong, R. (1995). ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience*, 25(7):789–810. (Cited on page 53.)

Pennello, T. J. and DeRemer, F. (1978). A forward move algorithm for LR error recovery. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL 1978)*, pages 241–254, New York, NY, USA. ACM. (Cited on pages 49, 70, 76, 77, 87, 88, 93, 95, 99, and 170.)

Permissive (2011). The permissive grammars project. `http://strategoxt.org/Stratego/PermissiveGrammars`. (Cited on page 40.)

Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., and Yang, X. (2012). Test-case reduction for c compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 335–346, New York, NY, USA. ACM. (Cited on page 100.)

Reimann, J., Seifert, M., and Aßmann, U. (2010). Role-based generic model refactoring. In Petriu, D., Rouquette, N., and Øystein Haugen, editors, *Model Driven Engineering Languages and Systems, 13th International Conference, MODELS 2010*, Lecture Notes in Computer Science. Springer. (Cited on page 166.)

Rekers, J. and Koorn, J. W. C. (1991). Substring parsing for arbitrary context-free grammars. *SIGPLAN Notices*, 26(5):59–66. (Cited on page 52.)

Ripley, G. D. and Druseikis, F. C. (1978). A statistical analysis of syntax errors. *Computer Languages, Systems & Structures*, 3(4):227–240. (Cited on pages 76, 99, and 100.)

Roberts, D. B. (1999). Practical analysis for refactoring. Technical report, Champaign, IL, USA. (Cited on pages 11, 152, and 166.)

Salomon, D. J. and Cormack, G. V. (1989). Scannerless NSLR(1) parsing of programming languages. *SIGPLAN Notices*, 24(7):170–178. (Cited on pages 3, 14, and 25.)

Salomon, D. J. and Cormack, G. V. (1995). The disambiguation and scannerless parsing of complete character-level grammars for programming languages. Technical report, TR 95/06, Dept. of Comp. Sci., University of Manitoba, Winnipeg, Canada. (Cited on pages 3, 14, and 25.)

Sametinger, J. and Schiffer, S. (1995). Design and implementation aspects of an experimental c++ programming environment. In *Software - Practice and Experience. 25 (2) : 111 - 128*, pages 11112–8. (Cited on page 54.)

Saunders, S., Fields, D. K., and Belayev, E. (2006). *IntelliJ IDEA in Action*. Manning. (Cited on page 103.)

Schäfer, M., Ekman, T., and de Moor, O. (2008). Sound and extensible renaming for java. In Harris, G. E., editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 277–294. ACM. (Cited on pages 152, 155, and 166.)

Schäfer, M., Verbaere, M., Ekman, T., and Moor, O. (2009). Stepping stones over the refactoring rubicon. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 369–393, Berlin, Heidelberg. Springer-Verlag. (Cited on pages 5 and 11.)

Schmitz, S. (2006). Modular syntax demands verification. Technical Report I3S/RR-2006-32-FR, Laboratoire I3S, Université de Nice-Sophia Antipolis, France. (Cited on pages 13 and 26.)

Schwerdfeger, A. and Van Wyk, E. (2009a). Verifiable composition of deterministic grammars. In *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM. (Cited on page 12.)

Schwerdfeger, A. C. and Van Wyk, E. R. (2009b). Verifiable composition of deterministic grammars. *SIGPLAN Not.*, 44(6):199–210. (Cited on page 26.)

Scott, E. and Johnstone, A. (2010). Gll parsing. *Electr. Notes Theor. Comput. Sci.*, 253(7):177–189. (Cited on pages 3 and 13.)

Soares, G. L. (2010). Making program refactoring safer. In Kramer, J., Bishop, J., Devanbu, P. T., and Uchitel, S., editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010*, pages 521–522. ACM. (Cited on page 152.)

Spoofax (2011). The Spoofax project. `http://www.spoofax.org/`. (Cited on page 141.)

Stocker, M. (2010). *Scala Refactoring*. PhD thesis, University of Applied Sciences Rapperswil. (Cited on page 148.)

Strein, D., Kratz, H., and Lowe, W. (2006). Cross-language program analysis and refactoring. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, SCAM '06, pages 207–216, Washington, DC, USA. IEEE Computer Society. (Cited on page 175.)

Sultana, N. and Thompson, S. (2008). Mechanical verification of refactorings. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '08, pages 51–60, New York, NY, USA. ACM. (Cited on page 167.)

Swierstra, S. D. and Duponcheel, L. (1996). Deterministic, error-correcting combinator parsers. In *Advanced Functional Programming, Second International School–Tutorial Text*, pages 184–207, London, UK. Springer-Verlag. (Cited on page 53.)

Tai, K.-C. (1978). Syntactic error correction in programming languages. *IEEE Trans. Software Eng.*, 4(5):414–425. (Cited on pages 4, 52, 59, and 72.)

Tip, F., Fuhrer, R. M., Kieżun, A., Ernst, M. D., Balaban, I., and Sutter, B. D. (2011). Refactoring using type constraints. *ACM Trans. Program. Lang. Syst.*, 33(3):9:1–9:47. (Cited on pages 5 and 11.)

Tomita, M. (1988). Efficient parsing for natural language: A fast algorithm for practical systems. *Computational Linguistics*, 14(2). (Cited on pages 3, 13, 22, 25, and 44.)

Tratt, L. (2010). Direct left-recursive parsing expression grammars. Technical Report EIS-10-01, School of Engineering and Information Sciences, Middlesex University. (Cited on page 12.)

Valkering, R. (2007). Syntax error handling in scannerless generalized LR parsers. Master's thesis, University of Amsterdam. (Cited on page 52.)

van den Brand, M., Scheerder, J., Vinju, J. J., and Visser, E. (2002). Disambiguation filters for scannerless generalized LR parsers. In *CC*, pages 143–158. (Cited on pages 26, 28, and 41.)

van den Brand, M. and Vinju, J. (2000). Rewriting with layout. In Kirchner, C. and Dershowitz, N., editors, *Proceedings of RULE*. (Cited on pages 118 and 145.)

van den Brand, M. and Visser, E. (1996). Generation of formatters for context-free languages. *ACM Transactions on Software Engineering Methodology*, 5(1):1–41. (Cited on pages 122 and 134.)

van der Storm, T. (2011). The Rascal Language Workbench. Rapport de recherche. (Cited on pages 103 and 117.)

Verbaere, M., Payement, A., and de Moor, O. (2006). Scripting refactorings with jungl. In Tarr, P. L. and Cook, W. R., editors, *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006*, pages 651–652. ACM. (Cited on page 166.)

Vinju, J. (2005). *Analysis and Transformation of Source Code by Parsing and Rewriting*. PhD thesis, UvA. (Cited on page 146.)

Visser, E. (1995). A family of syntax definition formalisms. In van den Brand, M. G. J. et al., editors, *ASF+SDF 1995. A Workshop on Generating Tools from Algebraic Specifications*, pages 89–126. Technical Report P9504, Programming Research Group, University of Amsterdam. (Cited on page 105.)

Visser, E. (1997a). A case study in optimizing parsing schemata by disambiguation filters. In *International Workshop on Parsing Technologies (IWPT 1997)*, pages 210–224, Boston, USA. Massachusetts Institute of Technology. (Cited on page 26.)

Visser, E. (1997b). Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam. (Cited on pages 3, 14, 22, 25, and 104.)

Visser, E. (1997c). *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam. (Cited on pages 3, 22, 26, 33, 53, 118, and 133.)

Visser, E. (2002). Meta-programming with concrete object syntax. In Batory, D. S., Consel, C., and Taha, W., editors, *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315. Springer. (Cited on page 24.)

Visser, E. (2004). Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer, C. et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag. (Cited on pages 3, 89, 133, 153, and 164.)

Visser, E. (2007). WebDSL: A case study in domain-specific language engineering. In *GTTSE*, pages 291–373. (Cited on page 120.)

Visser, E., Benaissa, Z.-E.-A., and Tolmach, A. P. (1998). Building program optimizers with rewriting strategies. In Felleisen, M., Hudak, P., and Queinnec, C., editors, *Functional programming*, pages 13–26. ACM. (Cited on page 153.)

Waddington, D. and Yao, B. (2007). High-fidelity C/C++ code transformation. *Science of Computer Programming*, 68(2):64–78. (Cited on page 27.)

Wagner, T. A. and Graham, S. L. (1997). History-sensitive error recovery. In *IEEE TRANSACTIONS ON SOFTWARE ENGINEEERING*, volume XX. (Cited on page 174.)

Wyk, E. V. and Schwerdfeger, A. (2007). Context-aware scanning for parsing extensible languages. In Consel, C. and Lawall, J. L., editors, *Generative Programming and Component Engineering, 6th International Conference, GPCE 2007*, pages 63–72, Salzburg, Austria. ACM. (Cited on page 14.)

Zelenov, S. V. and Zelenova, S. A. (2005). Generation of positive and negative tests for parsers. *Programming and Computer Software*, 31(6):310–320. (Cited on page 100.)

Samenvatting

TAALPARAMETRISCHE TECHNIEKEN VOOR TAALSPECIFIEKE EDITORS

– Maartje de Jonge –

Programmeertalen zijn kunstmatige talen die het mogelijk maken om machines aan te sturen. Broncode van computerprogramma's is in één of meer van deze talen geschreven. Er bestaan veel verschillende programmeertalen en er komen nog altijd meer talen bij. De implementatie van programmeertalen vereist veel werk. Ten eerste moet een compiler of interpreter geïmplementeerd worden om programma's in een taal te kunnen uitvoeren op een machine. Ten tweede is het voor het succes van een taal belangrijk om een editor te implementeren die taalspecifieke ondersteuning biedt voor het bewerken van broncode.

Een vereiste voor taalspecifieke editors is het rapporteren van syntactische en semantische fouten in de broncode. Daarnaast bieden deze editors functionaliteit om de structuur van een programma beter te kunnen begrijpen, functionaliteit om te navigeren door deze structuur, en functionaliteit om de structuur te veranderen door middel van voorgeprogrammeerde standaardbewerkingen op de broncode.

Dit proefschrift beschrijft technieken die het implementeren van taalspecifieke editor functionaliteit eenvoudiger maken. Het eerste deel van dit proefschrift presenteert een techniek voor het diagnosticeren en corrigeren van syntaxfouten die typisch zijn voor het interactief bewerken van broncode. Het tweede deel beschrijft technieken voor het implementeren van semi-automatische code modificaties die bedoeld zijn om de kwaliteit van broncode te verbeteren.

Correctietechnieken voor syntaxfouten.

Het ondersteunen van taalspecifieke functionaliteit in editors vereist een taalspecifieke analyse van de broncode. Aan de basis van deze analyse staat de parser. De parser vervult twee taken: 1) het controleren of de broncode syntactisch correct is en 2) het construeren van een gestructureerde representatie van de broncode, ook wel de syntaxisboom genoemd. De syntaxisboom vormt de basis voor verdere analyses en bewerkingen op de broncode, zoals welke uitgevoerd worden door de taalspecifieke editor functies.

Traditioneel worden parsers voor ontwikkelomgevingen handmatig geïmplementeerd. Dit maakt het mogelijk om de parser te optimaliseren, maar daar staat tegenover dat het implementeren en onderhouden van de parser zeer arbeidsintensief kan zijn. Een modernere aanpak is om de parser te genereren met behulp van een parsergenerator. Parsergenerators genereren een complete parser voor een taal op basis van een context-vrije grammatica die de syntactische structuur van de taal beschrijft.

Accurate terugkoppeling tijdens het bewerken van broncode vereist een interactieve parser die een actuele syntaxisboom bouwt, telkens wanneer de broncode verandert. Een probleem bij interactief parseren is dat de broncode vaak syntactisch incorrect is: de programmeur is dan immers bezig met het bewerken van de code. Om syntactische fouten te kunnen diagnosticeren en repareren maken parsers gebruik van correctietechnieken. Een correctietechniek voor syntaxfouten maakt het mogelijk om 1) alle aanwezige syntaxfouten te vinden en te rapporteren en om 2) een syntaxisboom te construeren voor de gecorrigeerde code. De geconstrueerde syntaxisboom maakt het mogelijk om ook voor incorrecte programma's accurate editor ondersteuning te bieden.

In de eerste helft van dit proefschrift presenteren we een nieuwe techniek om syntactische fouten correctie te realiseren voor gegenereerde parsers. De techniek is specifiek gericht op SGLR (Scannerless-Generalized LR) parsers, een parseertechniek die de volledige klasse van context-vrije grammatica's ondersteunt en het daarmee mogelijk maakt om verschillende talen modulair samen te stellen tot nieuwe, gecombineerde talen.

In Hoofdstuk 2 presenteren we een techniek om context-vrije grammatica's uit te breiden met automatisch gegenereerde correctieregels. Deze correctieregels simuleren het invoegen van een missend symbool, of het verwijderen van een misplaatst symbool of woord in de broncode. Om deze regels op een efficiënte manier te kunnen toepassen hebben we de parseertechniek uitgebreid met een heuristiek die verschillende combinaties van correctieregels uitprobeert in de buurt van de locatie waar een syntaxfout gedetecteerd is. De heuristiek is zo gekozen dat het resulterende, gecorrigeerde programma zo veel mogelijk de bestaande code volgt.

In sommige situaties leidt de bovengenoemde techniek niet binnen een acceptabele tijdspanne tot een bruikbare correctie. Daarom presenteren we in Hoofdstuk 3 een tweede techniek die toegepast kan worden als de eerste techniek faalt. Deze techniek parseert incorrecte broncode door de code constructen met syntaxfouten te negeren. Code constructen worden gedetecteerd door te kijken naar de gebruikte indentatie. Deze correctietechniek kan in bijna iedere situatie succesvol toegepast worden, maar resulteert in minder nauwkeurige fout correcties en diagnoses.

We hebben een extensieve evaluatie studie gedaan naar verschillende aspecten van onze correctietechniek. Deze studie toonde aan dat onze automatische techniek toepasbaar is op verschillende talen en dat de kwaliteit van onze correcties vergelijkbaar is met de kwaliteit van de correcties in de veel gebruikte Java editor in de Eclipse ontwikkelomgeving. In tegenstelling tot de Java editor in Eclipse is onze correctietechniek volledig automatisch gegenereerd.

De uitgebreidheid van de studie vereiste een automatische evaluatiemethode die we presenteren in Hoofdstuk 4. De evaluatiemethode combineert het automatisch genereren van testinvoerprogramma's met het automatische beoordelen van de gecorrigeerde uitvoerprogramma's. De invoerprogramma's worden gegenereerd door representatieve syntaxfouten te injecteren in cor-

recte broncode. De gecorrigeerde uitvoerprogramma's worden beoordeeld door de geselecteerde correcties te vergelijken met de optimale correcties voor de geïnjecteerde syntaxfouten. Hierbij hanteren we een metriek op boomstructuren om een uiteindelijke kwaliteitsscore te kunnen vaststellen.

In Hoofdstuk 5 onderzoeken we de praktische toepasbaarheid van de bovenstaande technieken. Het hoofdstuk beschrijft de implementatie van de techniek in een bestaande parsergenerator. Het hoofdstuk beschrijft ook hoe de techniek geïntegreerd kan worden met verschillende editorfuncties die direct afhankelijk zijn van de foutcorrectietechniek. Tot slot beschrijft het hoofdstuk een uitbreiding op de techniek die de foutcorrectie in de buurt van de cursor locatie verbetert. Deze uitbreiding is specifiek bedoeld om de editor functie te verbeteren die het automatisch aanvullen van code constructen en variabele namen ondersteunt.

Refactoring technieken.

De onderhoudbaarheid van computerprogramma's wordt bepaald door de structurele kwaliteit van de broncode. Refactorings zijn bewerkingen die toegepast worden op broncode met als doel om de kwaliteit van de code te verbeteren, zonder daarbij het gedrag van het programma te veranderen. Voorbeelden van refactorings zijn: het hernoemen van een variabele, het groeperen van opdrachten (statements) in een aparte functie en het generaliseren van een functie door het toevoegen van een extra parameter.

Refactorings kunnen handmatig uitgevoerd worden door de programmeur, maar dit vereist werk en brengt een risico op fouten met zich mee. Moderne editors bieden de mogelijkheid om veelvoorkomende refactorings automatisch op de broncode toe te passen. De programmeur kiest een voorgedefinieerde refactoring uit een lijst en verstrekt eventuele aanvullende informatie in een dialoogvenster. Vervolgens wordt er een voorbeeldweergave getoond van de broncode veranderingen en worden eventuele fouten gerapporteerd. Indien de programmeur de refactoring accepteert, worden de getoonde veranderingen automatisch toegepast op de broncode.

Refactorings zijn specifiek voor een bepaalde taal. Het implementeren van refactorings voor nieuwe talen is moeilijk en arbeidsintensief. Refactoring implementaties moeten niet alleen de bedoelde structurele bewerking specificeren, maar ook: de interactie met de gebruiker afhandelen, controleren of er geen fouten geïntroduceerd worden en de layout van de veranderde fragmenten corrigeren zodat deze overeenkomt met de layout die gehanteerd wordt in de rest van het programma. In de tweede helft van deze thesis presenteren we taalparametrische technieken voor het implementeren van refactorings.

Refactorings vereisen een complexe analyse en structurele modificatie op de structuur van broncode. Deze worden typisch geïmplementeerd op een abstracte syntaxisboom die de broncode representeert. Abstracte syntaxisbomen maken de programmastructuur expliciet en abstraheren over de specifieke layout van broncode. Echter, het eindresultaat van een refactoring is een modificatie op de concrete broncode, inclusief layout. Hoofdstuk 6 presenteert een algoritme om structurele veranderingen op de abstracte boomstructuur te vertalen naar tekstuele veranderingen op de concrete broncode. Vervolgens

wordt dit algoritme uitgebreid met een methode om de layout van gemodificeerde fragmenten aan te passen aan de layout van omringende fragmenten en met een heuristische methode voor het migreren van commentaar.

Refactorings mogen geen onverwachte effecten hebben die ervoor zorgen dat het resulterende programma ongeldig is of ander gedrag vertoont dan het oorspronkelijke programma. Een voorbeeld van een dergelijke situatie is wanneer een refactoring een nieuwe variabelenaam introduceert die conflicteert met een bestaande variabelenaam. In dat geval kan de naambinding van de bestaande variabele of de naambinding van de nieuw geïntroduceerde variabele overschreven worden. Hoofdstuk 7 presenteert een techniek om naambindingen te controleren en, waar mogelijk, te herstellen. De techniek maakt gebruik van de naambinding analyse die geïmplemnteerd is in de compiler voor de taal en is daarmee generiek toepasbaar op verschillende talen.

Curriculum Vitae

Maartje de Jonge

08 november 1979
Geboren te Amsterdam

1992–1997
Gymnasium diploma
Barlaeus Gymnasium te Amsterdam

1997–2004
M.A. in Philosophy
Utrecht University
Department of Philosophy

1997–2005
M.Sc. in Mathematics
Utrecht University
Department of Mathematics

2005–2006
Teacher in Mathematics
Hogeschool Utrecht

2004–2008
Software Developer
Operator Group Delft

2009–2013
Ph.D. in Computer Science
Delft University of Technology
Department of Software Technology

2013
Software Developer
NIPO Software